Journée Vue.js

Jaime Arias
arias@lipn.univ-paris13.fr

Jaime Arias

- CNRS Research Engineer at LIPN
- Responsible of the `dev-team`
- Member Collège Codes Sources et Logiciels
- Ambassador Sofware Heritage

arias@lipn.univ-paris13.fr

https://www.jaime-arias.fr

Disclaimer

This presentation will present the essentials of vuejs and will not cover technical aspects.

Introduction

What is Vue.js?

The Progressive JavaScript Framework

- An **approachable**, **performant** and **versatile** framework for building web user interfaces.
  - **Approachable**: Builds on top of standard HTML, CSS and JavaScript with intuitive API and world-class documentation.
  - **Performant:** Truly reactive, compiler-optimized rendering system that rarely requires manual optimization.
  - **Versatile:** A rich, incrementally adoptable ecosystem that scales between a library and a full-featured framework.

What is Vue.js ?

Timeline

v0.6
Dec 8, 2013
Evan You releases the first version of Vue.js

v1.0 (Evangelion)
Oct 26, 2015
First stable version

v2.0 (Ghost in the Shell)
Sep 30, 2016
Major rewrite

v3.0 (One Piece)
Sep 30, 2016
Composition API & TypeScript support

Vue 2 EOL
Dec 31, 2023
End of Support

Evan You ✓
@youyuxi
Follow

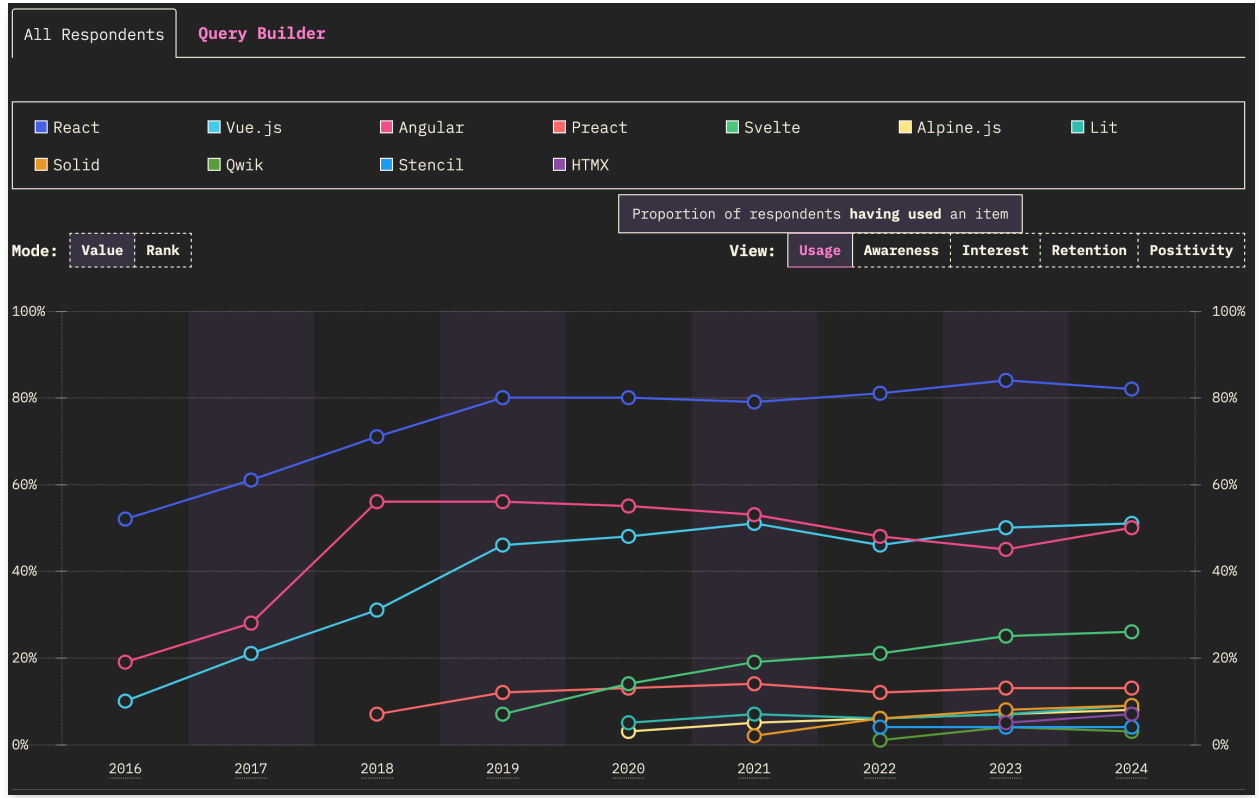
Husband / Father of two / Founder @voidzerodev / Creator @vuejs & @vite_js. Chinese-only alt: @yuxiyou

Singapore | voidzero.dev | 2010

1 735 Following **285,1 k** Followers

Why Vue.js?

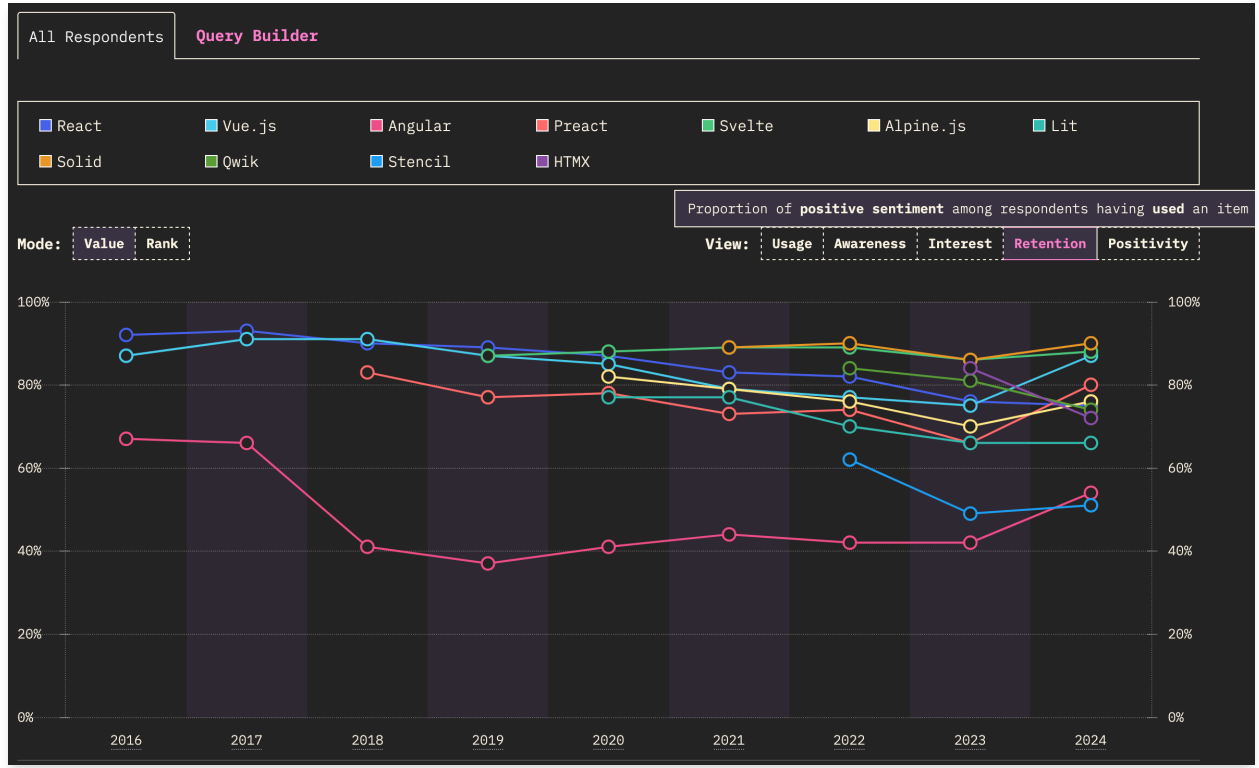
Top JS Frameworks Used in 2024



https://2024.stateofjs.com/en-US/libraries/front-end-frameworks/

Why Vue.js?

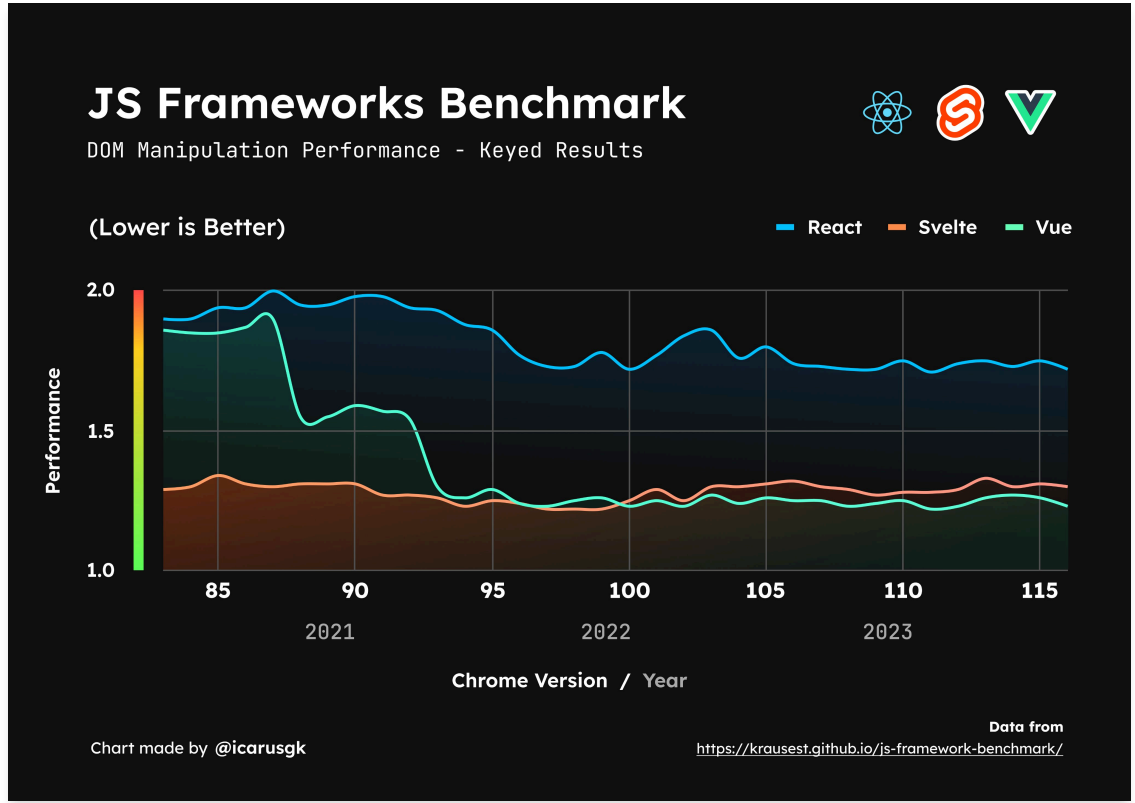
Favourable Opinion per Framework



https://2024.stateofjs.com/en-US/libraries/front-end-frameworks/

Why Vue.js?

JS Frameworks Benchmarks



https://x.com/icarusgkx/status/1693620200543862852

Setting up the Environment

Installation

Node.js

```
# Download and install nvm:
❯ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.2/install.sh | bash

# in lieu of restarting the shell
❯ \. "$HOME/.nvm/nvm.sh"

# Download and install Node.js:
❯ nvm install 24

# Verify the Node.js version:
❯ node -v # Should print "v24.0.2".
❯ nvm current # Should print "v24.0.2".

# Verify npm version:
❯ npm -v # Should print "11.3.0".
```

Installation

Vue.js

This command will install and execute `create-vue`, the official Vue project scaffolding tool.

```
❯ npm create vue@latest
```

Installation

Vue.js

```
┌  Vue.js - The Progressive JavaScript Framework
│
◆  Project name (target directory):
│  <your-project-name>
└
```

Installation

Vue.js

```
Vue.js - The Progressive JavaScript Framework

◇ Project name (target directory):
  vue-project

◆ Select features to include in your project:
  (↑/↓ to navigate, space to select, a to toggle all, enter to confirm)
  □ TypeScript
  □ JSX Support
  ■ Router (SPA development)
  □ Pinia (state management)
  □ Vitest (unit testing)
  □ End-to-End Testing
  ■ ESLint (error prevention)
  ■ Prettier (code formatting)
```

Installation

Vue.js

```
Vue.js - The Progressive JavaScript Framework

◇ Project name (target directory):
vue-project

◆ Select features to include in your project:
(↑/↓ to navigate, space to select, a to toggle all, enter to confirm)
□ TypeScript
□ JSX Support
■ Router (SPA development)
□ Pinia (state management)
□ Vitest (unit testing)
□ End-to-End Testing
■ ESLint (error prevention)
■ Prettier (code formatting)
```

Installation

Vue.js

```
Vue.js - The Progressive JavaScript Framework

◇ Project name (target directory):
vue-project

◆ Select features to include in your project:
(↑/↓ to navigate, space to select, a to toggle all, enter to confirm)
□ TypeScript
□ JSX Support
■ Router (SPA development)
□ Pinia (state management)
□ Vitest (unit testing)
□ End-to-End Testing
■ ESLint (error prevention)
■ Prettier (code formatting)
```

Installation

Vue.js

```
Vue.js - The Progressive JavaScript Framework

◇ Project name (target directory):
vue-project

◆ Select features to include in your project:
(↑/↓ to navigate, space to select, a to toggle all, enter to confirm)
□ TypeScript
□ JSX Support
■ Router (SPA development)
□ Pinia (state management)
□ Vitest (unit testing)
□ End-to-End Testing
■ ESLint (error prevention)
■ Prettier (code formatting)
```

