Differences Between Vue 2 and Vue 3

* Updated feature
* Newly added feature
* Removed feature

List of contents:

### Global API

* [Global Vue API is changed to use an application instance](https://v3-migration.vuejs.org/breaking-changes/global-api.html)
* [Global and internal APIs have been restructured to be tree-shakable](https://v3-migration.vuejs.org/breaking-changes/global-api-treeshaking.html)

New features in Vue 3

* [Composition API](https://vuejs.org/guide/extras/composition-api-faq.html)
* Fragments
* [Teleport](https://vuejs.org/guide/built-ins/teleport.html)
* [Suspense](https://vuejs.org/guide/built-ins/suspense.html)
* [SFC Composition API Syntax Sugar (<script setup>)](https://vuejs.org/api/sfc-script-setup.html)\*
* [SFC State-driven CSS Variables (v-bind in <style>)](https://vuejs.org/api/sfc-css-features.html#v-bind-in-css)\*
* [SFC <style scoped> can now include global rules or rules that target only slotted content](https://github.com/vuejs/rfcs/blob/master/active-rfcs/0023-scoped-styles-changes.md)

### Template Directives

* [v-model usage on components has been reworked, replacing v-bind.sync](https://v3-migration.vuejs.org/breaking-changes/v-model.html)
* [key usage on <template v-for> and non-v-for nodes has changed](https://v3-migration.vuejs.org/breaking-changes/key-attribute.html)
* [v-if and v-for precedence when used on the same element has changed](https://v3-migration.vuejs.org/breaking-changes/v-if-v-for.html)
* [v-bind="object" is now order-sensitive](https://v3-migration.vuejs.org/breaking-changes/v-bind.html)
* [v-on:event.native modifier has been removed](https://v3-migration.vuejs.org/breaking-changes/v-on-native-modifier-removed)

### Components

* [Component events should now be declared with the emits option](https://v3-migration.vuejs.org/breaking-changes/emits-option)
* [Functional components can only be created using a plain function](https://v3-migration.vuejs.org/breaking-changes/functional-components.html)
* [functional attribute on single-file component (SFC) <template> and functional component option are deprecated](https://v3-migration.vuejs.org/breaking-changes/functional-components.html)
* [Async components now require defineAsyncComponent method to be created](https://v3-migration.vuejs.org/breaking-changes/async-components.html)

### Render Function

* [Render function API changed](https://v3-migration.vuejs.org/breaking-changes/render-function-api.html)
* [$listeners has been removed / merged into $attrs](https://v3-migration.vuejs.org/breaking-changes/listeners-removed)
* [$attrs now includes class and style attributes](https://v3-migration.vuejs.org/breaking-changes/attrs-includes-class-style)
* [$scopedSlots property is removed and all slots are exposed via $slots as functions](https://v3-migration.vuejs.org/breaking-changes/slots-unification.html)

### Removed API

* [keyCode support as v-on modifiers](https://v3-migration.vuejs.org/breaking-changes/keycode-modifiers.html)
* [$on, $off and $once instance methods](https://v3-migration.vuejs.org/breaking-changes/events-api.html)
* [Filters](https://v3-migration.vuejs.org/breaking-changes/filters.html)
* [$children instance property](https://v3-migration.vuejs.org/breaking-changes/children.html)
* [propsData option](https://v3-migration.vuejs.org/breaking-changes/props-data.html)

### Other Minor Changes

* The destroyed lifecycle option has been renamed to unmounted
* The beforeDestroy lifecycle option has been renamed to beforeUnmount
* [Custom directive API changed to align with component lifecycle and binding.expression removed](https://v3-migration.vuejs.org/breaking-changes/custom-directives.html)
* [Props default factory function no longer has access to this context](https://v3-migration.vuejs.org/breaking-changes/props-default-this.html)
* [The data option should always be declared as a function](https://v3-migration.vuejs.org/breaking-changes/data-option.html)
* [The data option from mixins is now merged shallowly](https://v3-migration.vuejs.org/breaking-changes/data-option.html#mixin-merge-behavior-change)
* [Some transition classes got a rename](https://v3-migration.vuejs.org/breaking-changes/transition.html) - Done
* [<TransitionGroup> now renders no wrapper element by default](https://v3-migration.vuejs.org/breaking-changes/transition-group.html)
* [When watching an array, the callback will only trigger when the array is replaced. If you need to trigger on mutation, the deep option must be specified.](https://v3-migration.vuejs.org/breaking-changes/watch.html)
* <template> tags with no special directives (v-if/else-if/else, v-for, or v-slot) are now treated as plain elements and will result in a native <template> element instead of rendering its inner content.
* [Mounted application does not replace the element it's mounted to](https://v3-migration.vuejs.org/breaking-changes/mount-changes.html)
* [Lifecycle hook: events prefix changed to vnode](https://v3-migration.vuejs.org/breaking-changes/vnode-lifecycle-events.html)

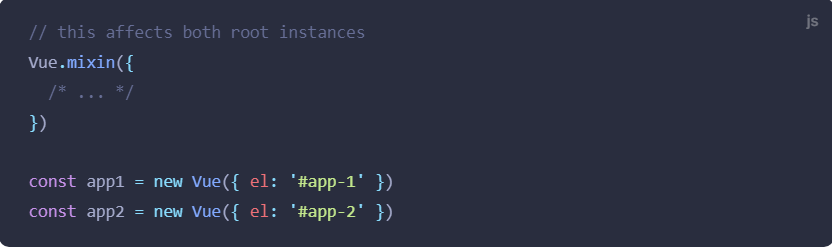
**Global API**

* 1. **Global Vue API is changed to use an application instance:**

Technically, Vue 2 doesn't have a concept of an "app". What we define as an app is simply a root Vue instance created via new Vue(). Every root instance created from the same Vue constructor **shares the same global configuration**.  As a result:

* Global configuration makes it easy to accidentally pollute other test cases during testing. Some APIs like Vue.use and Vue.mixin don't even have a way to revert their effects.  This makes tests involving plugins particularly tricky.
* Global configuration makes it difficult to share the same copy of Vue between multiple "apps" on the same page, but with different global configurations.

**2.x Syntax**



**3.x Update**

To avoid these problems, Vue 3 introduced:

**A New Global API: createApp**

Calling createApp returns an *app instance*, a new concept in Vue 3



Here is a table of the Vue 2 global APIs and their corresponding instance APIs:

|  |  |
| --- | --- |
| **2.x Global API** | **3.x Instance API (app)** |
| Vue.config | app.config |
| Vue.config.productionTip | removed |
| Vue.config.ignoredElements | app.config.compilerOptions.isCustomElement |
| Vue.component | app.component |
| Vue.directive | app.directive |
| Vue.mixin | app.mixin |
| Vue.use | app.use |
| Vue.prototype | app.config.globalProperties |
| Vue.extend | *removed* |

**2. Global API Treeshaking**

**2.x Syntax**

If you’ve ever had to manually manipulate DOM in Vue, you might have come across this pattern:

****

Vue.nextTick() is a global API exposed directly on a single Vue object – in fact, the instance method $nextTick() is just a handy wrapper around Vue.nextTick() with the callback’s this context automatically bound to the current instance for convenience.

