Vue

SFC -single file component

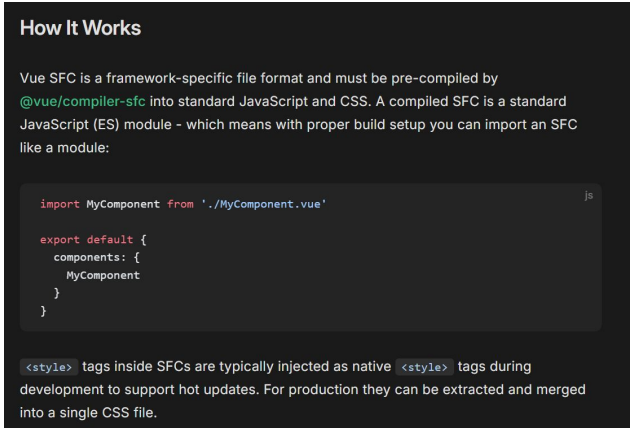
both html and css is there:

```vue
<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>
```

<script>: encapsulates logic

<template>: encapsulates html

<style>: encapsulates css



How It Works

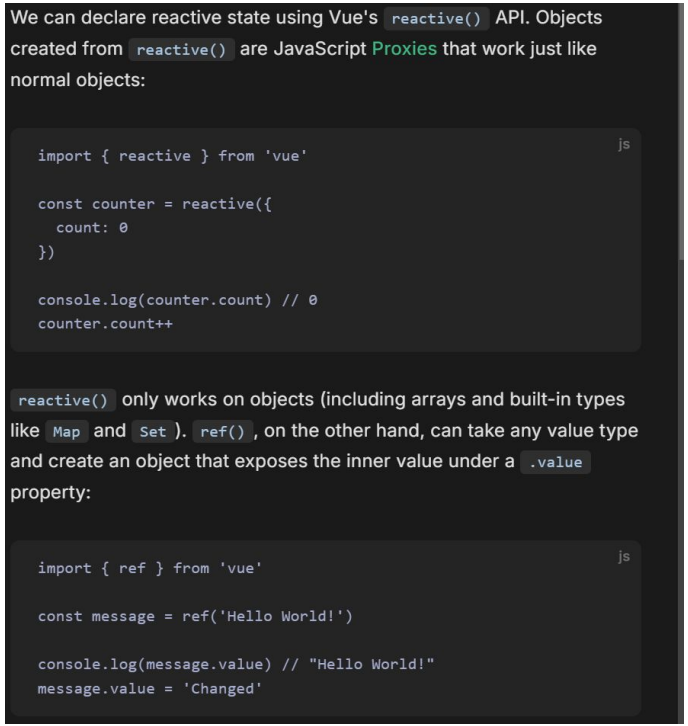
Vue SFC is a framework-specific file format and must be pre-compiled by @vue/compiler-sfc into standard JavaScript and CSS. A compiled SFC is a standard JavaScript (ES) module - which means with proper build setup you can import an SFC like a module:

```js
import MyComponent from './MyComponent.vue'

export default {
  components: {
    MyComponent
  }
}
```

`<style>` tags inside SFCs are typically injected as native `<style>` tags during development to support hot updates. For production they can be extracted and merged into a single CSS file.

Reactive() & ref

State that can trigger updates when changed is considered reactive.



We can declare reactive state using Vue's `reactive()` API. Objects created from `reactive()` are JavaScript Proxies that work just like normal objects:

```js
import { reactive } from 'vue'

const counter = reactive({
  count: 0
})

console.log(counter.count) // 0
counter.count++
```

`reactive()` only works on objects (including arrays and built-in types like `Map` and `Set`). `ref()`, on the other hand, can take any value type and create an object that exposes the inner value under a `.value` property:

```js
import { ref } from 'vue'

const message = ref('Hello World!')

console.log(message.value) // "Hello World!"
message.value = 'Changed'
```

we do not need .value when using it in the html

text interpolation

{{something}} only interpolates strings.

if we want something beyond that, we want ‘v-’ binders

Dynamically Binding Multiple Attributes

If you have a JavaScript object representing multiple attributes that looks like this:

```js
const objectOfAttrs = {
  id: 'container',
  class: 'wrapper'
}
```

