! You're browsing the documentation for v2.x and earlier. For v3.x, **click here [https://v3.vuejs.org/]** .

API

Global Config [#Global-Config]

`Vue.config` is an object containing Vue's global configurations. You can modify its properties listed below before bootstrapping your application:

# silent [#silent]

- **Type:** `boolean`
- **Default:** `false`
- **Usage:**

  ```js
  Vue.config.silent = true
  ```

  Suppress all Vue logs and warnings.

# optionMergeStrategies [#optionMergeStrategies]

- **Type:** `{ [key: string]: Function }`
- **Default:** `{}`
- **Usage:**

```
JS
Vue.config.optionMergeStrategies._my_option = function (parent, child,
  return child + 1
}

const Profile = Vue.extend({
  _my_option: 1
})

// Profile.options._my_option = 2
```

Define custom merging strategies for options.

The merge strategy receives the value of that option defined on the parent and child instances as the first and second arguments, respectively. The context Vue instance is passed as the third argument.

- **See also:** Custom Option Merging Strategies [../guide/mixins.html#Custom-Option-Merge-Strategies]

# devtools [#devtools]

- **Type:** `boolean`
- **Default:** `true` (`false` in production builds)
- **Usage:**

```
JS
// make sure to set this synchronously immediately after loading Vue
Vue.config.devtools = true
```

Configure whether to allow vue-devtools [https://github.com/vuejs/vue-devtools] inspection. This option's default value is `true` in development builds and `false` in production builds. You can set it to `true` to enable inspection for production builds.

# errorHandler [#errorHandler]

- **Type:** `Function`
- **Default:** `undefined`
- **Usage:**

```JS
Vue.config.errorHandler = function (err, vm, info) {
  // handle error
  // `info` is a Vue-specific error info, e.g. which lifecycle hook
  // the error was found in. Only available in 2.2.0+
}
```

Assign a handler for uncaught errors during component render function and watchers. The handler gets called with the error and the Vue instance.

> **In 2.2.0+, this hook also captures errors in component lifecycle hooks. Also, when this hook is `undefined`, captured errors will be logged with `console.error` instead of crashing the app.**

> **In 2.4.0+, this hook also captures errors thrown inside Vue custom event handlers.**

> **In 2.6.0+, this hook also captures errors thrown inside `v-on` DOM listeners. In addition, if any of the covered hooks or handlers returns a Promise chain (e.g. async functions), the error from that Promise chain will also be handled.**

> **Error tracking services Sentry [https://sentry.io/for/vue/] and Bugsnag [https://docs.bugsnag.com/platforms/browsers/vue/] provide official integrations using this option.**

# warnHandler [#warnHandler]

> **New in 2.4.0+**

- **Type:** `Function`

- **Default:** `undefined`
- **Usage:**

```js
Vue.config.warnHandler = function (msg, vm, trace) {
  // `trace` is the component hierarchy trace
}
```

Assign a custom handler for runtime Vue warnings. Note this only works during development and is ignored in production.

# ignoredElements [#ignoredElements]

- **Type:** `Array<string | RegExp>`
- **Default:** `[]`
- **Usage:**

```js
Vue.config.ignoredElements = [
  'my-custom-web-component',
  'another-web-component',
  // Use a `RegExp` to ignore all elements that start with "ion-"
  // 2.5+ only
  /^ion-/
]
```

Make Vue ignore custom elements defined outside of Vue (e.g., using the Web Components APIs). Otherwise, it will throw a warning about an `Unknown custom element`, assuming that you forgot to register a global component or misspelled a component name.

# keyCodes [#keyCodes]

- **Type:** `{ [key: string]: number | Array<number> }`

- **Default:** `{}`

- **Usage:**

JS
```js
Vue.config.keyCodes = {
  v: 86,
  f1: 112,
  // camelCase won`t work
  mediaPlayPause: 179,
  // instead you can use kebab-case with double quotation marks
  "media-play-pause": 179,
  up: [38, 87]
}
```

HTML
```html
<input type="text" @keyup.media-play-pause="method">
```

Define custom key alias(es) for `v-on` .

# performance [#performance]

> **New in 2.2.0+**

- **Type:** `boolean`

- **Default:** `false (from 2.2.3+)`

- **Usage:**

Set this to `true` to enable component init, compile, render and patch performance tracing in the browser devtool performance/timeline panel. Only works in development mode and in browsers that support the performance.mark [https://developer.mozilla.org/en-US/docs/Web/API/Performance/mark] API.

# productionTip [#productionTip]

> **New in 2.2.0+**

- **Type:** `boolean`
- **Default:** `true`
- **Usage:**

  Set this to `false` to prevent the production tip on Vue startup.

Global API [#Global-API]

# Vue.extend(options) [#Vue-extend]

- **Arguments:**
  - `{Object} options`
- **Usage:**

  Create a “subclass” of the base Vue constructor. The argument should be an object containing component options.