Installation

Vue.js

```
Vue.js - The Progressive JavaScript Framework

◇ Project name (target directory):
│ vue-project
│
◇ Select features to include in your project:
│ (↑/↓ to navigate, space to select, a to toggle all, enter to confirm)
│ Router (SPA development), ESLint (error prevention), Prettier (code formatting)
│
◆ Install Oxlint for faster linting? (experimental)
│ ○ Yes / ● No
└
```

Installation

Vue.js

```
┌  Vue.js - The Progressive JavaScript Framework
│
◇  Project name (target directory):
│  test
│
◇  Select features to include in your project:
│  (↑/↓ to navigate, space to select, a to toggle all, enter to confirm)
│  Router (SPA development), ESLint (error prevention), Prettier (code formatting)
│
◇  Install Oxlint for faster linting? (experimental)
│  No

Scaffolding project in /Users/himito/vue-project...
│
└  Done. Now run:

   cd test
   npm install
   npm run format
   npm run dev
```

Installation

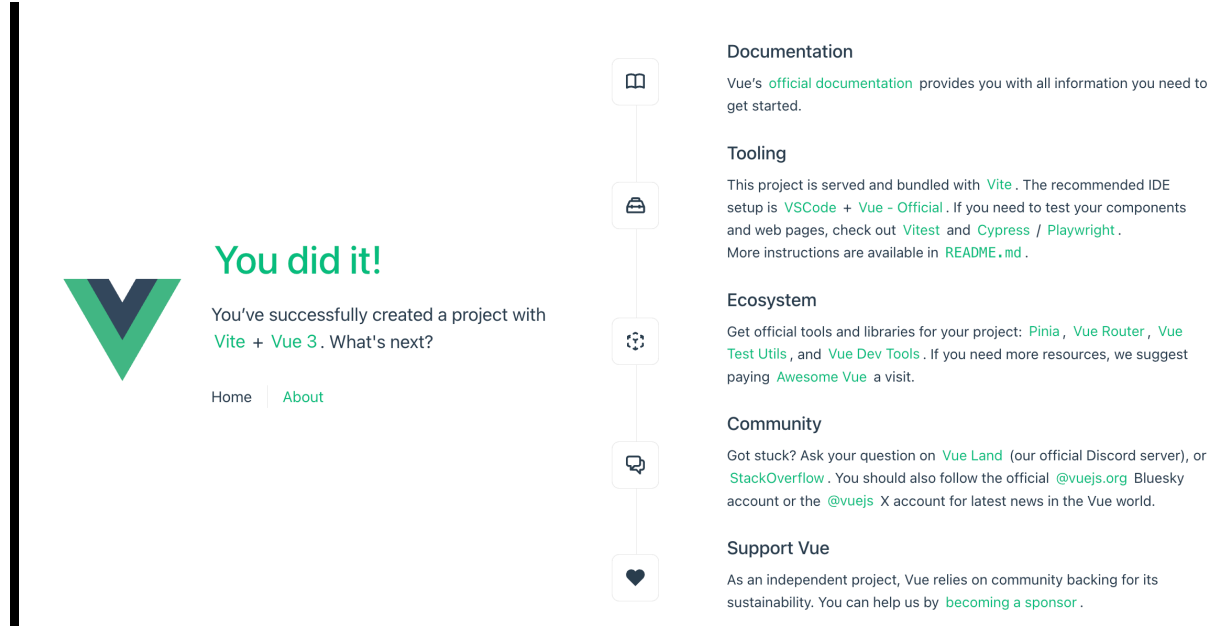
Vue.js

```
❯ npm install
❯ npm run dev
```

Installation

Vue.js

```
❯ npm install
❯ npm run dev
```



Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts
├── index.html
├── jsconfig.json
├── package.json
├── public
├── src
└── vite.config.js
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts    # Configures ESLint for code linting—rules
├── index.html          # The HTML entry point
├── jsconfig.json       # Configure JavaScript language features
├── package.json        # Lists project dependencies, scripts, metadata, and more
├── public              # Static files (favicon, robots.txt, etc.)
├── src                 # Main source code lives here
└── vite.config.js      # Vite's config file (tool that runs and builds the app)
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts     # Configures ESLint for code linting—rules
├── index.html           # The HTML entry point
├── jsconfig.json        # Configure JavaScript language features
├── package.json         # Lists project dependencies, scripts, metadata, and more
├── public               # Static files (favicon, robots.txt, etc.)
├── src                  # Main source code lives here
└── vite.config.js       # Vite's config file (tool that runs and builds the app)
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts    # Configures ESLint for code linting—rules
├── index.html          # The HTML entry point
├── jsconfig.json       # Configure JavaScript language features
├── package.json        # Lists project dependencies, scripts, metadata, and more
├── public              # Static files (favicon, robots.txt, etc.)
├── src                 # Main source code lives here
└── vite.config.js      # Vite's config file (tool that runs and builds the app)
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts     # Configures ESLint for code linting—rules
├── index.html           # The HTML entry point
├── jsconfig.json        # Configure JavaScript language features
├── package.json         # Lists project dependencies, scripts, metadata, and more
├── public               # Static files (favicon, robots.txt, etc.)
├── src                  # Main source code lives here
└── vite.config.js       # Vite's config file (tool that runs and builds the app)
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts     # Configures ESLint for code linting—rules
├── index.html           # The HTML entry point
├── jsconfig.json        # Configure JavaScript language features
├── package.json         # Lists project dependencies, scripts, metadata, and more
├── public               # Static files (favicon, robots.txt, etc.)
├── src                  # Main source code lives here
└── vite.config.js       # Vite's config file (tool that runs and builds the app)
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts     # Configures ESLint for code linting—rules
├── index.html           # The HTML entry point
├── jsconfig.json        # Configure JavaScript language features
├── package.json         # Lists project dependencies, scripts, metadata, and more
├── public               # Static files (favicon, robots.txt, etc.)
├── src                  # Main source code lives here
└── vite.config.js       # Vite's config file (tool that runs and builds the app)
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts    # Configures ESLint for code linting—rules
├── index.html          # The HTML entry point
├── jsconfig.json       # Configure JavaScript language features
├── package.json        # Lists project dependencies, scripts, metadata, and more
├── public              # Static files (favicon, robots.txt, etc.)
├── src                 # Main source code lives here
└── vite.config.js      # Vite's config file (tool that runs and builds the app)
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts
├── index.html
├── jsconfig.json
├── package.json
├── public
├── src                    # Main source code lives here
│   ├── App.vue
│   ├── assets
│   ├── components
│   ├── main.js
│   ├── router
│   └── views
└── vite.config.js
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts
├── index.html
├── jsconfig.json
├── package.json
├── public
├── src                 # Main source code lives here
│   ├── App.vue         # Root Vue component
│   ├── assets          # Static assets used in the app
│   ├── components      # Reusable Vue components
│   ├── main.js         # App bootstrap file
│   ├── router          # Vue Router setup (routes config, guards, etc.)
│   └── views           # Page-level components
└── vite.config.js
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts
├── index.html
├── jsconfig.json
├── package.json
├── public
├── src                 # Main source code lives here
│   ├── App.vue         # Root Vue component
│   ├── assets          # Static assets used in the app
│   ├── components      # Reusable Vue components
│   ├── main.js         # App bootstrap file
│   ├── router          # Vue Router setup (routes config, guards, etc.)
│   └── views           # Page-level components
└── vite.config.js
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts
├── index.html
├── jsconfig.json
├── package.json
├── public
├── src                 # Main source code lives here
│   ├── App.vue         # Root Vue component
│   ├── assets          # Static assets used in the app
│   ├── components      # Reusable Vue components
│   ├── main.js         # App bootstrap file
│   ├── router          # Vue Router setup (routes config, guards, etc.)
│   └── views           # Page-level components
└── vite.config.js
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts
├── index.html
├── jsconfig.json
├── package.json
├── public
├── src                 # Main source code lives here
│   ├── App.vue         # Root Vue component
│   ├── assets          # Static assets used in the app
│   ├── components      # Reusable Vue components
│   ├── main.js         # App bootstrap file
│   ├── router          # Vue Router setup (routes config, guards, etc.)
│   └── views           # Page-level components
└── vite.config.js
```

Installation

Vue.js

The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts
├── index.html
├── jsconfig.json
├── package.json
├── public
├── src                 # Main source code lives here
│   ├── App.vue         # Root Vue component
│   ├── assets          # Static assets used in the app
│   ├── components      # Reusable Vue components
│   ├── main.js         # App bootstrap file
│   ├── router          # Vue Router setup (routes config, guards, etc.)
│   └── views           # Page-level components
└── vite.config.js
```

Installation

Vue.js

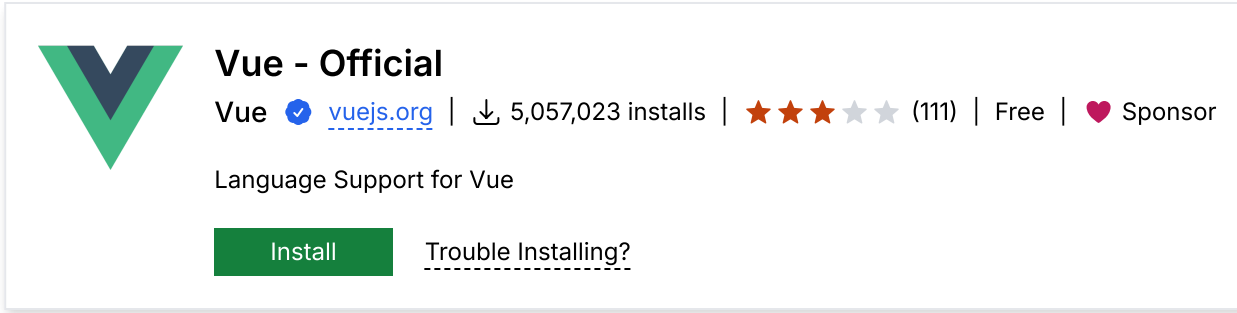
The newly created Vue project has the following folders and files:

```
.
├── eslint.config.ts
├── index.html
├── jsconfig.json
├── package.json
├── public
├── src                 # Main source code lives here
│   ├── App.vue         # Root Vue component
│   ├── assets          # Static assets used in the app
│   ├── components      # Reusable Vue components
│   ├── main.js         # App bootstrap file
│   ├── router          # Vue Router setup (routes config, guards, etc.)
│   └── views           # Page-level components
└── vite.config.js
```

Installation

IDE Support

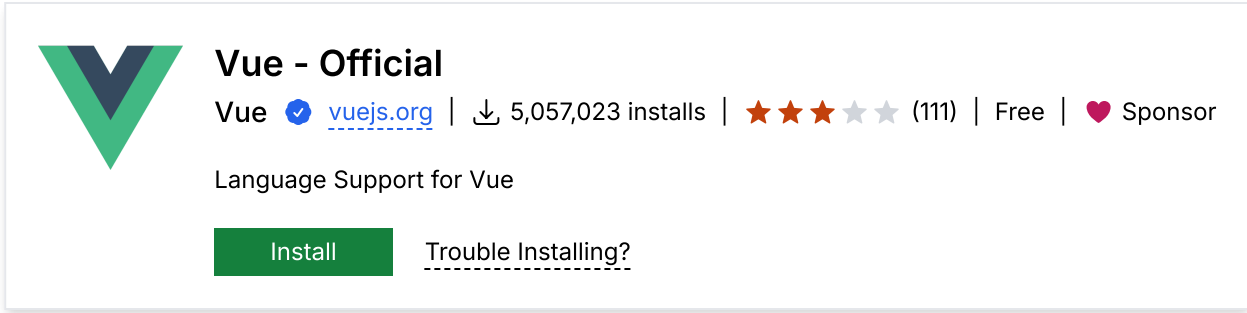
- The recommended IDE setup is VS Code + the `Vue - Official` extension



Installation

IDE Support

- The recommended IDE setup is VS Code + the `Vue - Official` extension

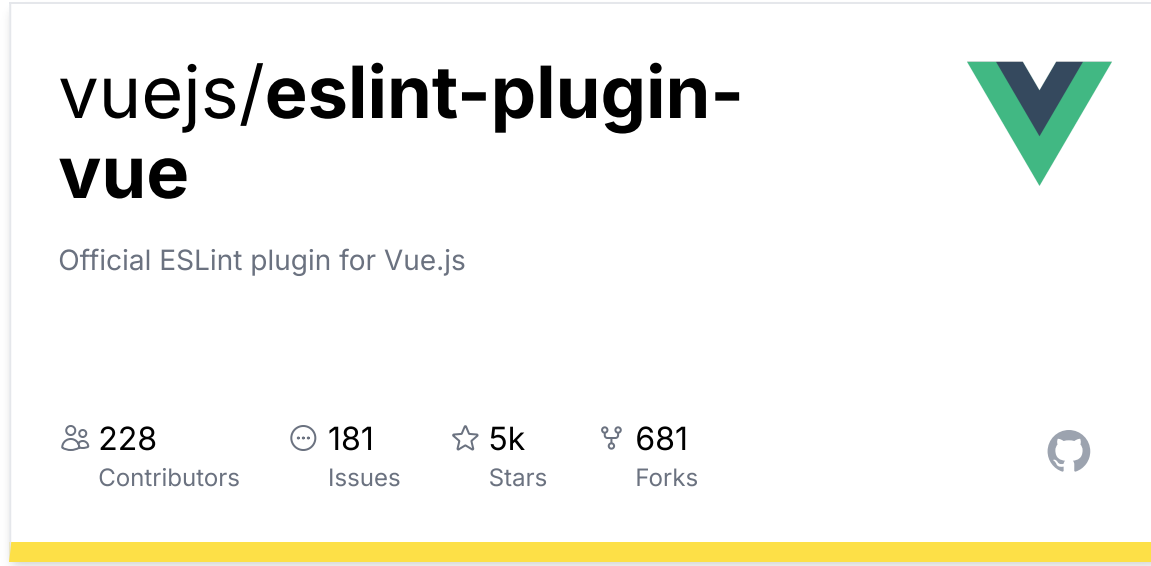


- WebStorm also provides great built-in support for Vue SFCs.
- Other IDEs that support the Language Service Protocol (LSP) can also leverage Volar's core functionalities via LSP:
  - vim / Neovim support via `coc-volar`.