You can bind them to a single element by using `v-bind` without an argument:

```template
<div v-bind="objectOfAttrs"></div>
```

Calling Functions

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

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

Raw HTML

The double mustaches interpret the data as plain text, not HTML. In order to output real HTML, you will need to use the `v-html` directive:

```template
<p>Using text interpolation: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

Using text interpolation: This should be red.
Using v-html directive: This should be red.

Attribute Bindings

Mustaches cannot be used inside HTML attributes. Instead, use a `v-bind` directive:

```template
<div v-bind:id="dynamicId"></div>
```

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

Same-name Shorthand 3.4+

If the attribute has the same name with the JavaScript value being bound, the s
be further shortened to omit the attribute value:

```
<!-- same as :id="id" -->
<div :id></div>

<!-- this also works -->
<div v-bind:id></div>
```

This is similar to the property shorthand syntax when declaring objects in JavaS
Note this is a feature that is only available in Vue 3.4 and above.

Using JavaScript Expressions

So far we've only been binding to simple property keys in our templates. But Vue act
supports the full power of JavaScript expressions inside all data bindings:

```
{{ number + 1 }}

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

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

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

These expressions will be evaluated as JavaScript in the data scope of the current component instance.

In Vue templates, JavaScript expressions can be used in the following positions:

- Inside text interpolations (mustaches)
- In the attribute value of any Vue directives (special attributes that start with `v-`)

```template
<button :disabled="isButtonDisabled">Button</button>
```

The `disabled` attribute will be included if `isButtonDisabled` has a truthy value. It will also be included if the value is an empty string, maintaining consistency with `<button disabled="">`. For other falsy values the attribute will be omitted.

Expressions Only

Each binding can only contain **one single expression**. An expression is a piece of code that can be evaluated to a value. A simple check is whether it can be used after `return`.

Therefore, the following will **NOT** work:

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

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

Directives

Directives are special attributes with the `v-` prefix. Vue provides a number of built-in directives, including `v-html` and `v-bind` which we have introduced above.

Directive attribute values are expected to be single JavaScript expressions (with the exception of `v-for`, `v-on` and `v-slot`, which will be discussed in their respective sections later). A directive's job is to reactively apply updates to the DOM when the value of its expression changes. Take `v-if` as an example:

template

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

Here, the `v-if` directive would remove / insert the `<p>` element based on the truthiness of the value of the expression `seen`.

Arguments

Some directives can take an "argument", denoted by a colon after the directive name. For example, the `v-bind` directive is used to reactively update an HTML attribute:

template

```
<a v-bind:href="url"> ... </a>

<!-- shorthand -->
<a :href="url"> ... </a>
```

Here, `href` is the argument, which tells the `v-bind` directive to bind the element's `href` attribute to the value of the expression `url`. In the shorthand, everything before the argument (i.e., `v-bind:`) is condensed into a single character, `:`.

Another example is the `v-on` directive, which listens to DOM events:

template

```
<a v-on:click="doSomething"> ... </a>

<!-- shorthand -->
<a @click="doSomething"> ... </a>
```

Here, the argument is the event name to listen to: `click`. `v-on` has a corresponding shorthand, namely the `@` character. We will talk about event handling in more detail too.

‘v-on:’ (shortcut: ‘@’)
‘v-bind:’ (shortcut: ‘:’)

Dynamic arguments

Dynamic Argument Value Constraints

Dynamic arguments are expected to evaluate to a string, with the exception of `null`. The special value `null` can be used to explicitly remove the binding. Any other non-string value will trigger a warning.

Dynamic Arguments

It is also possible to use a JavaScript expression in a directive argument by wrapping it with square brackets:

template

```
<!--
Note that there are some constraints to the argument expression,
as explained in the "Dynamic Argument Value Constraints" and "Dynamic Argument Synta
-->
<a v-bind:[attributeName]="url"> ... </a>

<!-- shorthand -->
<a :[attributeName]="url"> ... </a>
```

Here, `attributeName` will be dynamically evaluated as a JavaScript expression, and its evaluated value will be used as the final value for the argument. For example, if your component instance has a data property, `attributeName`, whose value is `"href"`, then this binding will be equivalent to `v-bind:href`.