Module bundlers like webpack support [tree-shaking](https://webpack.js.org/guides/tree-shaking/), is a fancy term commonly used in the JavaScript context for *“dead-code elimination”.* It relies on the [static structure](http://exploringjs.com/es6/ch_modules.html#static-module-structure) of ES2015 module syntax, i.e. [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) and [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export). The name and concept have been popularized by the ES2015 module bundler rollup.  Unfortunately, due to how the code is written in previous Vue versions, global APIs like Vue.nextTick() are not tree-shakeable and will be included in the final bundle regardless of where they are actually used or not.

**3.x Syntax**

In Vue 3, the global and internal APIs have been restructured with tree-shaking support in mind. As a result, the global APIs can now only be accessed as named exports for the ES Modules build. For example, our previous snippets should now look like this:



Calling Vue.nextTick() directly will now result in the infamous undefined is not a function error.

With this change, provided the module bundler supports tree-shaking, global APIs that are not used in a Vue application will be eliminated from the final bundle, resulting in an optimal file size.

**Affected APIs**

These global APIs in Vue 2.x are affected by this change:

* Vue.nextTick
* Vue.observable (replaced by Vue.reactive)
* Vue.version
* Vue.compile (only in full builds)
* Vue.set (only in compat builds)
* Vue.delete (only in compat builds)

### **New features in Vue 3**

# **What is Composition API?**

Composition API is a set of APIs that allows us to author Vue components using imported functions instead of declaring options. It is an umbrella term that covers the following APIs:

* [Reactivity API](https://vuejs.org/api/reactivity-core), e.g. ref() and reactive(), that allows us to directly create reactive state, computed state, and watchers.
* [Lifecycle Hooks](https://vuejs.org/api/composition-api-lifecycle), e.g. onMounted() and onUnmounted(), that allow us to programmatically hook into the component lifecycle.
* [Dependency Injection](https://vuejs.org/api/composition-api-dependency-injection), i.e. provide() and inject(), that allow us to leverage Vue's dependency injection system while using Reactivity APIs.

Composition API is a built-in feature of Vue 3 and [Vue 2.7](https://blog.vuejs.org/posts/vue-2-7-naruto.html). For older Vue 2 versions, use the officially maintained [@vue/composition-api](https://github.com/vuejs/composition-api) plugin. In Vue 3, it is also primarily used together with the [<script setup>](https://vuejs.org/api/sfc-script-setup) syntax in Single-File Components. Here's a basic example of a component using Composition API:



# **Fragments**

# In Vue 3, components now have official support for multi-root node components, i.e., fragments!

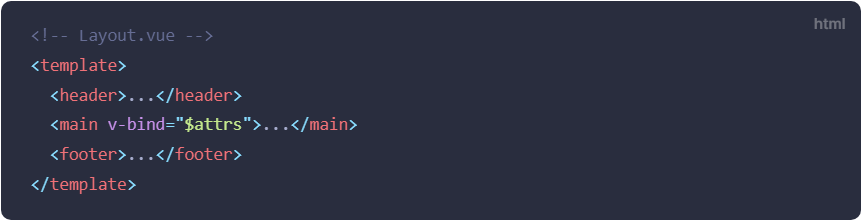
**2.x Syntax**

In 2.x, multi-root components were not supported and would emit a warning when a user accidentally created one. As a result, many components are wrapped in a single <div> in order to fix this error.



**3.x Syntax**

In 3.x, components now can have multiple root nodes! However, this does require developers to explicitly define where attributes should be distributed.



# **Teleport**

<Teleport> is a built-in component that allows us to "teleport" a part of a component's template into a DOM node that exists outside the DOM hierarchy of that component. In other words, with teleport, you can render a component in a different place in the DOM tree, even if this place is not in your app's scope.

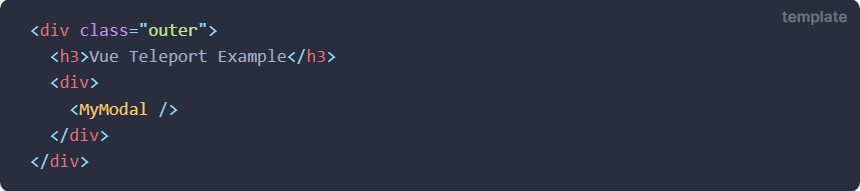
## Basic Usage[​](https://vuejs.org/guide/built-ins/teleport.html#basic-usage)

Sometimes we may run into the following scenario: a part of a component's template belongs to it logically, but from a visual standpoint, it should be displayed somewhere else in the DOM, outside of the Vue application.

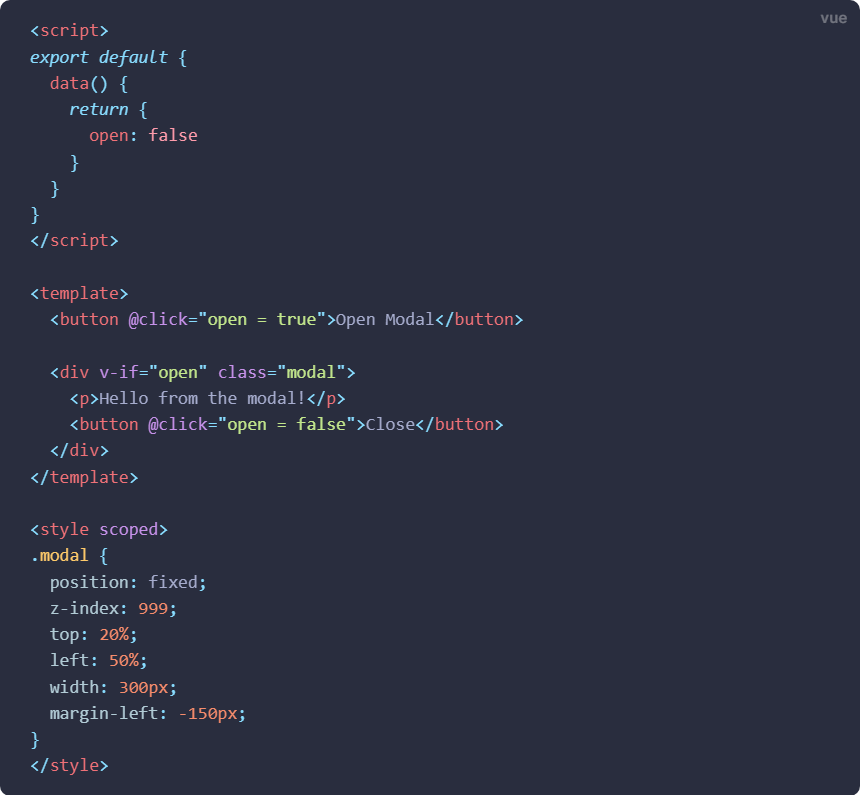
The most common example of this is when building a full-screen modal. Ideally, we want the modal's button and the modal itself to live within the same component, since they are both related to the open / close state of the modal. But that means the modal will be rendered alongside the button, deeply nested in the application's DOM hierarchy. This can create some tricky issues when positioning the modal via CSS.

Teleport is very handy when working with modals, notifications, popups or other elements that are sensitive to where they’re placed in the DOM tree.