Installation

Linting

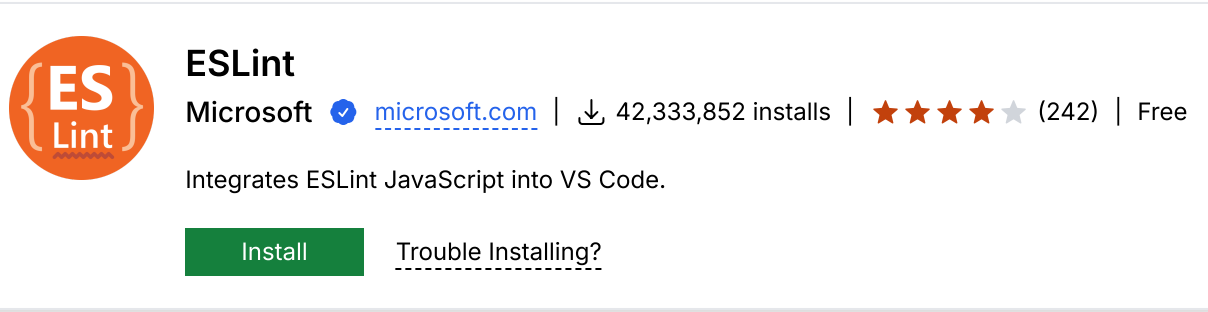
- The Vue team maintains `eslint-plugin-vue`, an ESLint plugin that supports SFC-specific linting rules.



Installation

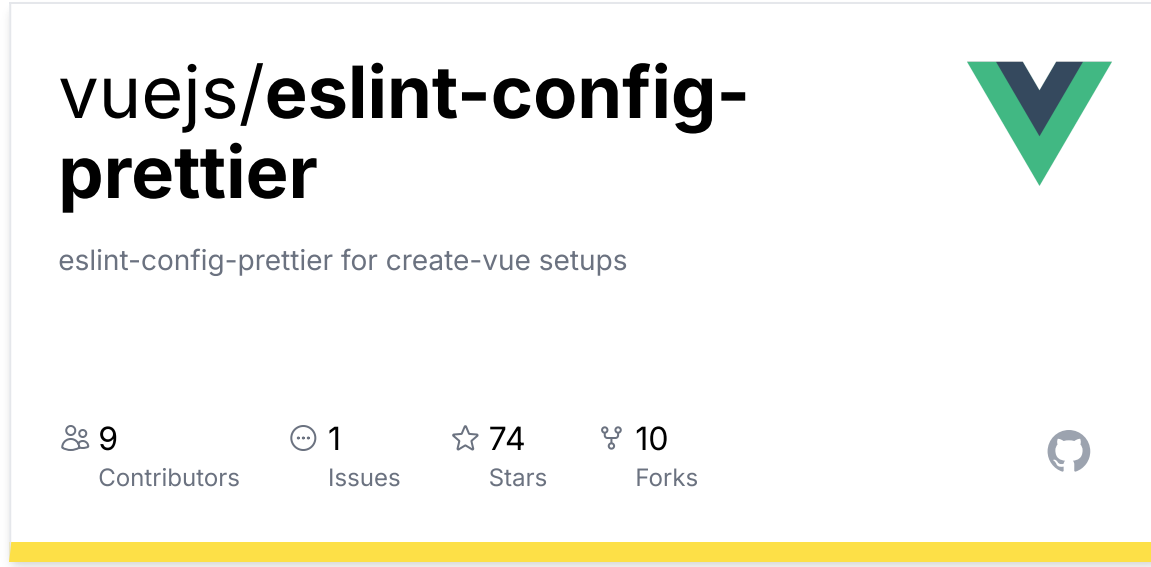
Linting

- The VS Code extension ESLint gives linter feedback right in the editor during development.



Formatting

- The `Vue - Official` VS Code extension provides formatting for Vue SFCs out of the box.
- Prettier provides built-in Vue SFC formatting support.



Installation

Formatting

- The VS Code extension Prettier ensures your code adheres to formatting standards without additional effort.

Prettier - Code formatter

Prettier prettier.io | 56,160,593 installs | (479) | Free | Sponsor

Code formatter using prettier

Install Trouble Installing?

Installation

Tailwind CSS

- A **utility-first CSS framework** for rapidly building custom interfaces.
- Instead of writing custom CSS, you use **predefined utility classes**.
- Designed for **developer productivity and consistency**.

```
<template>
  <div class="p-4 bg-white border-1 border-gray-9 rounded-lg">Hello</div>
</template>
```

Hello

Installation

Tailwind CSS

- Install `tailwindcss` and `@tailwindcss/vite` via `npm`

```
❯ npm install tailwindcss @tailwindcss/vite
```

- Add the `@tailwindcss/vite` plugin to `vite.config.ts`

```ts
import { defineConfig } from 'vite'
import tailwindcss from '@tailwindcss/vite'

export default defineConfig({
  plugins: [
    tailwindcss(),
  ],
})
```

- Add an `@import` to your CSS file that imports Tailwind CSS.

```css
@import "tailwindcss";
```

Core Concepts

Single-Page Application (SPA)

What is an Single-Page Application (SPA)?

An SPA is a web app that loads a single HTML page and dynamically updates the content **without reloading the whole page**.

Single-Page Application (SPA)

What is an Single-Page Application (SPA)?

An SPA is a web app that loads a single HTML page and dynamically updates the content **without reloading the whole page**.

Benefits

- Fast user experience (no full page reloads)
- Smooth navigation like a desktop app
- Ideal for modern web apps (e.g. Gmail, Trello)

Single-Page Application (SPA)

What is an Single-Page Application (SPA)?

An SPA is a web app that loads a single HTML page and dynamically updates the content **without reloading the whole page**.

Benefits

- Fast user experience (no full page reloads)
- Smooth navigation like a desktop app
- Ideal for modern web apps (e.g. Gmail, Trello)

Tradeoffs

- Slower first load (everything loads up front)
- Requires JavaScript to function properly
- SEO can be tricky without special setup

Single-File Components (SFC)

`*.vue` files

- A Vue SFC is a special file format that encapsulates the component's logic (`JavaScript`), template (`HTML`), and styles (`CSS`) in a single file.

```vue
<template>
  <p class="greeting">{{ greeting }}</p>
</template>

<script setup>
import { ref } from 'vue'
const greeting = ref('Hello World!')
</script>

<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

Template Syntax

Text Interpolation

- The most basic form of data binding is text interpolation using the "Mustache" syntax.

```vue
<template>
  <span>Message: {{ msg }}</span>
</template>

<script setup>
  const msg = "Hello World"
</script>
```

```
Message: Hello World
```

Template Syntax

Text Interpolation

- The most basic form of data binding is text interpolation using the "Mustache" syntax.

```
<template>
  <span>Message: {{ msg }}</span>
</template>

<script setup>
  const msg = "Hello World"
</script>
```

```
Message: Hello World
```

- The mustache tag will be replaced with the value of the `msg` property from the corresponding component instance.

Template Syntax

Text Interpolation

- The most basic form of data binding is text interpolation using the "Mustache" syntax.

```vue
<template>
  <span>Message: {{ msg }}</span>
</template>

<script setup>
  const msg = "Hello World"
</script>
```

```
Message: Hello World
```

- The mustache tag will be replaced with the value of the `msg` property from the corresponding component instance.
- It will also be updated whenever the `msg` property changes.

Template Syntax

Raw HTML

- Directives are prefixed with `v-` to indicate that they are special attributes provided by Vue

Template Syntax

Raw HTML

- Directives are prefixed with `v-` to indicate that they are special attributes provided by Vue
- In order to output real HTML, the `v-html` directive is used:

Template Syntax

Raw HTML

- Directives are prefixed with `v-` to indicate that they are special attributes provided by Vue
- In order to output real HTML, the `v-html` directive is used:

```
<template>
  <p><strong>Using text interpolation:</strong> {{ rawHtml }}</p>
  <p><strong>Using v-html directive:</strong> <span v-html="rawHtml"></span></p>
</template>
<script setup>
  const rawHtml = '<span style="color: red">This should be red.</span>'
</script>
```

Using text interpolation: This should be red.

Using v-html directive: This should be red.

Template Syntax

Raw HTML

- Directives are prefixed with `v-` to indicate that they are special attributes provided by Vue
- In order to output real HTML, the `v-html` directive is used:

```
<template>
  <p><strong>Using text interpolation:</strong> {{ rawHtml }}</p>
  <p><strong>Using v-html directive:</strong> <span v-html="rawHtml"></span></p>
</template>
<script setup>
  const rawHtml = '<span style="color: red">This should be red.</span>'
</script>
```

Using text interpolation: This should be red.

Using v-html directive: This should be red.

Template Syntax

Attribute Bindings

- Mustaches cannot be used inside HTML attributes. The `v-bind` directive is used.

```
<template>
  <div v-bind:id="id"></div>
</template>
<script setup>
  const id="myDiv"
</script>
```

- Vue will keep the element's `id` attribute in sync with the component's `id` property

Template Syntax

Attribute Bindings

- Mustaches cannot be used inside HTML attributes. The `v-bind` directive is used.

```
<template>
  <div :id="id"></div>
</template>
<script setup>
  const id="myDiv"
</script>
```

- Vue will keep the element's `id` attribute in sync with the component's `id` property

Template Syntax

Attribute Bindings

- Mustaches cannot be used inside HTML attributes. The `v-bind` directive is used.

```
<template>
  <!-- vuejs 3.4+: same as v-bind:id="id" -->
  <div v-bind:id></div>
</template>
<script setup>
  const id="myDiv"
</script>
```

- Vue will keep the element's `id` attribute in sync with the component's `id` property

Template Syntax

Attribute Bindings

- Mustaches cannot be used inside HTML attributes. The `v-bind` directive is used.

```
<template>
  <!-- vuejs 3.4+: same as v-bind:id="id" -->
  <div :id></div>
</template>
<script setup>
  const id="myDiv"
</script>
```

- Vue will keep the element's `id` attribute in sync with the component's `id` property

Template Syntax

Attribute Bindings

- Mustaches cannot be used inside HTML attributes. The `v-bind` directive is used.

```
<template>
  <!-- vuejs 3.4+: same as v-bind:id="id" -->
  <div :id></div>
</template>
<script setup>
  const id="myDiv"
</script>
```

- Vue will keep the element's `id` attribute in sync with the component's `id` property
- If the bound value is `null` or `undefined`, then the attribute will be removed from the rendered element.

Template Syntax

Boolean Attributes

- Boolean attributes are attributes that can indicate `true` / `false` values by their presence on an element.

```
<template>
  <button :disabled="isDisabled" class="px-3 py-2 border border-black">
    Follow
  </button>
</template>

<script setup>
  const isDisabled=false
</script>
```

Follow

Template Syntax

Boolean Attributes

- Boolean attributes are attributes that can indicate `true` / `false` values by their presence on an element.

```vue
<template>
  <button :disabled="isDisabled" class="px-3 py-2 border border-black">
    Follow
  </button>
</template>

<script setup>
  const isDisabled=false
</script>
```

Follow

- The `disabled` attribute will be included if `isDisabled` has a **truthy** value or is an empty string.

Template Syntax

Boolean Attributes

- Boolean attributes are attributes that can indicate `true` / `false` values by their presence on an element.

```vue
<template>
  <button :disabled="isDisabled" class="px-3 py-2 border border-black">
    Follow
  </button>
</template>

<script setup>
  const isDisabled=false
</script>
```

Follow

- The `disabled` attribute will be included if `isDisabled` has a **truthy** value or is an empty string.

Template Syntax

Binding Multiple Attributes

- Multiple attributes can be bind to a single element by using `v-bind` **without an argument**.

```
1   <template>
2     <div v-bind="objectOfAttrs"></div>
3   </template>
4
5   <script setup>
6     const objectOfAttrs = {
7       id: 'container',
8       class: 'wrapper',
9       style: 'background-color:green'
10    }
11  </script>
```

Template Syntax

JavaScript Expressions

- Vue supports the full power of JavaScript expressions inside all data bindings.

```
1  <template>
2    {{ number + 1 }}
3  </template>
```

Template Syntax

JavaScript Expressions

- Vue supports the full power of JavaScript expressions inside all data bindings.

```
<template>
  {{ number + 1 }}

  {{ ok ? 'YES' : 'NO' }}
</template>
```

Template Syntax

JavaScript Expressions

- Vue supports the full power of JavaScript expressions inside all data bindings.

```
<template>
  {{ number + 1 }}

  {{ ok ? 'YES' : 'NO' }}

  {{ message.split(' ').reverse().join('') }}
</template>
```

Template Syntax

JavaScript Expressions

- Vue supports the full power of JavaScript expressions inside all data bindings.

```
<template>
  {{ number + 1 }}

  {{ ok ? 'YES' : 'NO' }}

  {{ message.split('').reverse().join('') }}

  <div :id="`list-${id}`"></div>
</template>
```

Template Syntax

JavaScript Expressions

- Vue supports the full power of JavaScript expressions inside all data bindings.