Similarly, you can use dynamic arguments to bind a handler to a dynamic event name:

template

```
<a v-on:[eventName]="doSomething"> ... </a>

<!-- shorthand -->
<a @[eventName]="doSomething"> ... </a>
```

In this example, when `eventName`'s value is `"focus"`, `v-on:[eventName]` will be equivalent to `v-on:focus`.

V-binding and v-on examples

App.vue +

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

const titleClass = ref('title')
</script>

<template>
  <h1 :class="titleClass">Make me red</h1>
</template>

<style>
.title {
  color: red;
}
</style>
```

Show Error

PREVIEW

Make me red

Calling Methods in Inline Handlers

Instead of binding directly to a method name, we can also call methods in an inline handler. This allows us to pass the method custom arguments instead of the native event:

```js
function say(message) {
  alert(message)
}
```

```template
<button @click="say('hello')">Say hello</button>
<button @click="say('bye')">Say bye</button>
```

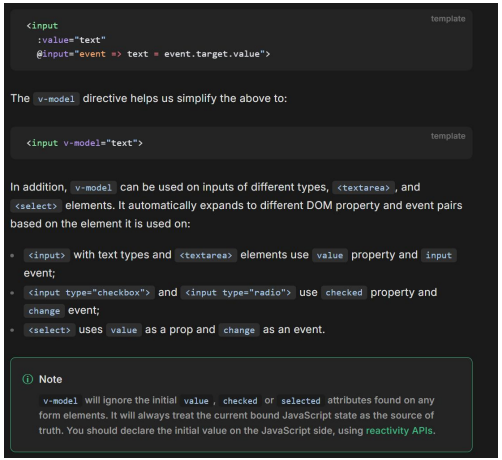
Try it in the Playground

App.vue + tsconfig.json Import Map PREVIEW

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

const name = ref('Vue.js')

function greet(event) {
  alert(`Hello ${name.value}!`)
  // `event` is the native DOM event
  if (event) {
    alert(event.target.tagName)
  }
}
</script>

<template>
  <button @click="greet">Greet</button>
</template>
```

Greet

v-model (like a combo of v-binding and v-on)



template

```
<input
  :value="text"
  @input="event => text = event.target.value">
```

The `v-model` directive helps us simplify the above to:

template

```
<input v-model="text">
```

In addition, `v-model` can be used on inputs of different types, `<textarea>`, and `<select>` elements. It automatically expands to different DOM property and event pairs based on the element it is used on:

- `<input>` with text types and `<textarea>` elements use `value` property and `input` event;
- `<input type="checkbox">` and `<input type="radio">` use `checked` property and `change` event;
- `<select>` uses `value` as a prop and `change` as an event.

ⓘ Note

`v-model` will ignore the initial `value`, `checked` or `selected` attributes found on any form elements. It will always treat the current bound JavaScript state as the source of truth. You should declare the initial value on the JavaScript side, using reactivity APIs.

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

const text = ref('')

function onInput(e) {
  text.value = e.target.value
}
</script>

<template>
  <input :value="text" @input="onInput" placeholder="Type here">
  <p>{{ text }}</p>
</template>
```

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

const text = ref('')
</script>

<template>
  <input v-model="text" placeholder="Type here">
  <p>{{ text }}</p>
</template>
```

v-if

App.vue +

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

const awesome = ref(true)

function toggle() {
  awesome.value = ! awesome.value
}
</script>

<template>
  <button @click="toggle">Toggle</button>
  <h1 v-if="awesome">Vue is awesome!</h1>
  <h1 v-else >Oh no 😢</h1>
</template>
```

PREVIEW

Toggle

Vue is awesome!

v-else-if

The `v-else-if`, as the name suggests, serves as an "else if block" for `v-if`. It can also be chained multiple times:

template

```vue
<div v-if="type === 'A'">
  A
</div>
<div v-else-if="type === 'B'">
  B
</div>
<div v-else-if="type === 'C'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

Similar to `v-else`, a `v-else-if` element must immediately follow a `v-if` or a `v-else-if` element.

v-if can be used in template element.