**2.x implementation**



MyModal.vue



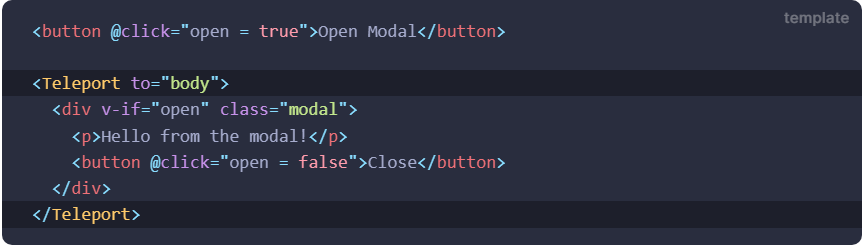
The component contains a <button> to trigger the opening of the modal, and a <div> with a class of .modal, which will contain the modal's content and a button to self-close.

When using this component inside the initial HTML structure, there are a number of potential issues:

* + - * 1. position: fixed only places the element relative to the viewport when no ancestor element has transform, perspective or filter property set. If, for example, we intend to animate the ancestor <div class="outer"> with a CSS transform, it would break the modal layout!
        2. The modal's z-index is constrained by its containing elements. If there is another element that overlaps with <div class="outer"> and has a higher z-index, it would cover our modal.

**3.x implementation using teleport**

<Teleport> provides a clean way to work around these, by allowing us to break out of the nested DOM structure. Let's modify <MyModal> to use <Teleport>:



The to target of <Teleport> expects a CSS selector string or an actual DOM node. Here, we are essentially telling Vue to "**teleport** this template fragment **to** the body tag".

# **Suspense(Experimental Feature)**

Experimental Feature:

**<Suspense>** is an experimental feature. It is not guaranteed to reach stable status and the API may change before it does.

Suspense is a new feature that renders a default/fallback component until the main component fetches the data.

**<Supense>** has two slots: #default and #fallback. The default slot content is shown if possible, and the fallback slot content is shown when waiting for async dependencies to resolve.

Sometimes we use async operations to fetch data from the server. Instead of handing the template with v-if and then setting it back when we return the data, Suspense does it for us.

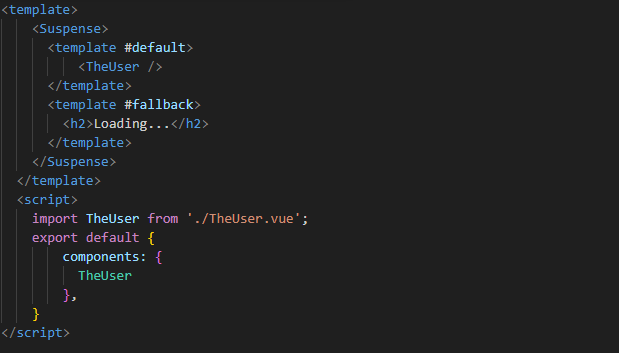
Suspense can be used for both parts of the template, or the whole template:

TheSuspense.vue

Example 1:

# 

Example 2:



# [**SFC Composition API Syntax Sugar (<script setup>)**](https://vuejs.org/api/sfc-script-setup.html)

<script setup> is a compile-time syntactic sugar for using Composition API inside Single-File Components (SFCs). It is the recommended syntax if you are using both SFCs and Composition API. It provides a number of advantages over the normal <script> syntax:

* More succinct code with less boilerplate
* Ability to declare props and emitted events using pure TypeScript
* Better runtime performance (the template is compiled into a render function in the same scope, without an intermediate proxy)
* Better IDE type-inference performance (less work for the language server to extract types from code)

**Basic Syntax**

To opt-in to the syntax, add the setup attribute to the <script> block:

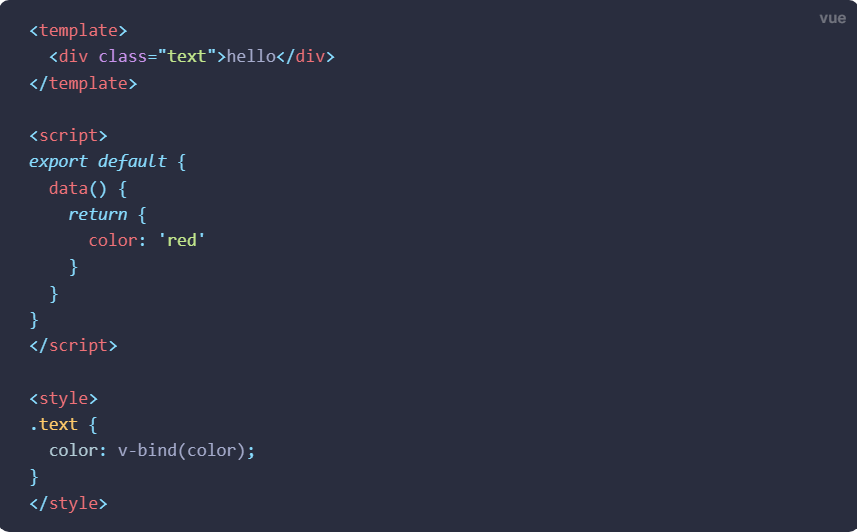


The code inside is compiled as the content of the component's setup() function. This means that unlike normal <script>, which only executes once when the component is first imported, code inside <script setup> will **execute every time an instance of the component is created**.

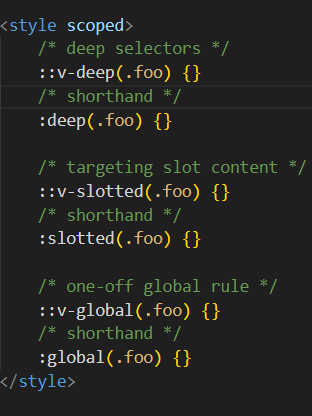
setup() is the function automatically called by Vue before the component is created once the props are resolved. It serves as the entry point for the composition API.

# [**SFC State-driven CSS Variables (v-bind in <style>)**](https://vuejs.org/api/sfc-css-features.html#v-bind-in-css)

SFC <style> tags support linking CSS values to dynamic component state using the v-bind CSS function:



# [**SFC <style scoped> can now include global rules or rules that target only slotted content**](https://github.com/vuejs/rfcs/blob/master/active-rfcs/0023-scoped-styles-changes.md)



Reference link: <https://github.com/vuejs/rfcs/blob/master/active-rfcs/0023-scoped-styles-changes.md>

# **Deep Selectors**

With scoped, the parent component's styles will not leak into child components. However, a child component's root node will be affected by both the parent's scoped CSS and the child's scoped CSS. This is by design so that the parent can style the child root element for layout purposes.

Or you can use :deep() pseudo class as in example below:



which will compile into:



# **Slotted Selectors:**

By default, scoped styles do not affect contents rendered by <slot/>, as they are considered to be owned by the parent component passing them in. To explicitly target slot content, use the :slotted pseudo-class:



# **Global Selectors**

If you want just one rule to apply globally, you can use the :global pseudo-class rather than creating another <style> (see below):



**Template Directives:**

1. **v-model**

In terms of what has changed, at a high level:

* + **When used on custom components, v-model prop and event default names are changed:**
  + **prop: value -> modelValue;**
  + **event: input -> update:modelValue;**
  + **v-bind's .sync modifier and component model option are removed and replaced with an argument on v-model;**
  + **Multiple v-model bindings on the same component are possible now;**
  + **Added the ability to create custom v-model modifiers.**





**Multiple v-model on same component:**

App.vue

****

TheForm.vue

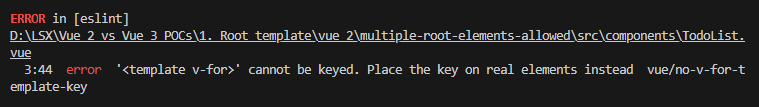
****

1. **With <template v-for>**

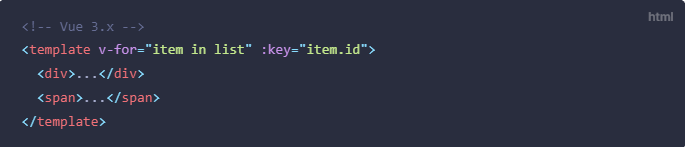
In Vue 2.x, a <template> tag could not have a key. Instead, you could place the keys on each of its children.