```
<template>
  {{ number + 1 }}

  {{ ok ? 'YES' : 'NO' }}

  {{ message.split('').reverse().join('') }}

  <div :id="`list-${id}`"></div>

  <time :title="toTitleDate(date)" :datetime="date">{{ formatDate(date) }}</time>
</template>
```

- It is possible to call a component-exposed method inside a binding expression.

Template Syntax

JavaScript Expressions

- Each binding can only contain one single expression, i.e., **a piece of code that can be evaluated to a value**

```
<template>
  <!-- this is a statement, not an expression: -->
  {{ var a = 1 }}

  <!-- flow control won't work either, use ternary expressions -->
  {{ if (ok) { return message } }}
</template>
```

Template Syntax

Directives

- Directives are special attributes with the `v-` prefix
- Vue provides a number of built-in directives

Template Syntax

Directives

- Directives are special attributes with the `v-` prefix
- Vue provides a number of built-in directives
- A directive's job is to reactively apply updates to the DOM when the value of its expression changes

```
<template>
  <p v-if="seen">Now you see me</p>
</template>

<script setup>
  const seen=false
</script>
```

Template Syntax

Directives

- Some directives can take an argument, denoted by a **colon** after the directive name.

```
1  <template>
2    <a v-bind:href="url">Homepage</a>
3  </template>
4
5  <script setup>
6    const url="https://lipn.univ-paris13.fr/"
7  </script>
```

Homepage

Template Syntax

Directives

- It is also possible to use a JavaScript expression in a directive.

```
<template>
  <a v-bind:[attrName]="url">Homepage</a>
  <!-- <a :['' + attrName]="url"> ... </a> -->
</template>

<script setup>
  const attrName="href"
  const url="https://lipn.univ-paris13.fr/"
</script>
```

Homepage

Template Syntax

Directives

- It is also possible to use a JavaScript expression in a directive.

```vue
<template>
  <a v-bind:[attrName]="url">Homepage</a>
  <!-- <a :['' + attrName]="url"> ... </a> -->
</template>

<script setup>
  const attrName="href"
  const url="https://lipn.univ-paris13.fr/"
</script>
```

Homepage

- Dynamic arguments are expected to evaluate to a string

Template Syntax

Directives

- The `v-on` directive listens to DOM events, e.g., a click.

```
<template>
  <button v-on:click="clickFn" class="btn-blue">
    Follow
  </button>
</template>

<script setup>
  function clickFn() {
    alert("clicked");
  }
</script>
```

Follow

Template Syntax

Modifiers

- Modifiers are special postfixes denoted by a **dot**, which indicate that a directive should be bound in some special way

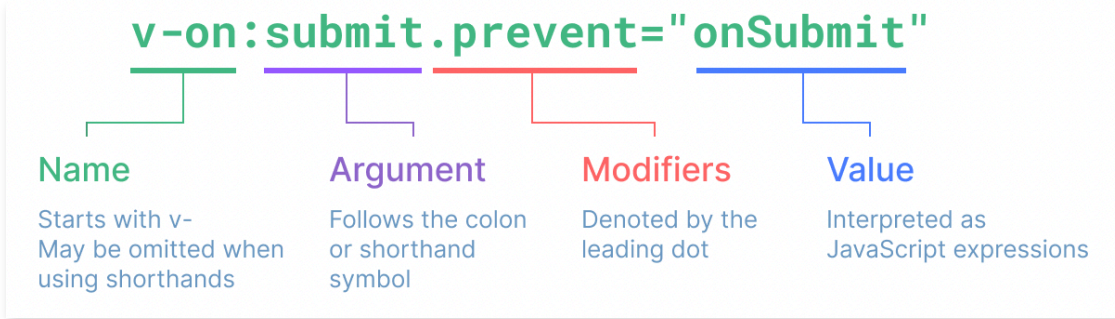
```
<!-- .prevent modifier tells the v-on directive to call event.preventDefault() -->
<form @submit.prevent="onSubmit">...</form>
```

Template Syntax

Modifiers

- Modifiers are special postfixes denoted by a **dot**, which indicate that a directive should be bound in some special way

```
<!-- .prevent modifier tells the v-on directive to call event.preventDefault() -->
<form @submit.prevent="onSubmit">...</form>
```



Composition API

API Styles

Options API

- Component's logic is defined using an object of options such as `data`, `methods`, and `computed`.

```
<script>
export default { }
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

API Styles

Options API

- Component's logic is defined using an object of options such as `data`, `methods`, and `computed`.

```vue
<script>
export default {
  // Properties returned from data() become reactive state
  // and will be exposed on `this`.
  data() {
    return {
      count: 0,
      firstName: 'John',
      lastName: 'Doe'
    }
  },
}
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

API Styles

Options API

- Component's logic is defined using an object of options such as `data`, `methods`, and `computed`.

```vue
<script>
export default {
  data() { /* ... */ },

 // Computed properties are special functions that automatically update when the
 // data they depend on changes
  methods: {
    increment() {
      this.count++
    }
  },
}
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

API Styles

Options API

- Component's logic is defined using an object of options such as `data`, `methods`, and `computed`.

```vue
<script>
export default {
  data() { /* ... */ },
  methods: { /* ... */ },

  // Computed properties are values that are automatically calculated based on
  // other data in your component
  computed: {
    fullName() {
      return this.firstName + ' ' + this.lastName;
    }
  }
}
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

API Styles

Composition API

- Component's logic is defined using imported API functions.
- In SFCs, Composition API is typically used with `<script setup>` which allow to use Composition API with less boilerplate.

```
<script setup>
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

API Styles

Composition API

- Component's logic is defined using imported API functions.
- In SFCs, Composition API is typically used with `<script setup>` which allow to use Composition API with less boilerplate.

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

// reactive state
const count = ref(0)
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

API Styles

Composition API

- Component's logic is defined using imported API functions.
- In SFCs, Composition API is typically used with `<script setup>` which allow to use Composition API with less boilerplate.

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

/** hidden code */

// functions that mutate state and trigger updates
function increment() {
  count.value++
}
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

API Styles

Composition API

- Component's logic is defined using imported API functions.
- In SFCs, Composition API is typically used with `<script setup>` which allow to use Composition API with less boilerplate.

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

/** hidden code */

// computed properties
const fullName = computed(() => `${firstName.value} ${lastName.value}`)
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

Reactivity Fundamentals

`ref()`

Declaring Reactive State with `ref()`

- The recommended way to declare reactive state is using the `ref()` function

```
1  import { ref } from 'vue'
2
3  const count = ref(0)
4  console.log(count)
```

```
RefImpl: {
  "dep": {
    "computed": undefined,
    "version": 0,
    "activeLink": undefined,
    "subs": undefined,
    "map": undefined,
    "key": undefined,
    "sc": 0,
    "subsHead": undefined
  },
  "__v_isRef": true,
  "__v_isShallow": false,
  "_rawValue": 0,
  "_value": 0
}
```

Reactivity Fundamentals

`ref()`

Declaring Reactive State with `ref()`

- `ref()` takes the argument and returns it wrapped within a `ref` object with a `.value` property

```
1  import { ref } from 'vue'
2
3  const count = ref(0)
4
5  count.value++
6  console.log(`count -> ${count.value}`)
```

```
count -> 1
```

Reactivity Fundamentals

`ref()`

Declaring Reactive State with `ref()`

- Refs can hold **any value type**, including deeply nested objects, arrays, or JavaScript built-in data structures like `Map`.
- A `ref` will make its value deeply reactive.

```
1  import { ref } from 'vue'
2
3  const obj = ref({
4    nested: { count: 0 },
5    arr: ['foo']
6  })
7
8  obj.value.nested.count++
9  console.log(obj.value)
```

Reactivity Fundamentals

ref()

Declaring Reactive State with `ref()`

```vue
<script setup>
import { ref } from 'vue'
const counter = ref(1)

function inc() { counter.value++ }
function dec() { counter.value-- }
</script>

<template>
  <div class="text-lg text-white flex items-center justify-between w-full">
    <button class="bg-red-600 p2 hover:bg-red-500" @click="dec">-1</button>
    <span class="text-orange-400 text-4xl">{{ counter }}</span>
    <button class="bg-green-600 p2 hover:bg-green-500" @click="inc">+1</button>
  </div>
</template>
```

-1 1 +1

Reactivity Fundamentals

reactive()

Declaring Reactive State with `reactive()`

- Unlike a `ref` which wraps the inner value in a special object, `reactive()` makes an object itself reactive.

```
1  import { reactive } from 'vue'
2
3  const state = reactive({ count: 0 })
4
5  state.count++
6  console.log(state)
```

```
{
  "count": 1
}
```

Reactivity Fundamentals

`reactive()`

Declaring Reactive State with `reactive()`

- Unlike a `ref` which wraps the inner value in a special object, `reactive()` makes an object itself reactive.

```
import { reactive } from 'vue'

const state = reactive({ count: 0 })

state.count++
console.log(state)
```

```
{
  "count": 1
}
```

- `reactive()` converts the object deeply: nested objects are also wrapped with `reactive()` when accessed.

Reactivity Fundamentals

`reactive()`

Declaring Reactive State with `reactive()`

- Reactive objects are JavaScript Proxies and behave just like normal objects.

Reactivity Fundamentals

`reactive()`

Declaring Reactive State with `reactive()`

- Reactive objects are JavaScript Proxies and behave just like normal objects.


```
1  import { reactive } from 'vue'
2
3  const raw = {}
4  const proxy = reactive(raw)
5
6  // proxy is NOT equal to the original.
7  console.log(proxy === raw)
8
9  // calling reactive() on the same object returns the same proxy
10 // console.log(reactive(raw) === proxy)
11
12 // calling reactive() on a proxy returns itself
13 // console.log(reactive(proxy) === proxy)
```

Reactivity Fundamentals

reactive()

Limitations of `reactive()`

- **Limited value types:** it only works for object types (objects, arrays, etc). It cannot hold primitive types such as string, number or boolean.

```
1  import { reactive } from 'vue'
2  let state = reactive({ count: 0 })
3
4  // the above reference ({ count: 0 }) is no longer being tracked
5  // (reactivity connection is lost!)
6  state = reactive({ count: 1 })
```

Reactivity Fundamentals

`reactive()`

Limitations of `reactive()`

- **Limited value types:** it only works for object types (objects, arrays, etc). It cannot hold primitive types such as string, number or boolean.
- **Cannot replace entire object:** we can't easily "replace" a reactive object because the reactivity connection to the first reference is lost.

```
1  import { reactive } from 'vue'
2  let state = reactive({ count: 0 })
3
4  // the above reference ({ count: 0 }) is no longer being tracked
5  // (reactivity connection is lost!)
6  state = reactive({ count: 1 })
```

Reactivity Fundamentals

reactive()

Limitations of `reactive()`

- **Not destructure-friendly:** when we destructure a reactive object's primitive type property, we will lose the reactivity connection

```
1  import { reactive } from 'vue'
2
3  const state = reactive({ count: 0 })
4
5  // count is disconnected from state.count when destructured.
6  let { count } = state
7
8  // it does not affect original state
9  count++
10
11 console.log(state)
```

```
{
  "count": 0
}
```

Reactivity Fundamentals

reactive()

Limitations of `reactive()`

- **Not destructure-friendly:** when we destructure a reactive object's primitive type property, we will lose the reactivity connection

```
1  import { reactive } from 'vue'
2
3  const state = reactive({ count: 0 })
4
5  // the function receives a plain number and won't be able to track changes
6  function increment(value){ value++ }
7  increment(state.count)
8
9  // we have to pass the entire object in to retain reactivity
10 // function incrementObj(obj){ obj.count++ }
11 // incrementObj(state)
12
13 console.log(state)
```

```
{
  "count": 0
}
```

Computed Properties

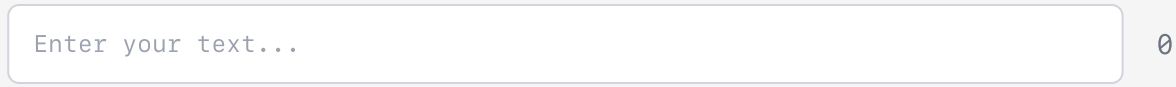
computed()

Don’t repeat yourself (DRY Principle)

- We don’t want to repeat ourselves if we need to include a calculation in the template more than once.

```vue
<script setup>
import { ref } from 'vue'
const newItem = ref("")
</script>

<template>
<div class="flex items-center">
  <input v-model="newItem" placeholder="Enter your text..."
    class="flex-grow p-3 border border-gray-300 rounded-md"/>
  <span class="ml-4 text-sm text-gray-500">{{ newItem.length }}</span>
</div>
</template>
```



Computed Properties

`computed()`

Don’t repeat yourself (DRY Principle)

- For complex logic that includes reactive data, it is recommended to use a **computed property**

Computed Properties

computed()

Don’t repeat yourself (DRY Principle)

- For complex logic that includes reactive data, it is recommended to use a **computed property**
- A computed property automatically tracks its reactive dependencies.

```
<script setup>
import { ref, computed } from 'vue'
const newItem = ref("")
const counter = computed(() => newItem.value.length)
</script>

<template>
<div class="flex items-center">
  <input v-model="newItem" placeholder="Enter your text..."
         class="flex-grow p-3 border border-gray-300 rounded-md"/>
  <span class="ml-4 text-sm text-gray-500">{{ counter }}</span>
</div>
</template>
```

Computed Properties

computed()

Computed Caching vs. Methods

- We can define the same function as a method.

```
1  <script setup>
2  import { ref } from 'vue'
3  const newItem = ref("")
4  const counter = () => newItem.value.length
5  </script>
6
7  <template>
8  <div class="flex items-center">
9    <input v-model="newItem" placeholder="Enter your text..."
10           class="flex-grow p-3 border border-gray-300 rounded-md"/>
11   <span class="ml-4 text-sm text-gray-500">{{ counter() }}</span>
12 </div>
13 </template>
```

Enter your text... 0

Computed Properties

computed()

Computed Caching vs. Methods

- We can define the same function as a method.

```vue
<script setup>
import { ref } from 'vue'
const newItem = ref("")
const counter = () => newItem.value.length
</script>

<template>
<div class="flex items-center">
  <input v-model="newItem" placeholder="Enter your text..."
         class="flex-grow p-3 border border-gray-300 rounded-md"/>
  <span class="ml-4 text-sm text-gray-500">{{ counter() }}</span>
</div>
</template>
```

Enter your text...