  The special case to note here is the `data` option - it must be a function when used with `Vue.extend()` .

  HTML
  ```
  <div id="mount-point"></div>
  ```

  JS
  ```
  // create constructor
  var Profile = Vue.extend({
    template: '<p>{{firstName}} {{lastName}} aka {{alias}}</p>',
    data: function () {
      return {
  ```

```
      firstName: 'Walter',
      lastName: 'White',
      alias: 'Heisenberg'
    }
  }
})
// create an instance of Profile and mount it on an element
new Profile().$mount('#mount-point')
```

Will result in:

HTML

```html
<p>Walter White aka Heisenberg</p>
```

- **See also:** Components [../guide/components.html]

# Vue.nextTick([callback, context]) [#Vue-nextTick]

- **Arguments:**
  - `{Function} [callback]`
  - `{Object} [context]`
- **Usage:**

  Defer the callback to be executed after the next DOM update cycle. Use it immediately after you've changed some data to wait for the DOM update.

  JS

```js
// modify data
vm.msg = 'Hello'
// DOM not updated yet
Vue.nextTick(function () {
  // DOM updated
})

// usage as a promise (2.1.0+, see note below)
Vue.nextTick()
```

```
  .then(function () {
    // DOM updated
  })
```

> **New in 2.1.0+: returns a Promise if no callback is provided and Promise is supported in the execution environment. Please note that Vue does not come with a Promise polyfill, so if you target browsers that don't support Promises natively (looking at you, IE), you will have to provide a polyfill yourself.**

- **See also:** Async Update Queue [../guide/reactivity.html#Async-Update-Queue]

# Vue.set(target, propertyName/index, value) [#Vue-set]

- **Arguments:**
  - `{Object | Array} target`
  - `{string | number} propertyName/index`
  - `{any} value`
- **Returns:** the set value.
- **Usage:**

  Adds a property to a reactive object, ensuring the new property is also reactive, so triggers view updates. This must be used to add new properties to reactive objects, as Vue cannot detect normal property additions (e.g. `this.myObject.newProperty = 'hi'`).

  > ! The target object cannot be a Vue instance, or the root data object of a Vue instance.

- **See also:** Reactivity in Depth [../guide/reactivity.html]

# Vue.delete(target, propertyName/index) [#Vue-delete]

- **Arguments:**
  - `{Object | Array} target`
  - `{string | number} propertyName/index`

  > **Only in 2.2.0+: Also works with Array + index.**

- **Usage:**

  Delete a property on an object. If the object is reactive, ensure the deletion triggers view updates. This is primarily used to get around the limitation that Vue cannot detect property deletions, but you should rarely need to use it.

  > ! The target object cannot be a Vue instance, or the root data object of a Vue instance.

- **See also:** Reactivity in Depth [../guide/reactivity.html]

# Vue.directive(id, [definition]) [#Vue-directive]

- **Arguments:**
  - `{string} id`
  - `{Function | Object} [definition]`

- **Usage:**

  Register or retrieve a global directive.

  JS
  ```
  // register
  Vue.directive('my-directive', {
    bind: function () {},
    inserted: function () {},
    update: function () {},
    componentUpdated: function () {},
    unbind: function () {}
  })
  ```

```
// register (function directive)
Vue.directive('my-directive', function () {
  // this will be called as `bind` and `update`
})

// getter, return the directive definition if registered
var myDirective = Vue.directive('my-directive')
```

- **See also:** Custom Directives [../guide/custom-directive.html]

# Vue.filter(id, [definition]) [#Vue-filter]

- **Arguments:**
  - `{string} id`
  - `{Function} [definition]`
- **Usage:**

  Register or retrieve a global filter.

  JS

```
// register
Vue.filter('my-filter', function (value) {
  // return processed value
})

// getter, return the filter if registered
var myFilter = Vue.filter('my-filter')
```

- **See also:** Filters [../guide/filters.html]

# Vue.component(id, [definition]) [#Vue-component]

- **Arguments:**
  - `{string} id`
  - `{Function | Object} [definition]`
- **Usage:**

  Register or retrieve a global component. Registration also automatically sets the component's `name` with the given `id` .

  JS
  ```js
  // register an extended constructor
  Vue.component('my-component', Vue.extend({ /* ... */ }))

  // register an options object (automatically call Vue.extend)
  Vue.component('my-component', { /* ... */ })

  // retrieve a registered component (always return constructor)
  var MyComponent = Vue.component('my-component')
  ```
- **See also:** Components [../guide/components.html]

# Vue.use(plugin) [#Vue-use]

- **Arguments:**
  - `{Object | Function} plugin`
- **Usage:**

  Install a Vue.js plugin. If the plugin is an Object, it must expose an `install` method. If it is a function itself, it will be treated as the install method. The install method will be called with Vue as the argument.