If key is placed on root element, the following error comes up.



In Vue 3.x, the key should be placed on the <template> tag instead.



1. **v-if vs. v-for Precedence**

**2.x Syntax**

In 2.x, when using v-if and v-for on the same element, v-for would take precedence.

**3.x Syntax**

In 3.x, v-if will always have the higher precedence than v-for.

Caveat: Its better to not to use v-if and v-for on the same node (as shown in the example below), rather use computed property for filtering out the elements that satisfy the mentioned in v-if directive. v-if can be placed on the parent node.

TodoList.vue



Error corresponding to the usage of v-if and v-for together on the same node:



1. **v-bind Merge Behavior**

**2.x Syntax**

In 2.x, if an element has both v-bind="object" and an identical individual attribute defined, the individual attribute would always overwrite bindings in the object.



**3.x Syntax**

## In 3x, if an element has both v-bind="object" and an identical individual attribute defined, the order of how the bindings are declared determines how they are merged. In other words, rather than assuming developers want the individual attribute to always override what is defined in the object, developers now have more control over the desired merging behavior.

## 

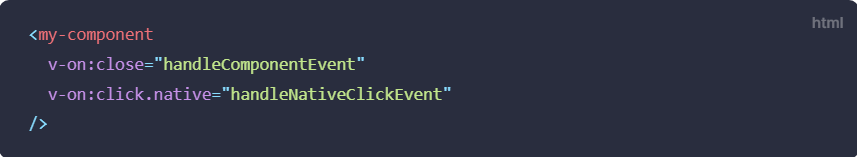
1. **v-bind Merge Behavior**

The .native modifier for v-on has been removed.

Native events are the DOM events of Syncfusion Vue component’s root element. Syncfusion Vue is a modern UI Components library that has been built from the ground up to be lightweight, responsive, modular and touch friendly..native modifier for v-on directive is used for binding these events. The Syntax for Binding Native Event is v-on:event-name.native="function".

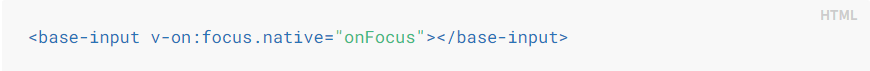
**2.x Syntax**

Event listeners passed to a component with v-on are by default only triggered by emitting an event with this.$emit. To add a native DOM listener to the child component's root element instead, the .native modifier can be used:

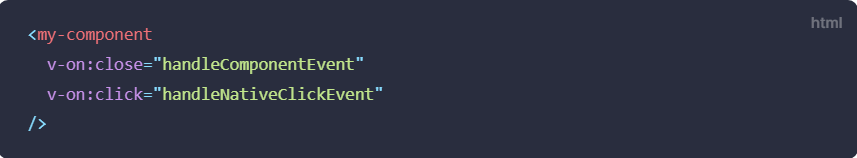


**3.x Update**

The .native modifier for v-on has been removed.



At the same time, the [new emits option](https://v3-migration.vuejs.org/breaking-changes/emits-option) allows the child to define which events it does indeed emit.



MyComponent.vue



**Components**

1. **Emits option**

As discussed right above, Vue 3 now offers an emits option, similar to the existing props option. This option can be used to define the events that a component can emit to its parent.

1. [**Functional components can only be created using a plain function**](https://v3-migration.vuejs.org/breaking-changes/functional-components.html)

In Vue 2, functional components had two primary use cases:

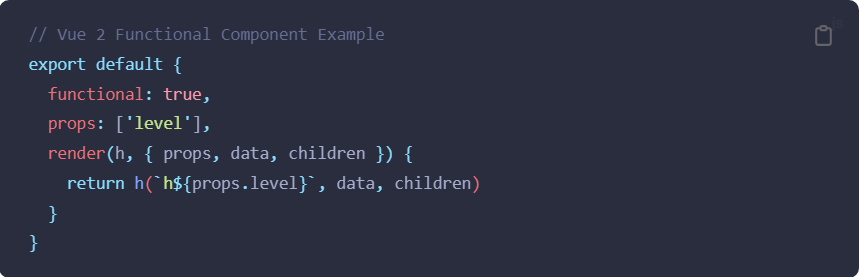
* + - as a performance optimization, because they initialized much faster than stateful components
    - to return multiple root nodes

However, in Vue 3, the performance of stateful components has improved to the point that the difference is negligible. In addition, stateful components now also include the ability to return multiple root nodes.

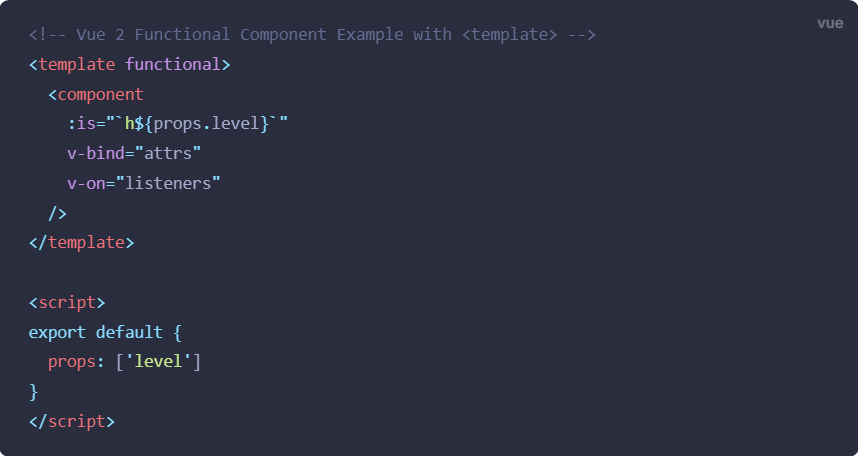
As a result, the only remaining use case for functional components is simple components, such as a component to create a dynamic heading. Otherwise, it is recommended to use stateful components as you normally would.

**2.x Syntax**

Using the <dynamic-heading> component, which is responsible for rendering out the appropriate heading (i.e., h1, h2, h3, etc.), this could have been written as a single-file component in 2.x as:



Or, for those who preferred the <template> in a single-file component:

****

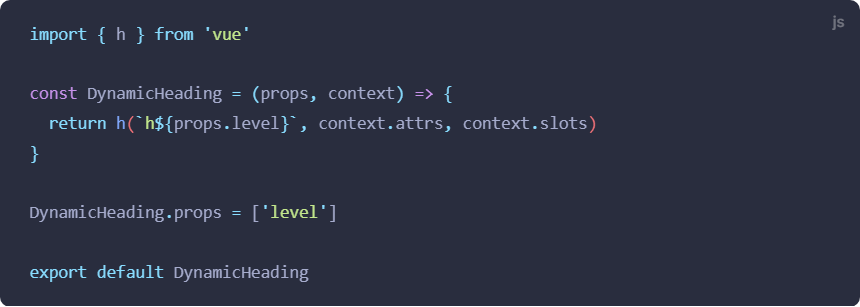
**3.x Syntax**

Now in Vue 3, all functional components are created with a plain function. In other words, there is no need to define the { functional: true } component option.