0

Computed Properties

computed()

Computed Caching vs. Methods

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

// Used to force a re-render
const { proxy } = getCurrentInstance()
const forceRender = () => proxy.$forceUpdate()

const now = computed(() => new Date().toISOString()) // Computed property
const getNow = () => new Date().toISOString()        // Method
</script>

<template>
  <div class="flex justify-center items-center space-x-6">
    <span><strong>Computed time:</strong> {{ now }}</span>
    <span><strong>Method time:</strong> {{ getNow() }}</span>
    <button @click="forceRender" class="btn-blue">
      Refresh
    </button>
  </div>
</template>
```

Computed Properties

`computed()`

Best Practices

- **Getters should be side-effect free:** its only responsibility should be computing and returning that value

Computed Properties

`computed()`

Best Practices

- **Getters should be side-effect free:** its only responsibility should be computing and returning that value
- **Avoid mutating computed value:** a computed return value should be treated as read-only and never be mutated

Computed Properties

`computed()`

Best Practices

- **Getters should be side-effect free:** its only responsibility should be computing and returning that value
- **Avoid mutating computed value:** a computed return value should be treated as read-only and never be mutated

Watchers

What is a watcher?

- A watcher triggers a callback **whenever** a reactive state changes
- Useful when you want to perform side effects (e.g., react to props or route changes)

```vue
<template>
  <button @click="count++" class="btn-blue">Count + 1</button>
</template>

<script setup>
import { ref, watch } from 'vue'
const count = ref(0)

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

Count + 1

Watchers

Watch Source Types

- The first argument of the `watch` function can be different types of reactive "sources":
  - a ref (including computed refs),
  - a reactive object,
  - a getter function, or
  - an array of multiple sources

```
<script setup>
import { ref, watch } from 'vue'
const x = ref(0)

// a ref
watch(x, (newX) => {
  console.log(`x is ${newX}`)
})
</script>
```

Watchers

Watch Source Types

- The first argument of the `watch` function can be different types of reactive "sources":
  - a ref (including computed refs),
  - a reactive object,
  - a getter function, or
  - an array of multiple sources

```vue
<script setup>
import { ref, watch } from 'vue'
const x = ref(0)
const y = ref(0)

// getter
watch(
  () => x.value + y.value,
  (sum) => {
    console.log(`sum of x + y is: ${sum}`)
  }
```

Watchers

Watch Source Types

- The first argument of the `watch` function can be different types of reactive "sources":
  - a ref (including computed refs),
  - a reactive object,
  - a getter function, or
  - an array of multiple sources

```
<script setup>
import { ref, watch } from 'vue'
const x = ref(0)
const y = ref(0)

// array of multiple sources
watch([x, () => y.value], ([newX, newY]) => {
  console.log(`x is ${newX} and y is ${newY}`)
})
 </script>
```

Watchers

Watch Source Types

- `watch()` does **not** detect changes in properties of reactive objects.

```vue
<script setup>
import { reactive, watch } from 'vue'
const obj = reactive({ count: 0 })

// this won't work because we are passing a number to watch()
watch(obj.count, (count) => {
  console.log(`Count is: ${count}`)
})
</script>
```

Watchers

Watch Source Types

- `watch()` does **not** detect changes in properties of reactive objects.

```vue
<script setup>
import { reactive, watch } from 'vue'
const obj = reactive({ count: 0 })

// instead, use a getter
watch(
  () => obj.count,
  (count) => {
    console.log(`Count is: ${count}`)
  }
)
</script>
```

Watchers

Watch Source Types

- When directly watching a reactive object, the watcher is automatically in `deep mode`

```vue
<script setup>
import { reactive, watch } from 'vue'
const state = reactive({ count: 0 })

watch(state, () => {
  /* triggers on deep mutation to state */
})
</script>
```

Watchers

Watch Source Types

- When directly watching a reactive object, the watcher is automatically in `deep mode`

```vue
<script setup>
import { reactive, watch } from 'vue'
const state = reactive({ count: 0 })

watch(
  () => state,
  (newValue, oldValue) => {
    // newValue === oldValue
  },
  { deep: true }
)
</script>
```

Watchers

Eager Watchers

- By default, the callback of a `watch` won't be called until the watched source has changed

Watchers

Eager Watchers

- By default, the callback of a `watch` won't be called until the watched source has changed
- We can force a watcher's callback to be executed immediately by passing the `immediate: true` option

```
<template>
  <button @click="count++" class="btn-blue">Count + 1</button>
</template>

<script setup>
import { ref, watch } from 'vue'
const count = ref(0)

watch(count,
  (newVal, oldVal) => { alert(`Count changed from ${oldVal} to ${newVal}`) },
  { immediate: true })
</script>
```

Watchers

Once Watchers

- If you want the callback to trigger **only once** when the source changes, use the `once: true` option.

```
<template>
  <button @click="count++" class="btn-blue">Count + 1</button>
</template>

<script setup>
import { ref, watch } from 'vue'
const count = ref(0)

watch(count,
  (newVal, oldVal) => { alert(`Count changed from ${oldVal} to ${newVal}`) },
  { once: true })
</script>
```

Count + 1

List Rendering

Directive `v-for`

- The `v-for` directive is used to render a list of items based on an array

```vue
<template>
  <div class="flex justify-between items-start">
    <ul>
      <li v-for="item in items" class="my-0!">
        {{ item.name }}
      </li>
    </ul>
    <button @click='items.push({name: `Item ${items.length + 1}`})'
            class="btn-blue">Add</button>
  </div>
</template>
<script setup>
  import { ref } from 'vue'
  const items = ref([ { name: 'Item 1' } ])
</script>
```

- Item 1

Add

List Rendering

Directive `v-for`

- It supports an optional second alias for the index of the current item.

```vue
<template>
  <div class="flex justify-between items-start">
    <ul>
      <li v-for="(item, index) in items" class="my-0!">
        {{ index }}: {{ item.name }}
      </li>
    </ul>
    <button @click='items.push({name: `Item ${items.length + 1}`})'
            class="btn-blue">Add</button>
  </div>
</template>
<script setup>
  import { ref } from 'vue'
  const items = ref([ { name: 'Item 1' } ])
</script>
```

- 0: Item 1

Add

List Rendering

Directive `v-for`

- You can use destructuring on the `item` alias.

```vue
<template>
  <div class="flex justify-between items-start">
    <ul>
      <li v-for="({ name }, index) in items" class="my-0!">
        {{ index }}: {{ name }}
      </li>
    </ul>
    <button @click='items.push({name: `Item ${items.length + 1}`})'
            class="btn-blue">Add</button>
  </div>
</template>
<script setup>
  import { ref } from 'vue'
  const items = ref([ { name: 'Item 1' } ])
</script>
```

- 0: Item 1

Add

List Rendering

Directive `v-for`

- You can also use `v-for` to iterate through the properties of an object

```
<template>
  <ul>
    <li v-for="(value, key, index) in book" class="my-0!">
      {{ index }}. <strong>{{ key }}:</strong> {{ value }}
    </li>
  </ul>
</template>
<script setup>
  import { reactive } from 'vue'
  const book = reactive({
    title: 'The Algorithm Design Manual',
    author: 'Steven S. Skiena',
    publishedAt: '2021-07-10'
  })
</script>
```

- 0. **title:** The Algorithm Design Manual
- 1. **author:** Steven S. Skiena

List Rendering

Directive `v-for`

- It can also take an integer to repeat the template *n* times using the range `1...n`

```vue
<template>
  <div class="grid grid-cols-5 gap-4">
    <div v-for="n in 5" :key="n" class="p-4 border border-gray-500 text-center">
      <div class="w-14 h-14 mx-auto bg-gray-300 rounded-full mb-2"></div>
      <p class="text-sm text-gray-600">Item {{ n }}</p>
    </div>
  </div>
</template>
```

List Rendering

Directive `v-for`

- Directives are attached to a single element. We can use the `<template>` element as an invisible wrapper.

```
<template>
  <div class="flex items-center space-x-4">
    <template v-for="user in users" :key="user.id">
      <img class="w-12 h-12 rounded-full" :src="user.avatar" :alt="user.name" />
      <span class="text-xs text-gray-700">{{ user.name }}</span>
    </template>
  </div>
</template>
<script setup>
const users = [
  { id: 1, name: 'Bob', avatar: 'https://placehold.co/100x100?text=Bob' },
  { id: 2, name: 'Eve', avatar: 'https://placehold.co/100x100?text=Eve' },
];
</script>
```

Conditional Rendering

Directive `v-if`

- The block is rendered if the directive's expression is a **truthy** value.
- The `v-else` and `v-else-if` directives can also be used.

```
<template>
  <div class="flex justify-between items-center">
    <span v-if="status === 'success'">Operation was successful!</span>
    <span v-else-if="status === 'error'">An error has occurred.</span>
    <span v-else>No status provided.</span>
    <div class="space-x-1">
      <button class="btn-blue" @click="status = 'success'">Success</button>
      <button class="btn-blue" @click="status = 'error'">Error</button>
    </div>
  </div>
</template>
<script setup>
  import { ref } from 'vue'
  const status = ref("")
</script>
```

Conditional Rendering

Directive `v-show`

- Unlike `v-if`, `v-show` will always be rendered and remain in the DOM. It only toggles the `display` CSS property of the element
- It doesn't support the `<template>` element, nor does it work with `v-else`.

```vue
<template>
  <div class="flex justify-between items-center">
    <button class="btn-blue" @click="isVisible = !isVisible">Toggle</button>
    <span v-show="isVisible">Hello! This message is visible.</span>
  </div>
</template>
<script setup>
  import { ref } from 'vue'
  const isVisible = ref(true)
</script>
```

Toggle

Hello! This message is visible.

Conditional Rendering

`v-if` vs. `v-show`

Conditional Rendering

`v-if` vs. `v-show`

- `v-if` is **"real" conditional rendering** because it ensures that event listeners and child components inside the conditional block are properly destroyed and re-created during toggles.

Conditional Rendering

`v-if` vs. `v-show`

- `v-if` is **"real" conditional rendering** because it ensures that event listeners and child components inside the conditional block are properly destroyed and re-created during toggles.
- `v-if` is also **lazy**: if the condition is `false` on initial render, it will not do anything - the conditional block won't be rendered until the condition becomes `true` for the first time.

Conditional Rendering

`v-if` vs. `v-show`

- `v-if` is **"real" conditional rendering** because it ensures that event listeners and child components inside the conditional block are properly destroyed and re-created during toggles.
- `v-if` is also **lazy**: if the condition is `false` on initial render, it will not do anything - the conditional block won't be rendered until the condition becomes `true` for the first time.
- In comparison, `v-show` is **much simpler** – the element is always rendered regardless of initial condition, with CSS-based toggling.

Conditional Rendering

`v-if` vs. `v-show`

- `v-if` is **"real" conditional rendering** because it ensures that event listeners and child components inside the conditional block are properly destroyed and re-created during toggles.
- `v-if` is also **lazy**: if the condition is `false` on initial render, it will not do anything - the conditional block won't be rendered until the condition becomes `true` for the first time.
- In comparison, `v-show` is **much simpler** – the element is always rendered regardless of initial condition, with CSS-based toggling.
- Generally speaking, `v-if` has higher toggle costs while `v-show` has higher initial render costs.

Conditional Rendering

`v-if` vs. `v-show`

- `v-if` is **"real" conditional rendering** because it ensures that event listeners and child components inside the conditional block are properly destroyed and re-created during toggles.
- `v-if` is also **lazy**: if the condition is `false` on initial render, it will not do anything - the conditional block won't be rendered until the condition becomes `true` for the first time.
- In comparison, `v-show` is **much simpler** – the element is always rendered regardless of initial condition, with CSS-based toggling.
- Generally speaking, `v-if` has higher toggle costs while `v-show` has higher initial render costs.
- So prefer `v-show` if you need to toggle something very often, and prefer `v-if` if the condition is unlikely to change at runtime.

Class Bindings

Binding HTML Classes

- Special enhancements when `v-bind` is used with `class` and `style`.
- We can pass an object to `v-bind:class` to dynamically toggle classes.

```
<template>
  <ul>
    <li v-for="{ label, purchased, highPriority } in items" class="my-0!"
        :class="{'line-through': purchased, 'text-red': highPriority}">
      {{ label }}
    </li>
  </ul>
</template>
<script setup>
  const items = [ { label: "1kg coffee", purchased: false, highPriority: true },
                  { label: "20 cups", purchased: true, highPriority: false} ]
</script>
```

- 1kg coffee
- ~~20 cups~~

Class Bindings

Binding HTML Classes

- The bound object doesn't have to be inline

```vue
<template>
  <span :class="classObject">Hello World!</span>
</template>
<script setup>
  const classObject = {'font-bold': true, 'text-red': false}
</script>
```

Hello World!

Class Bindings

Binding HTML Classes

- We can also bind to a computed property that returns an object.

```
1  <template>
2  |  <span :class="classObject">Hello World!</span>
3  </template>
4  <script setup>
5  | import { ref, computed } from 'vue'
6  | const isActive = ref(true)
7  | const classObject = computed(() => ({'font-bold': isActive.value }))
8  </script>
```

Hello World!