  This method has to be called before calling `new Vue()`

  When this method is called on the same plugin multiple times, the plugin will be installed only once.
- **See also:** Plugins [../guide/plugins.html]

# Vue.mixin(mixin) [#Vue-mixin]

- **Arguments:**
  - `{Object} mixin`
- **Usage:**

  Apply a mixin globally, which affects every Vue instance created afterwards. This can be used by plugin authors to inject custom behavior into components. **Not recommended in application code.**

- **See also:** Global Mixin [../guide/mixins.html#Global-Mixin]

# Vue.compile(template) [#Vue-compile]

- **Arguments:**
  - `{string} template`
- **Usage:**

  Compiles a template string into a render function. **Only available in the full build.**

  JS
  ```
  var res = Vue.compile('<div><span>{{ msg }}</span></div>')

  new Vue({
    data: {
      msg: 'hello'
    },
    render: res.render,
    staticRenderFns: res.staticRenderFns
  })
  ```

- **See also:** Render Functions [../guide/render-function.html]

# Vue.observable(object) [#Vue-observable]

> **New in 2.6.0+**

- **Arguments:**
  - `{Object} object`
- **Usage:**

  Make an object reactive. Internally, Vue uses this on the object returned by the `data` function.

  The returned object can be used directly inside **render functions [../guide/render-function.html]** and **computed properties [../guide/computed.html]** , and will trigger appropriate updates when mutated. It can also be used as a minimal, cross-component state store for simple scenarios:

  JS
  ```js
  const state = Vue.observable({ count: 0 })

  const Demo = {
    render(h) {
      return h('button', {
        on: { click: () => { state.count++ }}
      }, `count is: ${state.count}`)
    }
  }
  ```

  > ! In Vue 2.x, `Vue.observable` directly mutates the object passed to it, so that it is equivalent to the object returned, as **demonstrated here [../guide/instance.html#Data-and-Methods]** . In Vue 3.x, a reactive proxy will be returned instead, leaving the original object non-reactive if mutated directly. Therefore, for future compatibility, we recommend always working with the object returned by `Vue.observable` , rather than the object originally passed to it.

- **See also:** **Reactivity in Depth [../guide/reactivity.html]**

# Vue.version [#Vue-version]

- **Details**: Provides the installed version of Vue as a string. This is especially useful for community plugins and components, where you might use different strategies for different versions.

- **Usage:**

```js
var version = Number(Vue.version.split('.')[0])

if (version === 2) {
  // Vue v2.x.x
} else if (version === 1) {
  // Vue v1.x.x
} else {
  // Unsupported versions of Vue
}
```

Options / Data [#Options-Data]

# data [#data]

- **Type:** `Object | Function`

- **Restriction:** Only accepts `Function` when used in a component definition.

- **Details:**

  The data object for the Vue instance. Vue will recursively convert its properties into getter/setters to make it "reactive". **The object must be plain**: native objects such as browser API objects and prototype properties are ignored. A rule of thumb is that data should just be data - it is not recommended to observe objects with their own stateful behavior.

Once observed, you can no longer add reactive properties to the root data object. It is therefore recommended to declare all root-level reactive properties upfront, before creating the instance.

After the instance is created, the original data object can be accessed as `vm.$data`. The Vue instance also proxies all the properties found on the data object, so `vm.a` will be equivalent to `vm.$data.a`.

Properties that start with `_` or `$` will **not** be proxied on the Vue instance because they may conflict with Vue's internal properties and API methods. You will have to access them as `vm.$data._property`.

When defining a **component**, `data` must be declared as a function that returns the initial data object, because there will be many instances created using the same definition. If we use a plain object for `data`, that same object will be **shared by reference** across all instances created! By providing a `data` function, every time a new instance is created we can call it to return a fresh copy of the initial data.

If required, a deep clone of the original object can be obtained by passing `vm.$data` through `JSON.parse(JSON.stringify(...))`.

- **Example:**

JS
```js
var data = { a: 1 }

// direct instance creation
var vm = new Vue({
  data: data
})
vm.a // => 1
vm.$data === data // => true

// must use function when in Vue.extend()
var Component = Vue.extend({
  data: function () {
    return { a: 1 }
  }
})
```

Note that if you use an arrow function with the `data` property, `this` won't be the component's instance, but you can still access the instance as the function's first argument:

JS

```
data: vm => ({ a: vm.myProp })
```

- **See also:** **Reactivity in Depth [../guide/reactivity.html]**

# props [#props]

- **Type:** `Array<string> | Object`

- **Details:**

  A list/hash of attributes that are exposed to accept data from the parent component. It has an Array-based simple syntax and an alternative Object-based syntax that allows advanced configurations such as type checking, custom validation and default values.