They will receive two arguments: props and context. The context argument is an object that contains a component's attrs, slots, and emit properties.

In addition, rather than implicitly provide h in a render function, h is now imported globally.

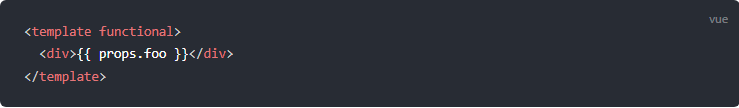
Using the previously mentioned example of a <dynamic-heading> component, here is how it looks now.

****

1. **functional attribute on single-file component (SFC) <template> and functional component option are deprecated:**

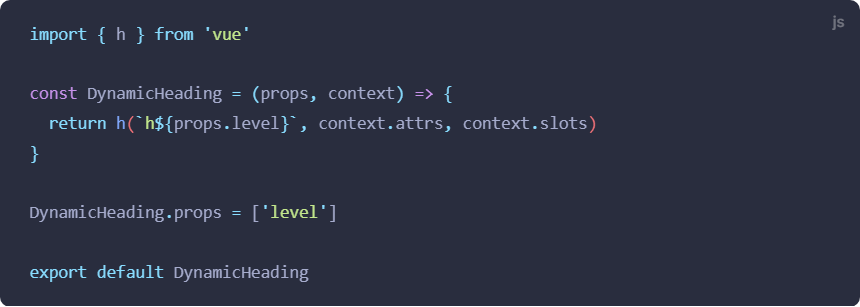
To denote a template that should be compiled as a functional component, the functional attribute needs to be added to the template block in Vue 2. This also allows omitting the functional option in the <script> block.

**2.x Syntax**

****

**3.x Syntax**

Now in Vue 3, all functional components are created with a plain function.

****

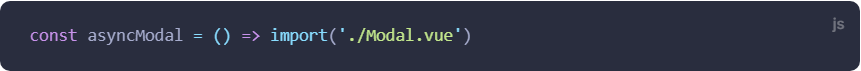
1. **Async components now require defineAsyncComponent method to be created**

Here is a high level overview of what has changed:

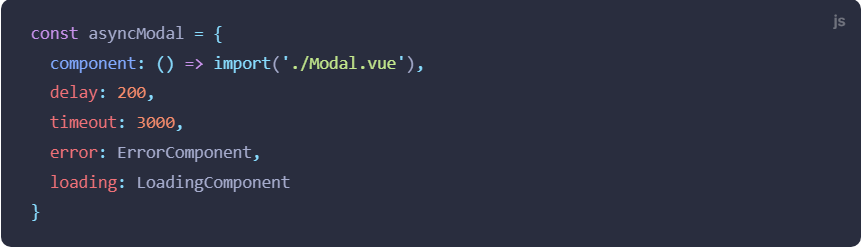
* + - New defineAsyncComponent helper method that explicitly defines async components
    - component option renamed to loader
    - Loader function does not inherently receive resolve and reject arguments and must return a Promise

**2.x Syntax**

Previously, async components were created by simply defining a component as a function that returned a promise, such as:



Or, for the more advanced component syntax with options:



**3.x Syntax**

Now, in Vue 3, since functional components are defined as pure functions, async components definitions need to be explicitly defined by wrapping it in a new defineAsyncComponent helper:



**Render Function**

**1.**[**Render function API changed**](https://v3-migration.vuejs.org/breaking-changes/render-function-api.html)**:**

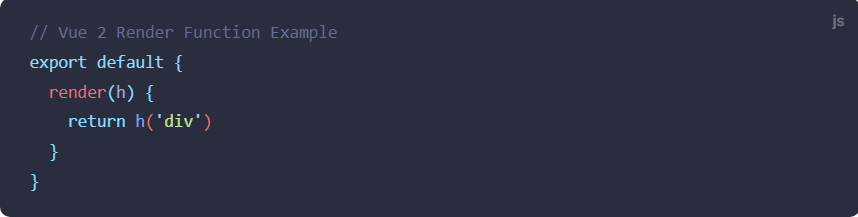
This change will not affect <template> users.

Here is a quick summary of what has changed:

* h is now globally imported instead of passed to render functions as an argument
* render function arguments changed to be more consistent between stateful and functional components
* VNodes now have a flat props structure

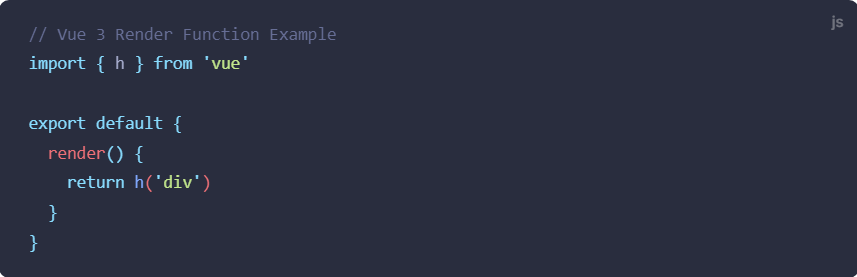
**2.x Syntax**

In 2.x, the render function would automatically receive the h function (which is a conventional alias for createElement) as an argument:



**3.x Update**

In 3.x, h is now globally imported instead of being automatically passed as an argument.

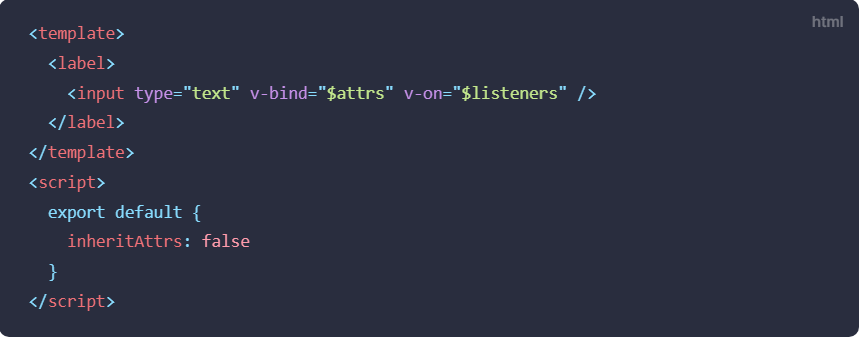


* 1. [**$listeners has been removed / merged into $attrs**](https://v3-migration.vuejs.org/breaking-changes/listeners-removed)

The $listeners object has been removed in Vue 3. Event listeners are now part of $attrs:

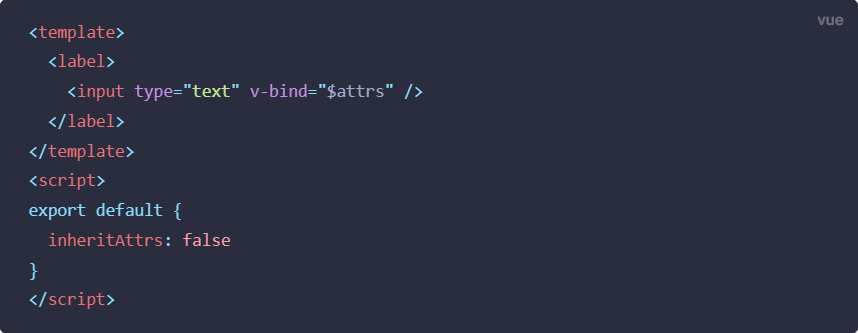
**2.x Syntax**

In Vue 2, you can access attributes passed to your components with this.$attrs, and event listeners with this.$listeners. In combination with inheritAttrs: false, they allow the developer to apply these attributes and listeners to some other element instead of the root element:

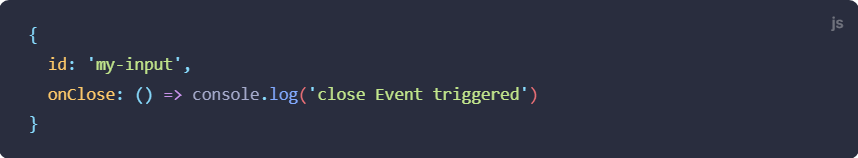


**3.x Syntax**

In Vue 3's virtual DOM, event listeners are now just attributes, prefixed with on, and as such are part of the $attrs object, so $listeners has been removed.



### If this component received an id attribute and a v-on:close listener, the $attrs object will now look like this:



* 1. [**$attrs now includes class and style attributes**](https://v3-migration.vuejs.org/breaking-changes/attrs-includes-class-style)

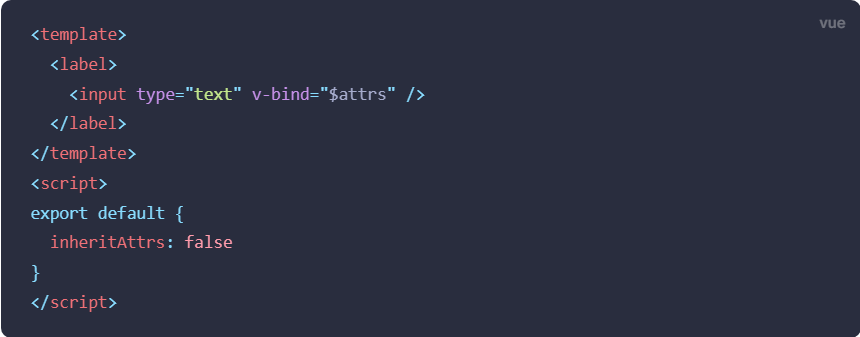
$attrs now contains *all* attributes passed to a component, including class and style.

**2.x Behavior**

“class” and “style” attributes get some special handling in the Vue 2 virtual DOM implementation. For that reason, they are *not* included in $attrs, while all other attributes are.

A side effect of this manifests when using inheritAttrs: false:

* Attributes in $attrs are no longer automatically added to the root element, leaving it to the developer to decide where to add them.
* But class and style, not being part of $attrs, will still be applied to the component's root element:



when used like this:

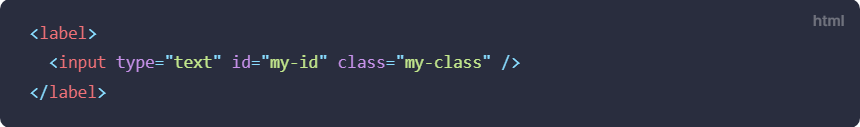


...will generate this HTML:



**3.x Behavior**

$attrs contains *all* attributes, which makes it easier to apply all of them to a different element. The example from above now generates the following HTML:



* 1. **$scopedSlots property is removed and all slots are exposed via $slots as function**

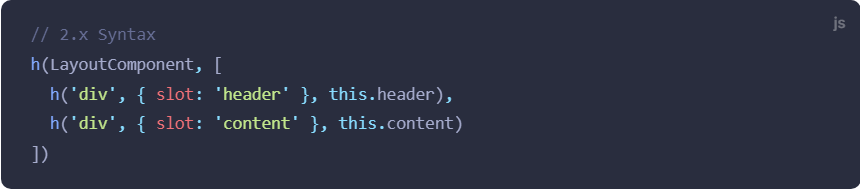
This change unifies normal and scoped slots in 3.x.

Here is a quick summary of what has changed:

* this.$slots now exposes slots as functions
* this.$scopedSlots is removed

**2.x Syntax:**

When using the render function, i.e., h, 2.x used to define the slot data property on the content nodes.

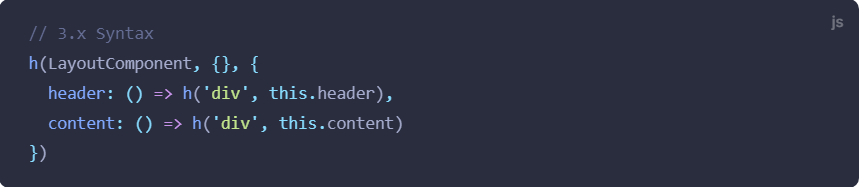


In addition, when referencing scoped slots, they could be referenced using the following syntax:



**3.x Syntax:**

In 3.x, slots are defined as children of the current node as an object:



And when you need to reference scoped slots programmatically, they are now unified into the $slots option.

### **Removed API**

**1. Events API (removed):**

$on, $off and $once instance methods are removed. Component instances no longer implement the event emitter interface. These could be used to create an event bus to create global event listeners used across the whole application:. $emit is still a part of the existing API as it's used to trigger event handlers declaratively attached by a parent component.

The event bus pattern can be replaced by using an external library implementing the event emitter interface, for example [mitt](https://github.com/developit/mitt) or [tiny-emitter](https://github.com/scottcorgan/tiny-emitter).

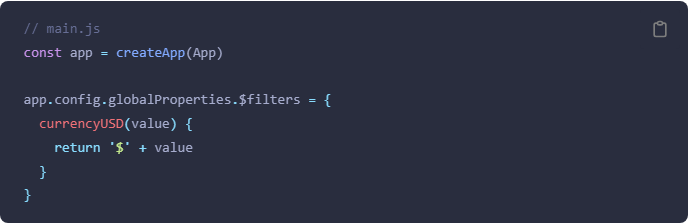
**2. Filters (removed):**

Filters are removed from Vue 3.0 and no longer supported. Instead, it is recommended replacing them with method calls or computed properties.

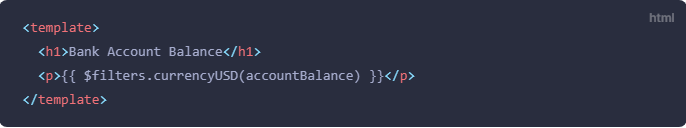
**Global Filters:**

If you are using filters that were globally registered and then used throughout your app, it's likely not convenient to replace them with computed properties or methods in each individual component.