Class Bindings

Binding to Arrays

- `:class` can be bind to an array to apply a list of classes

```
<template>
  <span :class="[activeClass, errorClass]">Hello World!</span>
</template>
<script setup>
  import { ref, computed } from 'vue'
  const activeClass = ref('font-bold')
  const errorClass = ref('text-red')
</script>
```

Hello World!

Class Bindings

Binding to Arrays

- It’s also possible to use the object syntax inside the array syntax

```vue
<template>
  <span :class="[{ [activeClass]: isActive }, errorClass]">Hello World!</span>
</template>
<script setup>
  import { ref, computed } from 'vue'
  const isActive = ref(true)
  const activeClass = ref('font-bold')
  const errorClass = ref('text-red')
</script>
```

Hello World!

Style Bindings

Binding to Objects

- `:style` property supports binding to JavaScript object values.

```
1  <template>
2  |  <span :style="{ 'font-size': fontSize + 'px' }">Hello World!</span>
3  </template>
4  <script setup>
5  |  import { ref, computed } from 'vue'
6  |  const fontSize = ref(18)
7  </script>
```

Hello World!

Style Bindings

Binding to Objects

- `:style` property supports binding to JavaScript object values.

```
<template>
  <span :style="{ 'font-size': fontSize + 'px' }">Hello World!</span>
</template>
<script setup>
  import { ref, computed } from 'vue'
  const fontSize = ref(18)
</script>
```

Hello World!

- `:style` directives can also coexist with regular `style` attributes.

```
<template>
    <span style="color: red" :style="'font-size: 18px'">Hello World!</span>
</template>
```

Hello World!

Style Bindings

Binding to Arrays

- `:style` can be bind to an array of multiple style objects

```
<template>
  <span :style="[baseStyle, activeStyle]">Hello World!</span>
</template>
<script setup>
import { reactive } from 'vue';

const baseStyle = reactive({
  color: 'blue',
});

const activeStyle = reactive({
  fontWeight: 'bold',
  fontSize: '14px'
});
</script>
```

Hello World!

Event Handling

Listening to Events

- The `v-on` directive is used to listen to DOM events and run some JavaScript when they're triggered.
- **Inline handlers:** Inline JavaScript to be executed when the event is triggered.

```
<template>
  <button @click="count++" class="btn-blue">Count is: {{ count }}</button>
</template>

<script setup>
  import { ref } from 'vue'
  const count = ref(0)
</script>
```

Count is: 0

Event Handling

Listening to Events

- **Method handlers:** A property name or path that points to a method defined on the component.

```
<template>
  <button @click="increment" class="btn-blue">Count is: {{ count }}</button>
</template>

<script setup>
  import { ref } from 'vue'
  const count = ref(0)

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

Count is: 0

Event Handling

Listening to Events

- A method handler automatically receives the native DOM Event object that triggers it

```vue
<template>
  <button @click="click" class="btn-blue">Click {{ tag }}</button>
</template>

<script setup>
  import { ref } from 'vue'
  const tag = ref("")

  function click(event){
    tag.value = event.target.tagName
  }
</script>
```

Click

Event Handling

Listening to Events

- Instead of binding directly to a method name, we can also call methods in an inline handler.

```
<template>
  <button @click="say('hello')" class="btn-blue">Say Hello</button>
</template>

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

  function say (message) {
    alert(message)
  }
</script>
```

Say Hello

Event Handling

Listening to Events

- If you need to access the original DOM event in an inline handler:
  1. you can pass it into a method using the special `$event` variable
  2. or use an inline arrow function

```
<template>
  <!-- using $event special variable -->
  <button @click="say('hello', $event)" class="btn-blue">$event</button>

  <!-- using inline arrow function -->
  <button @click="(event) => say('hello', event)" class="btn-blue ml-2">function</
</template>
<script setup>
  import { ref } from 'vue'
  function say (message, event) {
    if (event) { event.preventDefault() }
    alert(message)
  }
</script>
```

Event Handling

Event Modifiers

- Event modifiers are special suffixes that modify the event behaviour without writing extra JavaScript
- Event modifiers are directive postfixes denoted by a `dot` .

Event Handling

Event Modifiers

- Event modifiers are special suffixes that modify the event behaviour without writing extra JavaScript
- Event modifiers are directive postfixes denoted by a dot.

```
<!-- the click event's propagation will be stopped    -->
<!-- i.e., event.stopPropagation()                    -->
<a @click.stop="doSomething"></a>
```

Event Handling

Event Modifiers

- Event modifiers are special suffixes that modify the event behaviour without writing extra JavaScript
- Event modifiers are directive postfixes denoted by a dot.

```
<!-- the click event's propagation will be stopped    -->
<!-- i.e., event.stopPropagation()                    -->
<a @click.stop="doSomething"></a>

<!-- the submit event will no longer reload the page  -->
<!-- i.e, event.preventDefault()                      -->
<form @submit.prevent="submitForm">
  <button type="submit">Submit</button>
</form>
```

Event Handling

Event Modifiers

- Event modifiers are special suffixes that modify the event behaviour without writing extra JavaScript
- Event modifiers are directive postfixes denoted by a dot.

```
<!-- the click event's propagation will be stopped    -->
<!-- i.e., event.stopPropagation()                    -->
<a @click.stop="doSomething"></a>

<!-- the submit event will no longer reload the page  -->
<!-- i.e, event.preventDefault()                      -->
<form @submit.prevent="submitForm">
  <button type="submit">Submit</button>
</form>

<!-- Only triggers if clicked directly on the div (not a child) -->
<div @click.self="doThat">...</div>
```

Event Handling

Event Modifiers

- Event modifiers are special suffixes that modify the event behaviour without writing extra JavaScript
- Event modifiers are directive postfixes denoted by a dot.

```
<!-- the click event's propagation will be stopped    -->
<!-- i.e., event.stopPropagation()                    -->
<a @click.stop="doSomething"></a>

<!-- the submit event will no longer reload the page  -->
<!-- i.e, event.preventDefault()                      -->
<form @submit.prevent="submitForm">
  <button type="submit">Submit</button>
</form>

<!-- Only triggers if clicked directly on the div (not a child) -->
<div @click.self="doThat">...</div>

<!-- the click event will be triggered at most once   -->
<a @click.once="doThis"></a>
```

Event Handling

Event Modifiers: `.stop`

The click event's propagation will be stopped: `event.stopPropagation()`

```vue
<template>
  <div @click="parentClick" class="block-indigo space-x-4">
    <span class="font-semibold">Parent Div</span>
    <button @click.stop="buttonClick" class="btn-blue">Child Button</button>
  </div>
</template>

<script setup>
function parentClick() { alert('Parent div clicked!'); }
function buttonClick() { alert('Button clicked!'); }
</script>
```

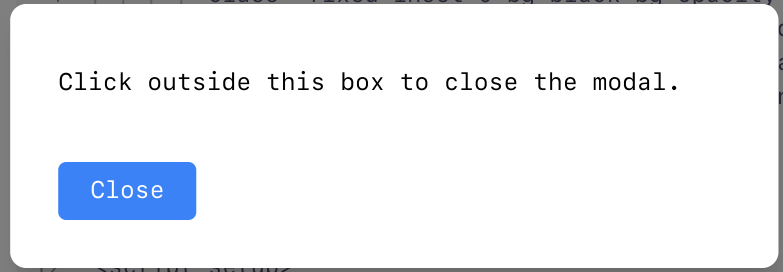
Event Handling

Event Modifiers: `.self`

Only triggers the handler if the event was fired on the element itself.

```vue
<template>
  <div  v-if="isOpen"
        @click.self="closeModal"
        class="fixed inset-0 bg-black bg-opacity-50 flex items-center justify-cent
                                                                dow-lg">
                                                                al.</p>
                                                                n-blue">Close</button>
<script setup>
import { ref } from 'vue';

const isOpen = ref(true);
const closeModal = () => isOpen.value = false
</script>
```



Event Handling

Event Modifiers: `.once`

The handler will be triggered only once. Vue automatically removes the listener after the first trigger.

```vue
<template>
  <div class="flex flex-col items-center justify-center space-y-6">
    <button @click.once="visible = !visible" class="btn-blue">
      Click Me
    </button>
    <div v-if="visible">🎉 You clicked the button!</div>
  </div>
</template>

<script setup>
import { ref } from 'vue';
const visible = ref(false);
</script>
```

Click Me

Event Handling

Event Modifiers

- Modifiers can be chained

```
<a @click.stop.prevent="doThat"></a>
```

Event Handling

Event Modifiers

- Modifiers can be chained

```
<a @click.stop.prevent="doThat"></a>
```


- Order matters when using modifiers because the relevant code is generated in the same order

Event Handling

Event Modifiers

- Modifiers can be chained

```
<a @click.stop.prevent="doThat"></a>
```


- Order matters when using modifiers because the relevant code is generated in the same order

Example

- `@click.prevent.self` will prevent click's default action on the element itself and its children

Event Handling

Event Modifiers

- Modifiers can be chained

```
<a @click.stop.prevent="doThat"></a>
```


- Order matters when using modifiers because the relevant code is generated in the same order

Example

- `@click.prevent.self` will prevent click's default action on the element itself and its children
- `@click.self.prevent` will only prevent click's default action on the element itself.

Event Handling

Event Modifiers

- The `.capture`, `.once`, and `.passive` modifiers mirror the options of the native `addEventListener` method

Event Handling

Event Modifiers

- The `.capture`, `.once`, and `.passive` modifiers mirror the options of the native `addEventListener` method

```
<!-- use capture mode when adding the event listener     -->
<!-- i.e. an event targeting an inner element is handled -->
<!-- here before being handled by that element           -->
<div @click.capture="doThis">...</div>
```

Event Handling

Event Modifiers

- The `.capture`, `.once`, and `.passive` modifiers mirror the options of the native `addEventListener` method

```
<!-- use capture mode when adding the event listener     -->
<!-- i.e. an event targeting an inner element is handled -->
<!-- here before being handled by that element           -->
<div @click.capture="doThis">...</div>

<!-- the scroll event's default behavior (scrolling) will happen -->
<!-- immediately, instead of waiting for `onScroll` to complete  -->
<!-- in case it contains `event.preventDefault()`                -->
<div @scroll.passive="onScroll">...</div>
```

Event Handling

Event Modifiers

- The `.capture`, `.once`, and `.passive` modifiers mirror the options of the native `addEventListener` method

```
<!-- use capture mode when adding the event listener     -->
<!-- i.e. an event targeting an inner element is handled -->
<!-- here before being handled by that element           -->
<div @click.capture="doThis">...</div>

<!-- the scroll event's default behavior (scrolling) will happen -->
<!-- immediately, instead of waiting for `onScroll` to complete  -->
<!-- in case it contains `event.preventDefault()`                -->
<div @scroll.passive="onScroll">...</div>
```

- The `.passive` modifier is typically used with touch event listeners for improving performance on mobile devices.

Event Handling

Event Modifiers: `.capture`

Adds the event listener in capture mode instead of the default bubbling (i.e., child → parent).

```vue
<template>
  <div @click.capture="handleParentClick" class="block-indigo space-y-2">
    <p class="font-bold text-lg text-blue-800">Parent Div (click capture)</p>
    <button @click="handleChildClick" class="btn-blue">Child Button</button>
  </div>
</template>

<script setup>
  const handleParentClick = () => alert('✅ Parent clicked (captured first)')
  const handleChildClick = () => alert('🟡 Child clicked')
</script>
```

Event Handling

Keys Modifiers

- Vue allows adding key modifiers for `@` when listening for key events.

```vue
<template>
  <div class="bg-white p-3 w-full space-y-2">
    <input  v-model="text" @keyup.esc="clear" placeholder=""
            class="w-full border border-gray-400 p-2 focus:outline-none"/>
    <p> Press <strong>Enter</strong> to submit, <strong>Esc</strong> to clear</p>
  </div>
</template>

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

const text = ref('');
function clear() { text.value = ''; }
</script>
```

Press **Enter** to submit, **Esc** to clear

Event Handling

Keys Modifiers

- Any valid key names exposed via KeyboardEvent.key can be used as modifiers (kebab-case), e.g., `page-down`

Event Handling

Keys Modifiers

- Any valid key names exposed via KeyboardEvent.key can be used as modifiers (kebab-case), e.g., `page-down`
- Vue provides aliases for the most commonly used keys: `.enter`, `.tab`, `.delete`, `.esc`, `.space`, `.up`, `.down`, `.left`, `.right`

Event Handling

Keys Modifiers

- Any valid key names exposed via KeyboardEvent.key can be used as modifiers (kebab-case), e.g., `page-down`
- Vue provides aliases for the most commonly used keys: `.enter`, `.tab`, `.delete`, `.esc`, `.space`, `.up`, `.down`, `.left`, `.right`
- When using the modifiers `.ctrl`, `.alt`, `.shift`, `.meta`, the event listener is triggered only when the corresponding key is pressed.

Event Handling

Keys Modifiers

- Any valid key names exposed via KeyboardEvent.key can be used as modifiers (kebab-case), e.g., `page-down`
- Vue provides aliases for the most commonly used keys: `.enter`, `.tab`, `.delete`, `.esc`, `.space`, `.up`, `.down`, `.left`, `.right`
- When using the modifiers `.ctrl`, `.alt`, `.shift`, `.meta`, the event listener is triggered only when the corresponding key is pressed.

```
<!-- Alt + Enter -->
<input @keyup.alt.enter="clear" />
```

Event Handling

Keys Modifiers

- Any valid key names exposed via KeyboardEvent.key can be used as modifiers (kebab-case), e.g., `page-down`
- Vue provides aliases for the most commonly used keys: `.enter`, `.tab`, `.delete`, `.esc`, `.space`, `.up`, `.down`, `.left`, `.right`
- When using the modifiers `.ctrl`, `.alt`, `.shift`, `.meta`, the event listener is triggered only when the corresponding key is pressed.