  With Object-based syntax, you can use following options:

  - `type` : can be one of the following native constructors: `String` , `Number` , `Boolean` , `Array` , `Object` , `Date` , `Function` , `Symbol` , any custom constructor function or an array of those. Will check if a prop has a given type, and will throw a warning if it doesn't. **More information [../guide/components-props.html#Prop-Types]** on prop types.
  - `default` : `any`
    Specifies a default value for the prop. If the prop is not passed, this value will be used instead. Object or array defaults must be returned from a factory function.
  - `required` : `Boolean`
    Defines if the prop is required. In a non-production environment, a console warning will be thrown if this value is truthy and the prop is not passed.
  - `validator` : `Function`
    Custom validator function that takes the prop value as the sole argument. In a non-production environment, a console warning will be thrown if this function returns a falsy value (i.e. the validation fails). You can read more about prop validation **here [../guide/components-props.html#Prop-Validation]** .

- **Example:**

JS

```
// simple syntax
Vue.component('props-demo-simple', {
  props: ['size', 'myMessage']
})
```

```
// object syntax with validation
Vue.component('props-demo-advanced', {
  props: {
    // type check
    height: Number,
    // type check plus other validations
    age: {
      type: Number,
      default: 0,
      required: true,
      validator: function (value) {
        return value >= 0
      }
    }
  }
})
```

- **See also:** Props [../guide/components-props.html]

# propsData [#propsData]

- **Type:** `{ [key: string]: any }`
- **Restriction:** only respected in instance creation via `new` .
- **Details:**

  Pass props to an instance during its creation. This is primarily intended to make unit testing easier.

- **Example:**

JS

```
var Comp = Vue.extend({
  props: ['msg'],
  template: '<div>{{ msg }}</div>'
})
```

```
var vm = new Comp({
  propsData: {
    msg: 'hello'
  }
})
```

# computed [#computed]

- **Type:** `{ [key: string]: Function | { get: Function, set: Function } }`

- **Details:**

  Computed properties to be mixed into the Vue instance. All getters and setters have their `this` context automatically bound to the Vue instance.

  Note that if you use an arrow function with a computed property, `this` won't be the component's instance, but you can still access the instance as the function's first argument:

  JS
  ```
  computed: {
    aDouble: vm => vm.a * 2
  }
  ```

  Computed properties are cached, and only re-computed on reactive dependency changes. Note that if a certain dependency is out of the instance's scope (i.e. not reactive), the computed property will **not** be updated.

- **Example:**

  JS
  ```
  var vm = new Vue({
    data: { a: 1 },
    computed: {
      // get only
      aDouble: function () {
        return this.a * 2
      },
  ```

```
        // both get and set
        aPlus: {
          get: function () {
            return this.a + 1
          },
          set: function (v) {
            this.a = v - 1
          }
        }
      }
    })
    vm.aPlus   // => 2
    vm.aPlus = 3
    vm.a       // => 2
    vm.aDouble // => 4
```

- **See also: Computed Properties [../guide/computed.html]**

# methods [#methods]

- **Type:** `{ [key: string]: Function }`

- **Details:**

  Methods to be mixed into the Vue instance. You can access these methods directly on the VM instance, or use them in directive expressions. All methods will have their `this` context automatically bound to the Vue instance.

  > ! Note that **you should not use an arrow function to define a method** (e.g. `plus: () => this.a++`). The reason is arrow functions bind the parent context, so `this` will not be the Vue instance as you expect and `this.a` will be undefined.

- **Example:**

JS
```
var vm = new Vue({
  data: { a: 1 },
  methods: {
    plus: function () {
      this.a++
    }
  }
})
vm.plus()
vm.a // 2
```

- **See also:** Event Handling [../guide/events.html]

# watch [#watch]

- **Type:** `{ [key: string]: string | Function | Object | Array}`
- **Details:**

  An object where keys are expressions to watch and values are the corresponding callbacks. The value can also be a string of a method name, or an Object that contains additional options. The Vue instance will call `$watch()` for each entry in the object at instantiation.

- **Example:**

JS
```
var vm = new Vue({
  data: {
    a: 1,
    b: 2,
    c: 3,
    d: 4,
    e: {
      f: {
        g: 5
      }
    }
  },
  watch: {
    a: function (val, oldVal) {
      console.log('new: %s, old: %s', val, oldVal)
    },
    // string method name
    b: 'someMethod',
    // the callback will be called whenever any of the watched object p
    c: {
      handler: function (val, oldVal) { /* ... */ },
      deep: true
    },
    // the callback will be called immediately after the start of the c
    d: {
      handler: 'someMethod',
      immediate: true
    },
    // you can pass array of callbacks, they will be called one-by-one
    e: [
      'handle1',
      function handle2 (val, oldVal) { /* ... */ },
      {
        handler: function handle3 (val, oldVal) { /* ... */ },
        /* ... */
      }
    ],
    // watch vm.e.f's value: {g: 5}
    'e.f': function (val, oldVal) { /* ... */ }
  }
})
vm.a = 2 // => new: 2, old: 1
```

!

Note that **you should not use an arrow function to define a watcher** (e.g. `searchQuery: newValue => this.updateAutocomplete(newValue)`). The reason is arrow functions bind the parent context, so `this` will not be the Vue instance as you expect and `this.updateAutocomplete` will be undefined.