Instead, you can make your global filters available to all components through [globalProperties](https://vuejs.org/api/application.html#app-config-globalproperties): app.config.globalProperties



Then you can fix all templates using this $filters object like this:



**3. KeyCode Modifiers (removed):**

Here is a quick summary of what has changed:

* **Using numbers, i.e. keyCodes, as v-on modifiers is no longer supported**
* **config.keyCodes is no longer supported**

## 2.x Syntax:

## 

## 3.x Syntax



As a result, it is now recommended to use the kebab-case name for any key you want to use as a modifier.

# **4. $children(removed):**

The $children instance property has been removed from Vue 3.0 and is no longer supported.

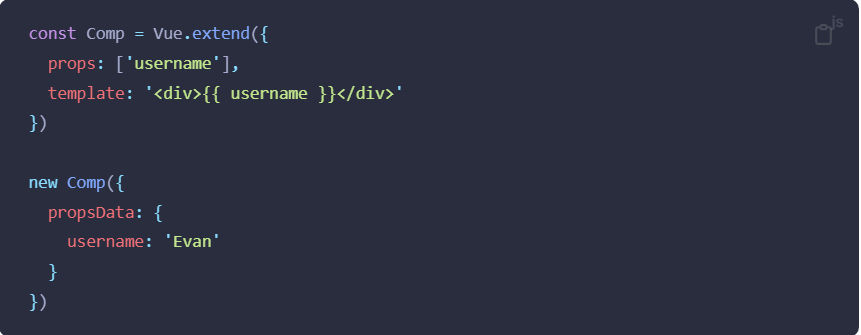
In 3.x, the $children property is removed and no longer supported. Instead, if you need to access a child component instance, we recommend using [template refs](https://vuejs.org/guide/essentials/template-refs.html#template-refs).

# **5.** [**propsData option**](https://v3-migration.vuejs.org/breaking-changes/props-data.html)

The propsData option, used to pass props to the Vue instance during its creation, is removed. To pass props to the root component of a Vue 3 application, use the second argument of [createApp](https://vuejs.org/api/application.html#createapp).

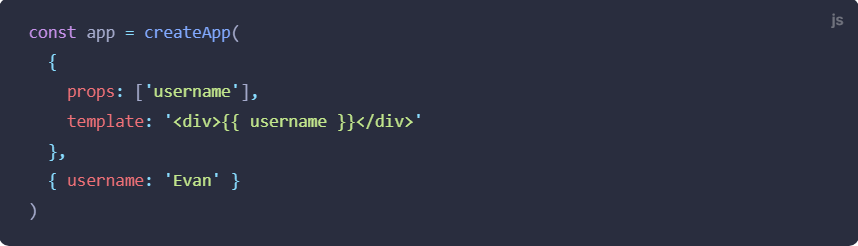
**2.x Syntax**

In 2.x, we were able to pass props to a Vue instance during its creation:

****

**3.x Update**

The propsData option has been removed. If you need to pass props to the root component instance during its creation, you should use the second argument of createApp:



### **Other Minor Changes**

# **1. A few hooks renamed:**

The **destroyed** lifecycle option has been renamed to **unmounted**

The **beforeDestroy** lifecycle option has been renamed to **beforeUnmount**

**2. Custom Directives:**

**2.x Syntax**

In Vue 2, custom directives were created by using the hooks listed below to target an element’s lifecycle, all of which are optional:

* **bind** - Called once the directive is bound to the element. Called only once.
* **inserted**- Called once the element is inserted into the parent DOM.
* **update** - This hook is called when the element updates, but children haven't been updated yet.
* **componentUpdated**- This hook is called once the component and the children have been updated.
* **unbind**- This hook is called once the directive is removed. Also called only once.

**3.x Update**

In Vue 3, however, we’ve created a more cohesive API for custom directives. As you can see, they differ greatly from our component lifecycle methods even though we’re hooking into similar events. We’ve now unified them like so:

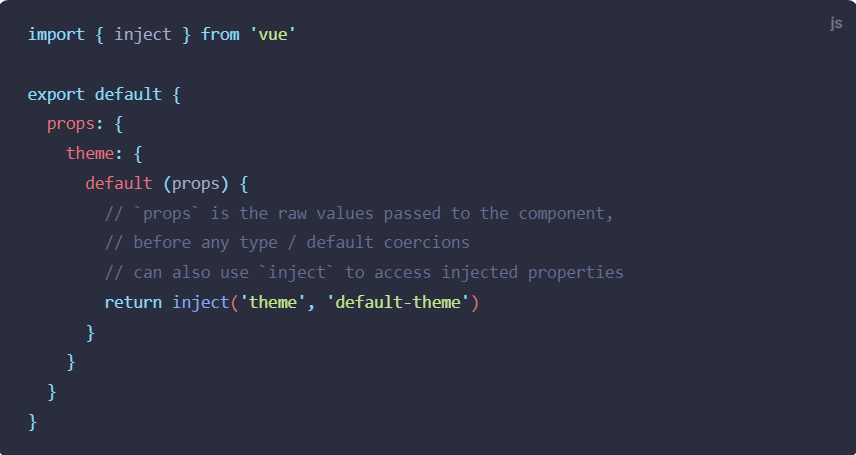
* **created -** new! This is called before the element's attributes or event listeners are applied.
* **bind → beforeMount**
* **inserted → mounted**
* **beforeUpdate:** new! This is called before the element itself is updated, much like the component lifecycle hooks.
* **update →** removed! There were too many similarities to updated, so this is redundant. Please use updated instead.
* **componentUpdated → updated**
* **beforeUnmount:** new! Similar to component lifecycle hooks, this will be called right before an element is unmounted.
* **unbind ->**unmounted

**3. Props Default Function this Access**

Props default value factory functions no longer have access to this

Instead:

* Raw props received by the component are passed to the default function as argument;
* The [inject](https://vuejs.org/api/composition-api-dependency-injection.html#inject) API can be used inside default functions.



**4. Data Option:**

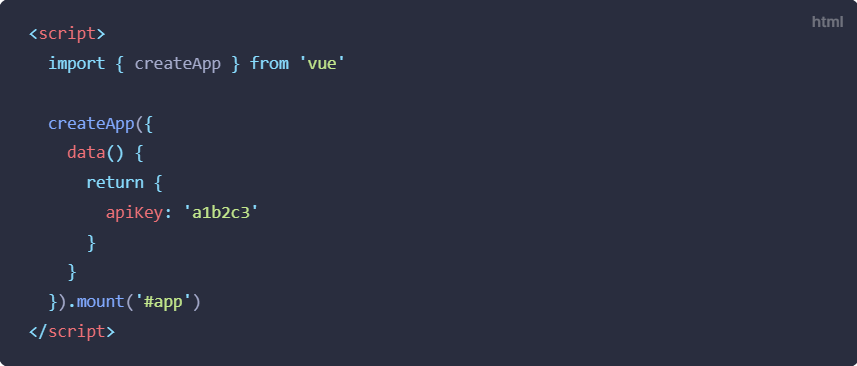
**2.x Syntax**

In 2.x, developers could define the data option with either an object or a function.



**3.x Update**

In 3.x, the data option has been standardized to only accept a function that returns an object.