```
<!-- Alt + Enter -->
<input @keyup.alt.enter="clear" />

<!-- Ctrl + Click -->
<div @click.ctrl="doSomething">Do something</div>
```

Event Handling

Keys Modifiers

- The `.exact` modifier allows control of the exact combination of system modifiers needed to trigger an event.

```
<template>
<div class="flex gap-3 justify-center">
  <!-- this will fire even if Alt or Ctrl is also pressed -->
  <button @click.shift="onClick" class="btn-blue">Shift</button>

  <!-- this will only fire when Shift and no other keys are pressed -->
  <button @click.shift.exact="onClick" class="btn-blue">Shift Exact</button>

  <!-- this will only fire when no system modifiers are pressed -->
  <button @click.exact="onClick" class="btn-blue">No Modifiers</button>
</div>
</template>
<script setup>
const onClick = () => alert("Hello !")
</script>
```

Shift | Shift Exact | No Modifiers

Event Handling

Mouse Button Modifiers

- Vue provides the ability to restrict event handlers to only trigger on specific mouse buttons: `.left`, `.right`, `.middle`

```
1  <template>
2    <div class="flex items-center space-x-4">
3      <button class="btn-blue w-60"
4        @click.left="message = 'Left click detected!'"
5        @click.middle="message = ' Middle click detected!'"
6        @click.right.prevent="message = 'Right click detected!'"
7      > Click with mouse buttons </button>
8      <div v-show="message"> {{ message }} </div>
9    </div>
10 </template>
11
12 <script setup>
13 import { ref } from 'vue';
14 const message = ref('');
15 </script>
```

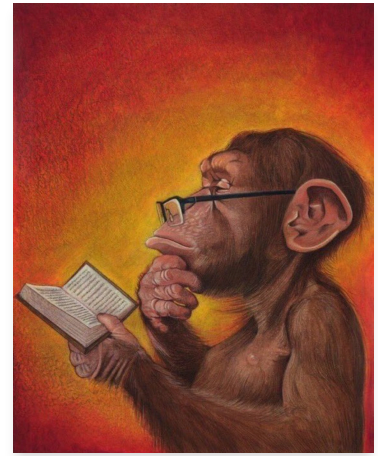
Click with mouse buttons

Input Bindings

Two-way binding

- How can we synchronize the value of `<input>` elements with corresponding variables in JavaScript ?

```
<template>
  <input type="text" id="firstname" name="firstname">
</template>
```



https://x.com/niggativebts/status/1388948646964703239

Input Bindings

Two-way binding

- Using `v-bind` and `v-on` together, we can create two-way bindings on form input elements

```
1  <template>
2    <div class="flex items-center">
3      <input :value="text" class="flex-grow input-full" placeholder="Write here ..."
4             @input="event => text = event.target.value"/>
5      <span class="ml-4 text-sm text-gray-500">{{ text.length }}</span>
6    </div>
7  </template>
8  <script setup>
9  import { ref } from 'vue'
10 const text = ref('')
11 </script>
```

Write here ...

0

Input Bindings

`v-model` Directive

- Vue provides a directive, `v-model`, which is essentially syntactic sugar for the above.

```vue
<template>
  <div class="flex items-center">
    <input v-model="text" class="flex-grow input-full" placeholder="Write here ...
    <span class="ml-4 text-sm text-gray-500">{{ text.length }}</span>
  </div>
</template>
<script setup>
import { ref } from 'vue'
const text = ref('')
</script>
```

Write here ...

0

Input Bindings

`v-model` Directive

- Vue provides a directive, `v-model`, which is essentially syntactic sugar for the above.

```vue
<template>
  <div class="flex items-center">
    <input v-model="text" class="flex-grow input-full" placeholder="Write here ...
    <span class="ml-4 text-sm text-gray-500">{{ text.length }}</span>
  </div>
</template>
<script setup>
import { ref } from 'vue'
const text = ref('')
</script>
```

Write here ... 0

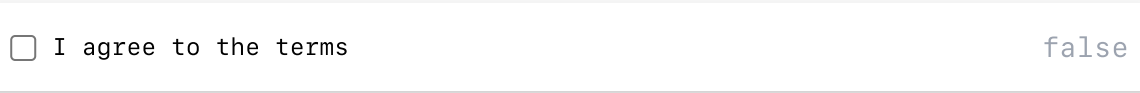
- `v-model` works not only on text inputs, but also on other input types such as checkboxes, radio buttons, and select dropdowns.

Input Bindings

Value Binding

- For radio, checkbox and select options, the `v-model` binding values are usually static strings (or booleans for checkbox)

```vue
<template>
  <div class="flex items-center justify-between bg-white p-3 shadow-md w-full">
    <label class="flex items-center space-x-2">
      <!-- `toggle` is either true or false -->
      <input type="checkbox" v-model="agreed" />
      <span>I agree to the terms</span>
    </label>
    <span class="text-sm text-gray-400">{{ agreed }}</span>
  </div>
</template>
<script setup>
import { ref } from 'vue'
const agreed = ref(false)
</script>
```

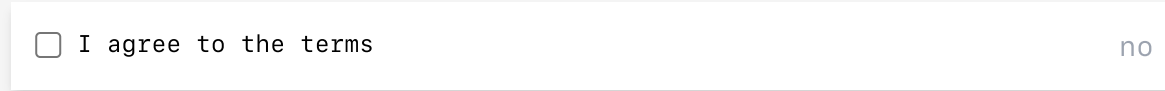


Input Bindings

Value Binding

- `v-bind` can be used to bind the value to a dynamic property

```vue
<template>
  <div class="flex items-center justify-between bg-white p-3 shadow-md w-full">
    <label class="flex items-center space-x-2">
      <!-- `agreed` is either true or false -->
      <input type="checkbox" v-model="agreed" :true-value="valueT" :false-value="v
      <span>I agree to the terms</span>
    </label>
    <span class="text-sm text-gray-400">{{ agreed }}</span>
  </div>
</template>
<script setup>
import { ref } from 'vue'
const agreed = ref('no')
const valueT = ref('yes')
const valueF = ref('no')
</script>
```



Input Bindings

Modifiers

- You can trim automatically whitespaces from user input (`.trim`) or typecast automatically as a number (`.number`)
- By default, `v-model` updates the data on every **input event**. `.lazy` modifies this behaviour to update only after the **change event**

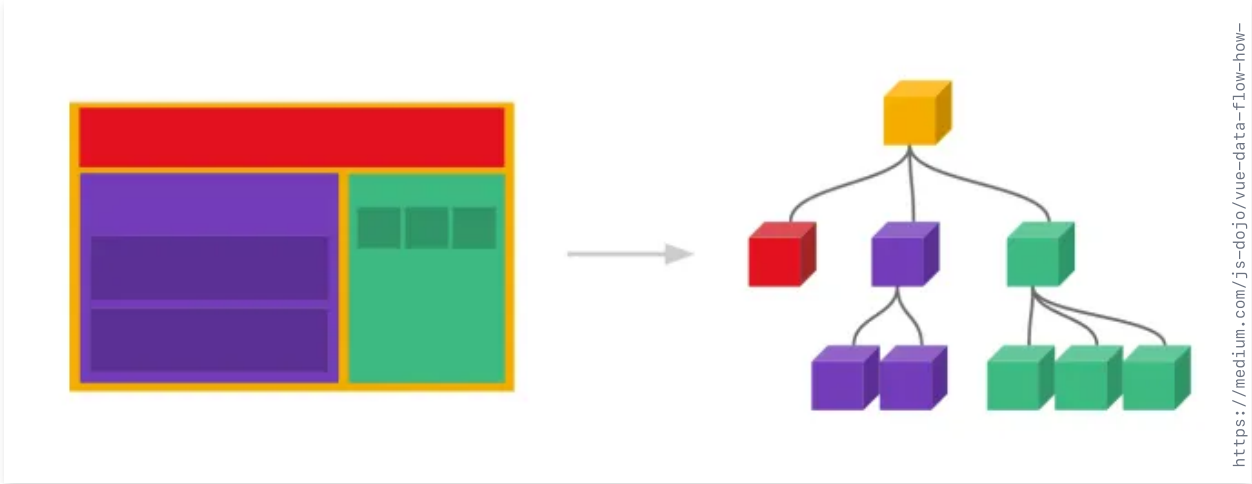
```vue
<template>
  <div class="flex flex-col bg-white p-3 shadow-md space-y-2">
    <input v-model.lazy="text" type="text" class="input-full" />
    <span class="text-sm">Lazy value: <strong>{{ text }}</strong></span>
  </div>
</template>
<script setup>
import { ref } from 'vue'
const text = ref('')
</script>
```

Reusable Components

Reusable Components

What is a component?

- Vue allows to split an app into independent and reusable pieces.
- A **component** encapsulates its own *structure*, *style*, and *behavior*



https://medium.com/js-dojo/vue-data-flow-how-it-works-3ff316a7ffcd

Reusable Components

How to define a component?

- A Vue component in defined in a dedicated file using the `.vue` extension (SFC)

Reusable Components

How to define a component?

- A Vue component in defined in a dedicated file using the `.vue` extension (SFC)
- It can be also defined as a plain JavaScript object containing Vue-specific options

```
import { ref } from 'vue'

export default {
  setup() {
    const count = ref(0)
    return { count }
  },
  template: `
    <button @click="count++">
      You clicked me {{ count }} times.
    </button>`
}
```

Reusable Components

How to define a component?

- A Vue component in defined in a dedicated file using the `.vue` extension (SFC)
- It can be also defined as a plain JavaScript object containing Vue-specific options

```
import { ref } from 'vue'

export default {
  setup() {
    const count = ref(0)
    return { count }
  },
  // Can also target an in-DOM template:
  template: '#my-template-element'
}
```

Reusable Components

How to define a component?

- A Vue component in defined in a dedicated file using the `.vue` extension (SFC)
- It can be also defined as a plain JavaScript object containing Vue-specific options

```
import { ref } from 'vue'

export default {
  setup() {
    const count = ref(0)
    return { count }
  },
  // Can also target an in-DOM template:
  template: '#my-template-element'
}
```

- Multiple components can be exported from the same file

Reusable Components

How to use a component?

- To use a child component, we need to import it in the parent component

```vue
<template>
  <button @click="incr" class="btn-blue">Clicked {{ count }} times</button>
</template>
<script setup>
  import { ref } from 'vue'
  const count = ref(0)
  const incr = () => count.value++
</script>
```

```vue
<template>
  <ButtonCounter />
</template>
<script setup>
  import ButtonCounter from './ButtonCounter.vue'
</script>
```

Clicked 0 times

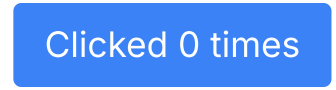
Reusable Components

How to use a component?

- To use a child component, we need to import it in the parent component

```vue
<template>
  <button @click="incr" class="btn-blue">Clicked {{ count }} times</button>
</template>
<script setup>
  import { ref } from 'vue'
  const count = ref(0)
  const incr = () => count.value++
</script>
```

```vue
<template>
  <ButtonCounter />
</template>
<script setup>
  import ButtonCounter from './ButtonCounter.vue'
</script>
```



Reusable Components

Declaring Props

- Components can **pass data** to their child components via **props**

```
<template>
  <Button label="Click me" />
</template>
```

Reusable Components

Declaring Props

- Components can **pass data** to their child components via **props**

```
<template>
  <Button label="Click me" />
</template>
```

- Enable **customization and reuse** of components with dynamic values

Reusable Components

Declaring Props

- Components can **pass data** to their child components via **props**

```
<template>
  <Button label="Click me" />
</template>
```

- Enable **customization and reuse** of components with dynamic values
- Think of them like **function parameters**, but for components

Reusable Components

Declaring Props

- Components can **pass data** to their child components via **props**

```
<template>
  <Button label="Click me" />
</template>
```

- Enable **customization and reuse** of components with dynamic values
- Think of them like **function parameters**, but for components
- Props flow **one-way**: from **parent to child**

Reusable Components

Declaring Props

- `defineProps` declares the list of *props* accepted by a component
- It's only available inside `<script setup>` and doesn't need to be explicitly imported

```
1  <template>
2    <button class="btn-blue">{{ label }}</button>
3  </template>
4  <script setup>
5    defineProps(['label'])
6  </script>
```

```
<script setup>
const props = defineProps(['title'])
console.log(props.title)
</script>
```

Reusable Components

Declaring Props

- `defineProps` declares the list of *props* accepted by a component
- It's only available inside `<script setup>` and doesn't need to be explicitly imported

```
1  <template>
2    <button class="btn-blue">{{ label }}</button>
3  </template>
4  <script setup>
5    defineProps(['label'])
6  </script>
```

```
<script setup>
const props = defineProps(['title'])
console.log(props.title)
</script>
```

Reusable Components

Declaring Props

- `defineProps` declares the list of *props* accepted by a component
- It's only available inside `<script setup>` and doesn't need to be explicitly imported

```
<template>
  <button class="btn-blue">{{ label }}</button>
</template>
<script setup>
  defineProps(['label'])
</script>
```

- `defineProps` also returns an object that we can be accessed in JavaScript if needed.

```
<script setup>
const props = defineProps(['title'])
console.log(props.title)
</script>
```

Reusable Components

Declaring Props

- Props can also be declared using object syntax
- The `key` is the name of the prop, while the `value` should be the constructor function of the expected type.

```
<script setup>
  defineProps({
    id: Number,
    title: String,
  })
</script>
```

```
<BlogPost v-bind="post" />
<BlogPost :id="post.id" :title="post.title" />
```

Reusable Components

Declaring Props

- Props can also be declared using object syntax
- The `key` is the name of the prop, while the `value` should be the constructor function of the expected type.