- **See also:** Instance Methods / Data - vm.$watch [#vm-watch]

Options / DOM [#Options-DOM]

# el [#el]

- **Type:** `string | Element`
- **Restriction:** only respected in instance creation via `new`.
- **Details:**

  Provide the Vue instance an existing DOM element to mount on. It can be a CSS selector string or an actual HTMLElement.

  After the instance is mounted, the resolved element will be accessible as `vm.$el`.

  If this option is available at instantiation, the instance will immediately enter compilation; otherwise, the user will have to explicitly call `vm.$mount()` to manually start the compilation.

  > ! The provided element merely serves as a mounting point. Unlike in Vue 1.x, the mounted element will be replaced with Vue-generated DOM in all cases. It is therefore not recommended to mount the root instance to `<html>` or `<body>`.

  > ! If neither `render` function nor `template` option is present, the in-DOM HTML of the mounting DOM element will be extracted as the template. In this case, Runtime + Compiler build of Vue should be used.

- **See also:**
  - Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]
  - Runtime + Compiler vs. Runtime-only [../guide/installation.html#Runtime-Compiler-vs-Runtime-only]

# template [#template]

- **Type:** `string`
- **Details:**

  A string template to be used as the markup for the Vue instance. The template will **replace** the mounted element. Any existing markup inside the mounted element will be ignored, unless content distribution slots are present in the template.

  If the string starts with `#` it will be used as a querySelector and use the selected element's innerHTML as the template string. This allows the use of the common `<script type="x-template">` trick to include templates.

  > ! From a security perspective, you should only use Vue templates that you can trust. Never use user-generated content as your template.

  > ! If render function is present in the Vue option, the template will be ignored.

- **See also:**
  - Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]
  - Content Distribution with Slots [../guide/components.html#Content-Distribution-with-Slots]

# render [#render]

- **Type:** `(createElement: () => VNode) => VNode`
- **Details:**

  An alternative to string templates allowing you to leverage the full programmatic power of JavaScript. The render function receives a `createElement` method as it's first argument used to create `VNode` s.

If the component is a functional component, the render function also receives an extra argument `context`, which provides access to contextual data since functional components are instance-less.

> ! The `render` function has priority over the render function compiled from `template` option or in-DOM HTML template of the mounting element which is specified by the `el` option.

- **See also:** Render Functions [../guide/render-function.html]

# renderError [#renderError]

> **New in 2.2.0+**

- **Type:** `(createElement: () => VNode, error: Error) => VNode`

- **Details:**

  Only works in development mode.

  Provide an alternative render output when the default `render` function encounters an error. The error will be passed to `renderError` as the second argument. This is particularly useful when used together with hot-reload.

- **Example:**

  JS
  ```
  new Vue({
    render (h) {
      throw new Error('oops')
    },
    renderError (h, err) {
      return h('pre', { style: { color: 'red' }}, err.stack)
    }
  }).$mount('#app')
  ```

- **See also:** Render Functions [../guide/render-function.html]

Options / Lifecycle Hooks [#Options-Lifecycle-Hooks]

All lifecycle hooks automatically have their `this` context bound to the instance, so that you can access data, computed properties, and methods. This means **you should not use an arrow function to define a lifecycle method** (e.g. `created: () => this.fetchTodos()`). The reason is arrow functions bind the parent context, so `this` will not be the Vue instance as you expect and `this.fetchTodos` will be undefined.

# beforeCreate [#beforeCreate]

- **Type:** `Function`
- **Details:**

  Called synchronously immediately after the instance has been initialized, before data observation and event/watcher setup.

- **See also:** Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]

# created [#created]

- **Type:** `Function`
- **Details:**

  Called synchronously after the instance is created. At this stage, the instance has finished processing the options which means the following have been set up: data observation, computed properties, methods, watch/event callbacks. However, the mounting phase has not been started, and the `$el` property will not be available yet.

- **See also:** Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]

# beforeMount [#beforeMount]

- **Type:** `Function`
- **Details:**

  Called right before the mounting begins: the `render` function is about to be called for the first time.

  This hook is not called during server-side rendering.

- **See also:** Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]

# mounted [#mounted]

- **Type:** `Function`
- **Details:**

  Called after the instance has been mounted, where `el` is replaced by the newly created `vm.$el`. If the root instance is mounted to an in-document element, `vm.$el` will also be in-document when `mounted` is called.

  Note that `mounted` does **not** guarantee that all child components have also been mounted. If you want to wait until the entire view has been rendered, you can use vm.$nextTick [#vm-nextTick] inside of `mounted`:

  JS
  ```
  mounted: function () {
    this.$nextTick(function () {
      // Code that will run only after the
      // entire view has been rendered
    })
  }
  ```

  This hook is not called during server-side rendering.