## Mixin Merge Behavior Change

When data() from a component and its mixins or extends base are merged, the merge is now performed shallowly:



**2.x Result**



**3.x Result**

****

# [**Some transition classes got a rename**](https://v3-migration.vuejs.org/breaking-changes/transition.html)

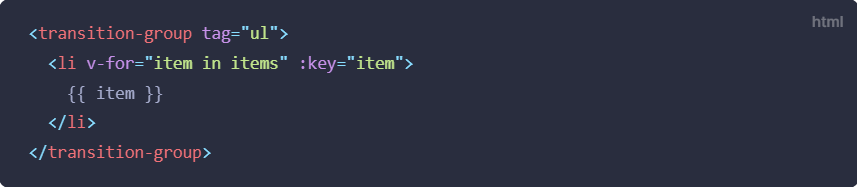
# The v-enter transition class has been renamed to v-enter-from and the v-leave transition class has been renamed to v-leave-from.

# [**<TransitionGroup> now renders no wrapper element by default**](https://v3-migration.vuejs.org/breaking-changes/transition-group.html)

<transition-group> no longer renders a root element by default, but can still create one with the tag attribute.

**2.x Syntax**

In Vue 2, <transition-group>, like other custom components, needed a root element, which by default was a <span> but was customizable via the tag attribute



**3.x Syntax**

In Vue 3, we have [fragment support](https://v3-migration.vuejs.org/new/fragments), so components no longer *need* a root node. Consequently, <transition-group> no longer renders one by default.

* If you already have the tag attribute defined in your Vue 2 code, like in the example above, everything will work as before
* If you didn't have one defined *and* your styling or other behaviors relied on the presence of the <span> root element to work properly, simply add tag="span" to the <transition-group>:

****

* 1. **Watch on Arrays:**

When using the watch option to watch an array, the callback will only trigger when the array is replaced. In other words, the watch callback will no longer be triggered on array mutation. To trigger on mutation, the deep option must be specified.

# **<template> tags with directives updates**

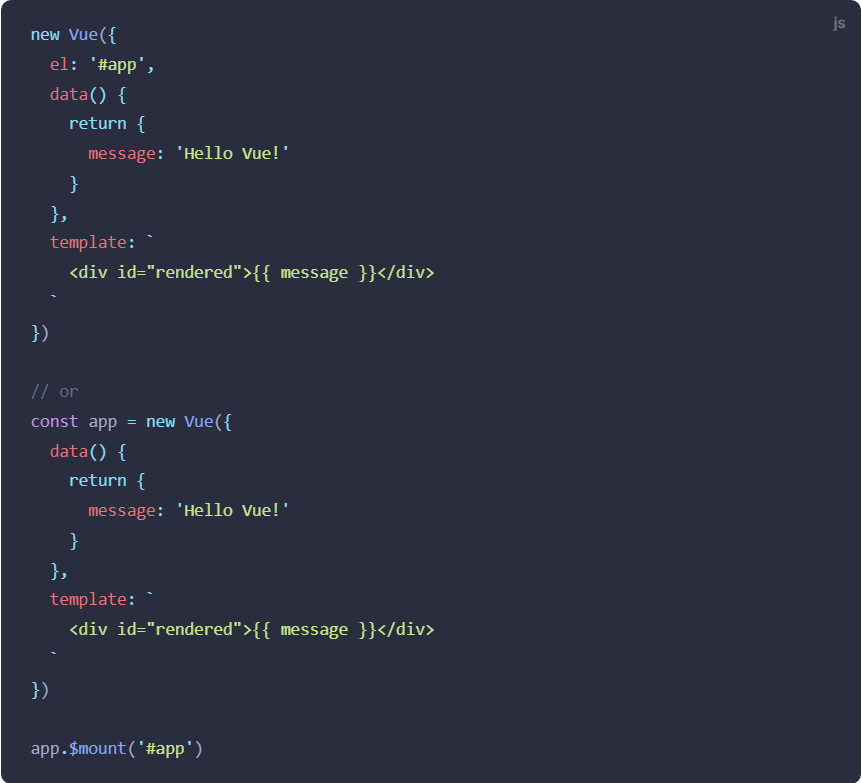
# <template> tags with no special directives (v-if/else-if/else, v-for, or v-slot) are now treated as plain elements and will result in a native <template> element instead of rendering its inner content.

# [**Mounted application does not replace the element it's mounted to**](https://v3-migration.vuejs.org/breaking-changes/mount-changes.html)**:**

In Vue 2.x, when mounting an application that has a template, the rendered content replaces the element we mount to. In Vue 3.x, the rendered application is appended as a child of such an element, replacing element's innerHTML.

**2.x Syntax**

In Vue 2.x, we pass an HTML element selector to new Vue() or $mount:



When we mount this application to the page that has a div with the passed selector (in our case, it's id="app"):

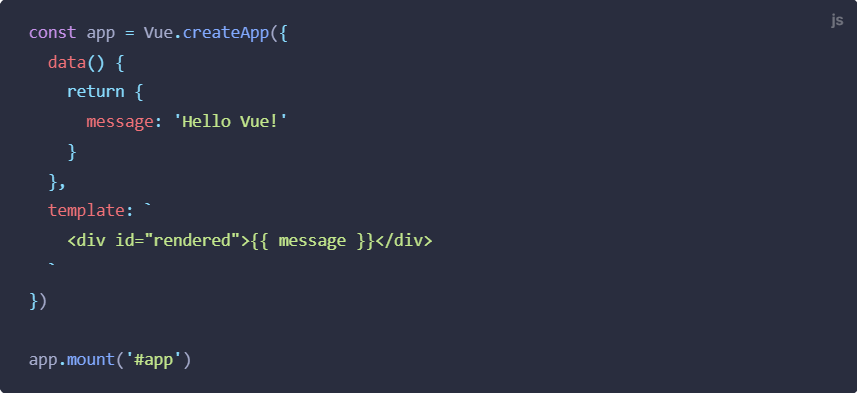


in the rendered result, the mentioned div will be replaced with the rendered application content:

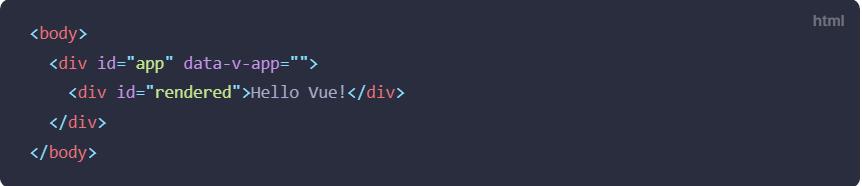


**3.x Syntax**

In Vue 3.x, when we mount an application, its rendered content will replace the innerHTML of the element we pass to mount:



When this app is mounted to the page that has a div with id="app", this will result in:



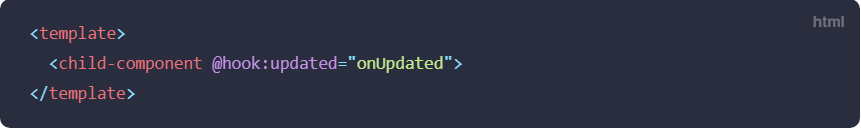
# **VNode Lifecycle Events:**

In Vue 2, it was possible to use events to listen for key stages in a component's lifecycle. These events had names that started with the prefix hook:, followed by the name of the corresponding lifecycle hook.

In Vue 3, this prefix has been changed to vue:. In addition, these events are now available for HTML elements as well as components.

**2.x Syntax**

In Vue 2, the event name is the same as the equivalent lifecycle hook, prefixed with hook::

****

**3.x Update**

In Vue 3, the event name is prefixed with vue:

****