```
<script setup>
  defineProps({
    id: Number,
    title: String,
  })
</script>
```

- You can use `v-bind` without an argument to pass all the properties of an object as props

```
<BlogPost v-bind="post" />
<BlogPost :id="post.id" :title="post.title" />
```

Reusable Components

Declaring Props

- Props can also be declared using object syntax
- The `key` is the name of the prop, while the `value` should be the constructor function of the expected type.

```
<script setup>
  defineProps({
    id: Number,
    title: String,
  })
</script>
```

- You can use `v-bind` without an argument to pass all the properties of an object as props

```
<BlogPost v-bind="post" />
<BlogPost :id="post.id" :title="post.title" />
```

Reusable Components

Props Validation

- Components can specify requirements for their props.
- If a requirement is not met, Vue will warn you in the browser's JavaScript console
- The type can be a **custom class** or one of the following native constructors: `String`, `Number`, `Boolean`, `Array`, `Object`, `Date`, `Function`, `Symbol`, `Error`

```
<script setup>
  defineProps({
    // `null` and `undefined` values will allow any type
    propA: null,
    propB: undefined,
  })
<script>
```

Reusable Components

Props Validation

- Components can specify requirements for their props.
- If a requirement is not met, Vue will warn you in the browser's JavaScript console
- The type can be a **custom class** or one of the following native constructors: `String`, `Number`, `Boolean`, `Array`, `Object`, `Date`, `Function`, `Symbol`, `Error`

```
<script setup>
  defineProps({
    // Basic type check
    propC: Number,
  })
<script>
```

Reusable Components

Props Validation

- Components can specify requirements for their props.
- If a requirement is not met, Vue will warn you in the browser's JavaScript console
- The type can be a **custom class** or one of the following native constructors: `String`, `Number`, `Boolean`, `Array`, `Object`, `Date`, `Function`, `Symbol`, `Error`

```
<script setup>
  defineProps({
    // Multiple possible types
    propD: [String, Number],
  })
<script>
```

Reusable Components

Props Validation

- Components can specify requirements for their props.
- If a requirement is not met, Vue will warn you in the browser's JavaScript console
- The type can be a **custom class** or one of the following native constructors: `String`, `Number`, `Boolean`, `Array`, `Object`, `Date`, `Function`, `Symbol`, `Error`

```
<script setup>
  defineProps({
    // Required string
    propE: {
      type: String,
      required: true
    },
  })
<script>
```

Reusable Components

Props Validation

- Components can specify requirements for their props.
- If a requirement is not met, Vue will warn you in the browser's JavaScript console
- The type can be a **custom class** or one of the following native constructors: `String`, `Number`, `Boolean`, `Array`, `Object`, `Date`, `Function`, `Symbol`, `Error`

```
<script setup>
  defineProps({
    // Required but nullable string
    propF: {
      type: [String, null],
      required: true
    },
  })
<script>
```

Reusable Components

Props Validation

- Components can specify requirements for their props.
- If a requirement is not met, Vue will warn you in the browser's JavaScript console
- The type can be a **custom class** or one of the following native constructors: `String`, `Number`, `Boolean`, `Array`, `Object`, `Date`, `Function`, `Symbol`, `Error`

```
<script setup>
  defineProps({
    // Number with a default value
    propG: {
      type: Number,
      default: 100
    },
  })
<script>
```

Reusable Components

Props Validation

- Components can specify requirements for their props.
- If a requirement is not met, Vue will warn you in the browser's JavaScript console
- The type can be a **custom class** or one of the following native constructors: `String`, `Number`, `Boolean`, `Array`, `Object`, `Date`, `Function`, `Symbol`, `Error`

```
<script setup>
  defineProps({
    // Function with a default value
    propH: {
      type: Function,
      default() {
        return 'Default function'
      }
  })
<script>
```

Reusable Components

Props Validation

- Components can specify requirements for their props.
- If a requirement is not met, Vue will warn you in the browser's JavaScript console
- The type can be a **custom class** or one of the following native constructors: `String`, `Number`, `Boolean`, `Array`, `Object`, `Date`, `Function`, `Symbol`, `Error`

```
<script setup>
  defineProps({
    // Object with a default value
    propI: {
      type: Object,
      // Object or array defaults must be returned from a function.
      default(rawProps) {
        return { message: 'hello' }
      }
    },
```

Reusable Components

Props Validation

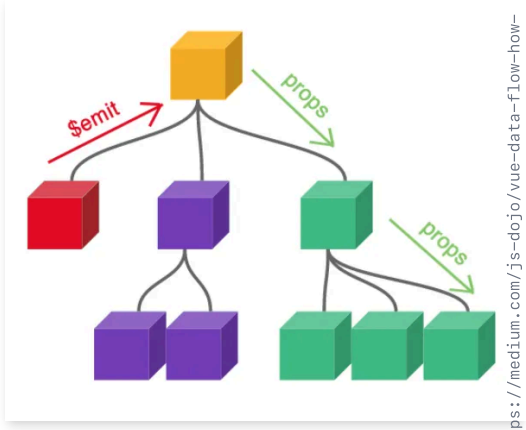
- Components can specify requirements for their props.
- If a requirement is not met, Vue will warn you in the browser's JavaScript console
- The type can be a **custom class** or one of the following native constructors: `String`, `Number`, `Boolean`, `Array`, `Object`, `Date`, `Function`, `Symbol`, `Error`

```
<script setup>
  defineProps({
    // Custom validator function
    // full props passed as 2nd argument in 3.4+
    propJ: {
      validator(value, props) {
        return ['success', 'warning', 'danger'].includes(value)
      }
    },
```

Reusable Components

Listening to Events

- The parent can choose to listen to any event on the child component instance with `v-on`
- The child component can emit an event on itself by calling the built-in `$emit` method, passing the name of the event.

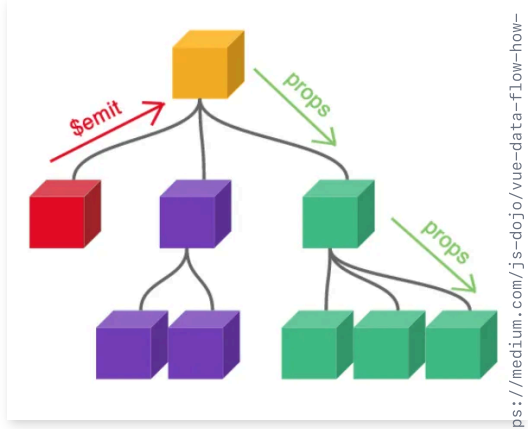


https://medium.com/js-dojo/vue-data-flow-how-it-works-3ff316a7ffcd

Reusable Components

Listening to Events

- The parent can choose to listen to any event on the child component instance with `v-on`
- The child component can emit an event on itself by calling the built-in `$emit` method, passing the name of the event.



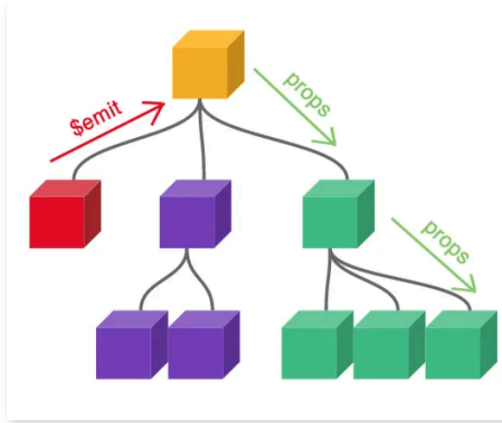
https://medium.com/js-dojo/vue-data-flow-how-it-works-3ff316a7ffcd

```vue
<template>
  <Button @btn-clicked="counter++" />
</template>
<script setup>
  import { ref } from 'vue'
  const counter = ref(0)
</script>
```

Reusable Components

Listening to Events

- The parent can choose to listen to any event on the child component instance with `v-on`
- The child component can emit an event on itself by calling the built-in `$emit` method, passing the name of the event.



https://medium.com/js-dojo/vue-data-flow-how-it-works-3ff316a7ffcd

```vue
<template>
  <Button @btn-clicked="counter++" />
</template>
<script setup>
  import { ref } from 'vue'
  const counter = ref(0)
</script>
```

```vue
<template>
  <button @click="$emit('btn-clicked')">
    Click
  </button>
</template>
```

Reusable Components

Listening to Events

- The `defineEmits` macro can be used to declare emitted events
- The `defineEmits` macro **cannot** be used inside a function, it must be placed directly within `<script setup>`
- It returns an emit function that is equivalent to the `$emit` method
- The first argument to `emit()` is the event name. Any additional arguments are passed on to the event listener.

```
<template>
  <button class="btn-blue" @click="handleBtn">Click</button>
</template>
<script setup>
  const emit = defineEmits(['btn-clicked'])
  const handleBtn = () => emit('btn-clicked', 1, 'hello')
</script>
```

Reusable Components

Listening to Events

- An emitted event can be validated if it is defined with the object syntax instead of the array syntax
- To add validation, the event is assigned a function that receives the arguments passed to the emit call and returns a boolean.

```
<script setup>
  const emit = defineEmits({
    click: null, // No validation
  })
</script>
```

Reusable Components

Listening to Events

- An emitted event can be validated if it is defined with the object syntax instead of the array syntax
- To add validation, the event is assigned a function that receives the arguments passed to the emit call and returns a boolean.

```
<script setup>
  const emit = defineEmits(['submit'])

  function submitForm(email, password) {
    emit('submit', { email, password })
  }
</script>
```

Reusable Components

Listening to Events

- An emitted event can be validated if it is defined with the object syntax instead of the array syntax
- To add validation, the event is assigned a function that receives the arguments passed to the emit call and returns a boolean.

```
<script setup>
  const emit = defineEmits({
    // Validate submit event
    submit: ({ email, password }) => email && password
  })

  function submitForm(email, password) {
    emit('submit', { email, password })
  }
</script>
```

Reusable Components

Content Insertion in Components

- Slots let you pass custom content into a component
- Think of it like a **placeholder** in the child component

```vue
<template>
  <div class="p-4 rounded-lg border-l-4 border-red-500 bg-red-100 text-red-800">
    <slot />  <!-- slot outlet -->
  </div>
</template>
```

```vue
<template>
  <AlertBox>
    Something went wrong. Please try again. <!-- slot content -->
  </AlertBox>
</template>
<script setup>
  import AlertBox from './AlertBox.vue'
</script>
```

Reusable Components

Content Insertion in Components

- Slots let you pass custom content into a component
- Think of it like a **placeholder** in the child component

```vue
<template>
  <div class="p-4 rounded-lg border-l-4 border-red-500 bg-red-100 text-red-800">
    <slot />  <!-- slot outlet -->
  </div>
</template>
```

```vue
<template>
  <AlertBox>
    Something went wrong. Please try again. <!-- slot content -->
  </AlertBox>
</template>
<script setup>
  import AlertBox from './AlertBox.vue'
</script>
```

Reusable Components

Content Insertion in Components

- Slots let you pass custom content into a component
- Think of it like a **placeholder** in the child component

```vue
<template>
  <div class="p-4 rounded-lg border-l-4 border-red-500 bg-red-100 text-red-800">
    <slot />  <!-- slot outlet -->
  </div>
</template>
```

```vue
<template>
  <AlertBox>
    Something went wrong. Please try again. <!-- slot content -->
  </AlertBox>
</template>
<script setup>
  import AlertBox from './AlertBox.vue'
</script>
```

Project (Tasks App)

Project - Tasks App

Goal

Build a **Task Manager App** using **Vue.js 3**, designed to help users manage their daily tasks.

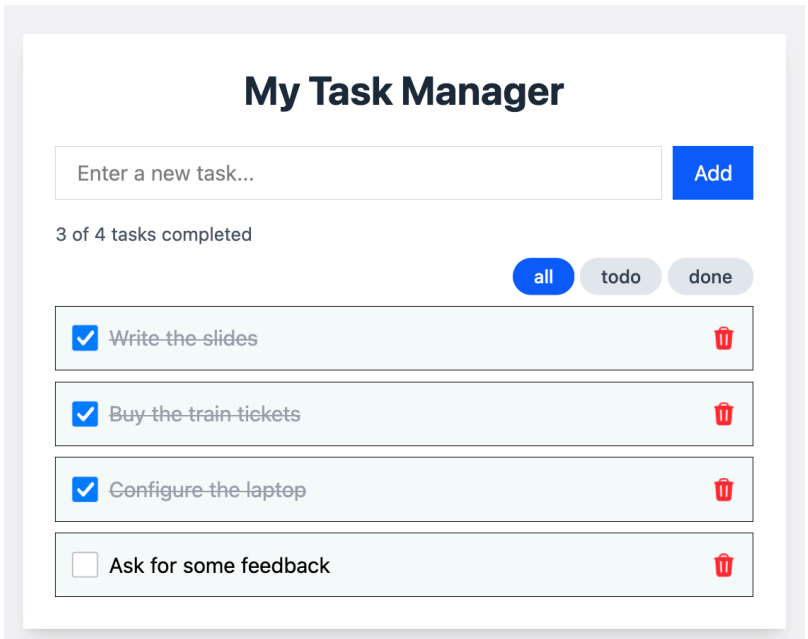
Features

- Add a new task
- List tasks
- Delete a task
- Toggle task completion
- Filter tasks (all/todo/done)

Suggested Component Structure

- `TaskList.vue` – Displays the list of tasks
- `AddTaskForm.vue` – Input form to add new tasks
- `FilterButton.vue` – Filter buttons
- `Card.vue` – Reusable component using `<slot>`

Project - Tasks App



Thank you | 117 of 117