- **See also:** Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]

# beforeUpdate [#beforeUpdate]

- **Type:** `Function`
- **Details:**

  Called when data changes, before the DOM is patched. This is a good place to access the existing DOM before an update, e.g. to remove manually added event listeners.

  This hook is not called during server-side rendering, because only the initial render is performed server-side.

- **See also:** Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]

# updated [#updated]

- **Type:** `Function`
- **Details:**

  Called after a data change causes the virtual DOM to be re-rendered and patched.

  The component's DOM will have been updated when this hook is called, so you can perform DOM-dependent operations here. However, in most cases you should avoid changing state inside the hook. To react to state changes, it's usually better to use a computed property [#computed] or watcher [#watch] instead.

  Note that `updated` does **not** guarantee that all child components have also been re-rendered. If you want to wait until the entire view has been re-rendered, you can use vm.$nextTick [#vm-nextTick] inside of `updated` :

  JS
  ```
  updated: function () {
    this.$nextTick(function () {
      // Code that will run only after the
      // entire view has been re-rendered
    })
  }
  ```

  This hook is not called during server-side rendering.

- **See also:** **Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]**

# activated [#activated]

- **Type:** `Function`
- **Details:**

  Called when a kept-alive component is activated.

  This hook is not called during server-side rendering.

- **See also:**
  - **Built-in Components - keep-alive [#keep-alive]**
  - **Dynamic Components - keep-alive [../guide/components.html#keep-alive]**

# deactivated [#deactivated]

- **Type:** `Function`
- **Details:**

  Called when a kept-alive component is deactivated.

  This hook is not called during server-side rendering.

- **See also:**
  - **Built-in Components - keep-alive [#keep-alive]**
  - **Dynamic Components - keep-alive [../guide/components.html#keep-alive]**

# beforeDestroy [#beforeDestroy]

- **Type:** `Function`

- **Details:**

  Called right before a Vue instance is destroyed. At this stage the instance is still fully functional.

  This hook is not called during server-side rendering.

- **See also:** Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]

# destroyed [#destroyed]

- **Type:** `Function`

- **Details:**

  Called after a Vue instance has been destroyed. When this hook is called, all directives of the Vue instance have been unbound, all event listeners have been removed, and all child Vue instances have also been destroyed.

  This hook is not called during server-side rendering.

- **See also:** Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]

# errorCaptured [#errorCaptured]

> **New in 2.5.0+**

- **Type:** `(err: Error, vm: Component, info: string) => ?boolean`

- **Details:**

  Called when an error from any descendent component is captured. The hook receives three arguments: the error, the component instance that triggered the error, and a string containing information on where the error was captured. The hook can return `false` to stop the error from propagating further.

> ! You can modify component state in this hook. However, it is important to have conditionals in your template or render function that short circuits other content when an error has been captured; otherwise the component will be thrown into an infinite render loop.

Error Propagation Rules

- By default, all errors are still sent to the global `config.errorHandler` if it is defined, so that these errors can still be reported to an analytics service in a single place.
- If multiple `errorCaptured` hooks exist on a component's inheritance chain or parent chain, all of them will be invoked on the same error.
- If the `errorCaptured` hook itself throws an error, both this error and the original captured error are sent to the global `config.errorHandler` .
- An `errorCaptured` hook can return `false` to prevent the error from propagating further. This is essentially saying “this error has been handled and should be ignored.” It will prevent any additional `errorCaptured` hooks or the global `config.errorHandler` from being invoked for this error.

Options / Assets [#Options-Assets]

# directives [#directives]

- **Type:** `Object`
- **Details:**

  A hash of directives to be made available to the Vue instance.

- **See also:** Custom Directives [../guide/custom-directive.html]

# filters [#filters]

- **Type:** `Object`
- **Details:**

  A hash of filters to be made available to the Vue instance.

- **See also:** `Vue.filter` [#Vue-filter]

# components [#components]

- **Type:** `Object`
- **Details:**

  A hash of components to be made available to the Vue instance.

- **See also:** Components [../guide/components.html]

Options / Composition [#Options-Composition]

# parent [#parent]

- **Type:** `Vue instance`
- **Details:**

  Specify the parent instance for the instance to be created. Establishes a parent-child relationship between the two. The parent will be accessible as `this.$parent` for the child, and the child will be pushed into the parent’s `$children` array.

  > ! Use `$parent` and `$children` sparingly - they mostly serve as an escape-hatch. Prefer using props and events for parent-child communication.

# mixins [#mixins]

- **Type:** `Array<Object>`

- **Details:**

  The `mixins` option accepts an array of mixin objects. These mixin objects can contain instance options like normal instance objects, and they will be merged against the eventual options using the same option merging logic in `Vue.extend()` . e.g. If your mixin contains a created hook and the component itself also has one, both functions will be called.

  Mixin hooks are called in the order they are provided, and called before the component's own hooks.