v-show is similar to v-if but element will always be on the DOM, just not visible

When `v-if` and `v-for` are both used on the same element, `v-if` will be evaluated first. See the list rendering guide for details.

v-for and array stuff

App.vue +

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

// give each todo a unique id
let id = 0

const newTodo = ref('')
const todos = ref([
  { id: id++, text: 'Learn HTML' },
  { id: id++, text: 'Learn JavaScript' },
  { id: id++, text: 'Learn Vue' }
])

function addTodo() {
  todos.value.push({ id: id++, text: newTodo.value})
  newTodo.value = ''
}

function removeTodo(todo) {
  todos.value = todos.value.filter(element => element.id !== todo.id)
}
</script>

<template>
  <form @submit.prevent="addTodo">
    <input v-model="newTodo" required placeholder="new todo">
    <button>Add Todo</button>
  </form>
  <ul>
    <li v-for="todo in todos" :key="todo.id">
      {{ todo.text }}
      <button @click="removeTodo(todo)">X</button>
    </li>
  </ul>
</template>
```

Show Error

PREVIEW

new todo Add Todo

- sfaf X
- gsad X
- sdgsdg X

adding classes if something is true

```
<script>
        todos = ref([{..., done: true}{...}...])
</script>

<template>
  <ul>
    <li v-for="todo in todos" :key="todo.id">
      <input type="checkbox" v-model="todo.done">
      <span :class="{ done: todo.done }">{{ todo.text }}</span>
    </li>
  </ul>
</template>

<style>
        .done {...}
</style>
```

computed ref

```
const filteredTodos = computed(()=>{
  if(hideCompleted.value){
    return todos.value.filter(todo => todo.done)
  }
  return todos.value
})
```

```
diff
- <li v-for="todo in todos">
+ <li v-for="todo in filteredTodos">
```

```
<ul>
  <li v-for="todo in filteredTodos" :key="todo.id">
    <input type="checkbox" v-model="todo.done">
    <span :class="{ done: todo.done }">{{ todo.text }}</span>
```

onMounted `onUpdated` and `onUnmounted`.

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

onMounted(()=>{
  pElementRef.value.textContent = 'blaslalala'
})

const pElementRef = ref(null)
</script>

<template>
  <p ref="pElementRef">default text</p>
</template>
```

watchers

watch(refName, callBack)

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

const count = ref(0)

watch(count, (newCount) => {
  // yes, console.log() is a side effect
  console.log(`new count is: ${newCount}`)
})
```

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

const todoId = ref(1)
const todoData = ref(null)

async function fetchData() {
  todoData.value = null
  const res = await fetch(
    `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
  )
  todoData.value = await res.json()
}

watch(todoId, (newId)=>{
  fetchData()
})

fetchData()
</script>

<template>
  <p>Todo id: {{ todoId }}</p>
  <button @click="todoId++" :disabled="!todoData">Fetch next todo</button>
  <p v-if="!todoData">Loading...</p>
  <pre v-else>{{ todoData }}</pre>
</template>
```

children and emit()

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

const greeting = ref('Hello from parent')
</script>

<template>
  <ChildComp msg="hi, this is the prop of the child" />
</template>
```

```vue
<script setup>
const props = defineProps({
  msg: String
})
</script>

<template>
  <h2>{{ msg || 'No props passed yet' }}</h2>
</template>
```

App.vue ChildComp.vue

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

const childMsg = ref('No child msg yet')
const childTalk = ref('No child talk')
</script>

<template>
  <ChildComp @response="(msg) => childMsg = msg" @talk="(talk) => childTalk = talk" />
  <p>{{ childMsg }}</p>
  <p>{{ childTalk }}</p>
</template>
```

App.vue ChildComp.vue

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

emit('response', 'hello from childz')
</script>

<template>
  <h2>Child component</h2>
</template>
```

children and slots

App.vue ChildComp.vue

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

const msg = ref('from parent')
</script>

<template>
  <ChildComp>
    {{msg}}
  </ChildComp>
</template>
```

App.vue ChildComp.vue

```vue
<template>
  <slot>Fallback content</slot>
  <slot>Fallback content</slot>
  <slot>Fallback content</slot>
</template>
```