


Introduction

 You are reading the documentation for Vue 3!

- Vue 2 support will end on Dec 31, 2023. Learn more about [Vue 2 Extended LTS](#).
- Vue 2 documentation has been moved to [v2.vuejs.org](#).
- Upgrading from Vue 2? Check out the [Migration Guide](#).



Learn Vue with video tutorials on [VueMastery.com](#)



What is Vue?

Vue (pronounced /vjuː/, like **view**) is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS, and JavaScript and provides a declarative and component-based programming model that helps you efficiently develop user interfaces, be they simple or complex.

Here is a minimal example:

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

createApp({
  setup() {
    return {
      count: ref(0)
    }
  }
}).mount('#app')
```

js

```
<div id="app">
  <button @click="count++">
    Count is: {{ count }}
  </button>
</div>
```

template

Result

Count is: 0

The above example demonstrates the two core features of Vue:

- **Declarative Rendering:** Vue extends standard HTML with a template syntax that allows us to declaratively describe HTML output based on JavaScript state.
- **Reactivity:** Vue automatically tracks JavaScript state changes and efficiently updates the DOM when changes happen.

You may already have questions - don't worry. We will cover every little detail in the rest of the documentation. For now, please read along so you can have a high-level understanding of what Vue offers.

Prerequisites

The rest of the documentation assumes basic familiarity with HTML, CSS, and JavaScript. If you are totally new to frontend development, it might not be the best idea to jump right into a framework as your first step - grasp the basics and then come back! You can check your knowledge level with [this JavaScript overview](#). Prior experience with other frameworks helps, but is not required.

The Progressive Framework

Vue is a framework and ecosystem that covers most of the common features needed in frontend development. But the web is extremely diverse - the things we build on the web may vary drastically in form and scale. With that in mind, Vue is designed to be flexible and incrementally adoptable. Depending on your use case, Vue can be used in different ways:

- Enhancing static HTML without a build step
- Embedding as Web Components on any page
- Single-Page Application (SPA)
- Fullstack / Server-Side Rendering (SSR)
- Jamstack / Static Site Generation (SSG)
- Targeting desktop, mobile, WebGL, and even the terminal

If you find these concepts intimidating, don't worry! The tutorial and guide only require basic HTML and JavaScript knowledge, and you should be able to follow along without being an expert in any of these.

If you are an experienced developer interested in how to best integrate Vue into your stack, or you are curious about what these terms mean, we discuss them in more detail in [Ways of Using Vue](#).

Despite the flexibility, the core knowledge about how Vue works is shared across all these use cases. Even if you are just a beginner now, the knowledge gained along the way will stay useful as you grow to tackle more ambitious goals in the future. If you are a veteran, you can pick the optimal way to leverage Vue based on the problems you are trying to solve, while retaining the same productivity. This is why we call Vue "The Progressive Framework": it's a framework that can grow with you and adapt to your needs.

Single-File Components

In most build-tool-enabled Vue projects, we author Vue components using an HTML-like file format called **Single-File Component** (also known as `*.vue` files, abbreviated as **SFC**). A Vue SFC, as the name suggests, encapsulates the component's logic (JavaScript), template (HTML), and styles (CSS) in a single file. Here's the previous example, written in SFC format:

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

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

<style scoped>
button {
  font-weight: bold;
}
</style>
```

SFC is a defining feature of Vue and is the recommended way to author Vue components **if** your use case warrants a build setup. You can learn more about the [how and why of SFC](#) in its dedicated section - but for now, just know that Vue will handle all the build tools setup for you.

API Styles

Vue components can be authored in two different API styles: **Options API** and **Composition API**.

Options API

With Options API, we define a component's logic using an object of options such as `data`, `methods`, and `mounted`. Properties defined by options are exposed on `this` inside functions, which points to the component instance:

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

  // Methods are functions that mutate state and trigger updates.
  // They can be bound as event handlers in templates.
  methods: {
    increment() {
      this.count++
    }
  },

  // Lifecycle hooks are called at different stages
  // of a component's lifecycle.
  // This function will be called when the component is mounted.
  mounted() {
    console.log(`The initial count is ${this.count}.`)
  }
}
</script>

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

 [Try it in the Playground](#)

Composition API

With Composition API, we define a component's logic using imported API functions. In SFCs, Composition API is typically used with `<script setup>`. The `setup` attribute is a hint that makes Vue perform compile-time transforms that allow us to use Composition API with less boilerplate. For example, imports and top-level variables / functions declared in `<script setup>` are directly usable in the template.

Here is the same component, with the exact same template, but using Composition API and `<script setup>` instead:

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

// reactive state
const count = ref(0)

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

// lifecycle hooks
onMounted(() => {
  console.log(`The initial count is ${count.value}.`)
})
</script>

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

 [Try it in the Playground](#)

Which to Choose?

Both API styles are fully capable of covering common use cases. They are different interfaces powered by the exact same underlying system. In fact, the Options API is implemented on top of the Composition API! The fundamental concepts and knowledge about Vue are shared across the two styles.

The Options API is centered around the concept of a "component instance" (`this` as seen in the example), which typically aligns better with a class-based mental model for users coming from OOP language backgrounds. It is also more beginner-friendly by abstracting away the reactivity details and enforcing code organization via option groups.

The Composition API is centered around declaring reactive state variables directly in a function scope and composing state from multiple functions together to handle complexity. It is more free-form and requires an understanding of how reactivity works in Vue to be used effectively. In return, its flexibility enables more powerful patterns for organizing and reusing logic.

You can learn more about the comparison between the two styles and the potential benefits of Composition API in the [Composition API FAQ](#).

If you are new to Vue, here's our general recommendation:

- For learning purposes, go with the style that looks easier to understand to you. Again, most of the core concepts are shared between the two styles. You can always pick up the other style later.
- For production use:
 - Go with Options API if you are not using build tools, or plan to use Vue primarily in low-complexity scenarios, e.g. progressive enhancement.
 - Go with Composition API + Single-File Components if you plan to build full applications with Vue.

You don't have to commit to only one style during the learning phase. The rest of the documentation will provide code samples in both styles where applicable, and you can toggle between them at any time using the **API Preference switches** at the top of the left sidebar.

Still Got Questions?

Check out our [FAQ](#).

Pick Your Learning Path

Different developers have different learning styles. Feel free to pick a learning path that suits your preference - although we do recommend going over all of the content, if possible!

Try the Tutorial

For those who prefer learning things hands-on.

Read the Guide

The guide walks you through every aspect of the framework in full detail.

Check out the Examples

Explore examples of core features and common UI tasks.