- **Example:**

  ```
  JS
  var mixin = {
    created: function () { console.log(1) }
  }
  var vm = new Vue({
    created: function () { console.log(2) },
    mixins: [mixin]
  })
  // => 1
  // => 2
  ```

- **See also:** Mixins [../guide/mixins.html]

# extends [#extends]

- **Type:** `Object | Function`

- **Details:**

  Allows declaratively extending another component (could be either a plain options object or a constructor) without having to use `Vue.extend` . This is primarily intended to make it easier to extend between single file components.

This is similar to `mixins` .

- **Example:**

  ```js
  var CompA = { ... }

  // extend CompA without having to call `Vue.extend` on either
  var CompB = {
    extends: CompA,
    ...
  }
  ```

# provide / inject [#provide-inject]

> **New in 2.2.0+**

- **Type:**
  - **provide:** `Object | () => Object`
  - **inject:** `Array<string> | { [key: string]: string | Symbol | Object }`

- **Details:**

  This pair of options are used together to allow an ancestor component to serve as a dependency injector for all its descendants, regardless of how deep the component hierarchy is, as long as they are in the same parent chain. If you are familiar with React, this is very similar to React's context feature.

  The `provide` option should be an object or a function that returns an object. This object contains the properties that are available for injection into its descendants. You can use ES2015 Symbols as keys in this object, but only in environments that natively support `Symbol` and `Reflect.ownKeys` .

  The `inject` option should be either:

  - an array of strings, or
  - an object where the keys are the local binding name and the value is either:
    - the key (string or Symbol) to search for in available injections, or

  - an object where:
    - the `from` property is the key (string or Symbol) to search for in available injections, and
    - the `default` property is used as fallback value

> **Note: the `provide` and `inject` bindings are NOT reactive. This is intentional. However, if you pass down an observed object, properties on that object do remain reactive.**

- **Example:**

JS

```
// parent component providing 'foo'
var Provider = {
  provide: {
    foo: 'bar'
  },
  // ...
}

// child component injecting 'foo'
var Child = {
  inject: ['foo'],
  created () {
    console.log(this.foo) // => "bar"
  }
  // ...
}
```

With ES2015 Symbols, function `provide` and object `inject` :

JS

```
const s = Symbol()

const Provider = {
  provide () {
    return {
      [s]: 'foo'
    }
  }
}

const Child = {
  inject: { s },
  // ...
}
```

> **The next 2 examples work with Vue 2.2.1+. Below that version, injected values were resolved after the `props` and the `data` initialization.**

Using an injected value as the default for a prop:

```js
const Child = {
  inject: ['foo'],
  props: {
    bar: {
      default () {
        return this.foo
      }
    }
  }
}
```

Using an injected value as data entry:

```js
const Child = {
  inject: ['foo'],
  data () {
    return {
      bar: this.foo
    }
  }
}
```

> **In 2.5.0+ injections can be optional with default value:**

JS
```
const Child = {
  inject: {
    foo: { default: 'foo' }
  }
}
```

If it needs to be injected from a property with a different name, use `from` to denote the source property:

JS
```
const Child = {
  inject: {
    foo: {
      from: 'bar',
      default: 'foo'
    }
  }
}
```

Similar to prop defaults, you need to use a factory function for non primitive values:

JS
```
const Child = {
  inject: {
    foo: {
      from: 'bar',
      default: () => [1, 2, 3]
    }
  }
}
```

Options / Misc [#Options-Misc]

# name [#name]

- **Type:** `string`
- **Restriction:** only respected when used as a component option.
- **Details:**

  Allow the component to recursively invoke itself in its template. Note that when a component is registered globally with `Vue.component()`, the global ID is automatically set as its name.

  Another benefit of specifying a `name` option is debugging. Named components result in more helpful warning messages. Also, when inspecting an app in the **vue-devtools [https://github.com/vuejs/vue-devtools]**, unnamed components will show up as `<AnonymousComponent>`, which isn't very informative. By providing the `name` option, you will get a much more informative component tree.

# delimiters [#delimiters]

- **Type:** `Array<string>`
- **Default:** `["{{", "}}"]`
- **Restrictions:** This option is only available in the full build, with in-browser compilation.
- **Details:**

  Change the plain text interpolation delimiters.

- **Example:**

```js
JS
new Vue({
  delimiters: ['${', '}']
})

// Delimiters changed to ES6 template string style
```

# functional [#functional]

- **Type:** `boolean`
- **Details:**

  Causes a component to be stateless (no `data`) and instanceless (no `this` context). They are only a `render` function that returns virtual nodes making them much cheaper to render.

- **See also:** Functional Components [../guide/render-function.html#Functional-Components]

# model [#model]

> **New in 2.2.0**

- **Type:** `{ prop?: string, event?: string }`
- **Details:**

  Allows a custom component to customize the prop and event used when it's used with `v-model` . By default, `v-model` on a component uses `value` as the prop and `input` as the event, but some input types such as checkboxes and radio buttons may want to use the `value` prop for a different purpose. Using the `model` option can avoid the conflict in such cases.

- **Example:**

```js
Vue.component('my-checkbox', {
  model: {
    prop: 'checked',
    event: 'change'
  },
  props: {
    // this allows using the `value` prop for a different purpose
    value: String,
    // use `checked` as the prop which take the place of `value`
    checked: {
      type: Number,
      default: 0
      }
    },
    // ...
  })
```

HTML

```html
<my-checkbox v-model="foo" value="some value"></my-checkbox>
```

The above will be equivalent to:

HTML

```html
<my-checkbox
  :checked="foo"
  @change="val => { foo = val }"
  value="some value">
</my-checkbox>
```

# inheritAttrs [#inheritAttrs]

> **New in 2.4.0+**

- **Type:** `boolean`
- **Default:** `true`
- **Details:**

  By default, parent scope attribute bindings that are not recognized as props will "fallthrough" and be applied to the root element of the child component as normal HTML attributes. When authoring a component that wraps a target element or another component, this may not always be the desired behavior. By setting `inheritAttrs` to `false` , this default behavior can be disabled. The attributes are available via the `$attrs` instance property (also new in 2.4) and can be explicitly bound to a non-root element using `v-bind` .

  Note: this option does **not** affect `class` and `style` bindings.

# comments [#comments]

> **New in 2.4.0+**

- **Type:** `boolean`
- **Default:** `false`
- **Restrictions:** This option is only available in the full build, with in-browser compilation.
- **Details:**

  When set to `true`, will preserve and render HTML comments found in templates. The default behavior is discarding them.

Instance Properties [#Instance-Properties]

vm.$data [#vm-data]

- **Type:** `Object`
- **Details:**

  The data object that the Vue instance is observing. The Vue instance proxies access to the properties on its data object.

- **See also:** **Options / Data - data [#data]**

vm.$props [#vm-props]

> **New in 2.2.0+**

- **Type:** `Object`

- **Details:**

  An object representing the current props a component has received. The Vue instance proxies access to the properties on its props object.

vm.$el [#vm-el]

- **Type:** `Element`
- **Read only**
- **Details:**

  The root DOM element that the Vue instance is managing.

vm.$options [#vm-options]

- **Type:** `Object`
- **Read only**
- **Details:**

  The instantiation options used for the current Vue instance. This is useful when you want to include custom properties in the options:

  ```js
  new Vue({
    customOption: 'foo',
    created: function () {
      console.log(this.$options.customOption) // => 'foo'
    }
  })
  ```

vm.$parent [#vm-parent]

- **Type:** `Vue instance`
- **Read only**
- **Details:**

  The parent instance, if the current instance has one.

vm.$root [#vm-root]

- **Type:** `Vue instance`
- **Read only**
- **Details:**

  The root Vue instance of the current component tree. If the current instance has no parents this value will be itself.

vm.$children [#vm-children]

- **Type:** `Array<Vue instance>`
- **Read only**
- **Details:**

  The direct child components of the current instance. **Note there's no order guarantee for `$children`, and it is not reactive.** If you find yourself trying to use `$children` for data binding, consider using an Array and `v-for` to generate child components, and use the Array as the source of truth.

vm.$slots [#vm-slots]

- **Type:** `{ [name: string]: ?Array<VNode> }`

- **Read only**
- **Reactive?** No
- **Details:**

  Used to programmatically access content **distributed by slots [../guide/components.html#Content-Distribution-with-Slots]** . Each **named slot [../guide/components.html#Named-Slots]** has its own corresponding property (e.g. the contents of `v-slot:foo` will be found at `vm.$slots.foo`). The `default` property contains either nodes not included in a named slot or contents of `v-slot:default` .

  Please note that slots are **not** reactive. If you need a component to re-render based on changes to data passed to a slot, we suggest considering a different strategy that relies on a reactive instance option, such as `props` or `data` .

  Note: `v-slot:foo` is supported in v2.6+. For older versions, you can use the **deprecated syntax [../guide/components-slots.html#Deprecated-Syntax]** .

  Accessing `vm.$slots` is most useful when writing a component with a **render function [../guide/render-function.html]** .

- **Example:**

HTML

```html
<blog-post>
  <template v-slot:header>
    <h1>About Me</h1>
  </template>

  <p>Here's some page content, which will be included in vm.$slots.defa

  <template v-slot:footer>
    <p>Copyright 2016 Evan You</p>
  </template>

  <p>If I have some content down here, it will also be included in vm.$
</blog-post>
```

JS

```js
Vue.component('blog-post', {
  render: function (createElement) {
    var header = this.$slots.header
```

```
        var body   = this.$slots.default
        var footer = this.$slots.footer
        return createElement('div', [
          createElement('header', header),
          createElement('main', body),
          createElement('footer', footer)
        ])
      }
    })
```

- **See also:**
  - `<slot>` **Component [#slot]**
  - **Content Distribution with Slots [../guide/components.html#Content-Distribution-with-Slots]**
  - **Render Functions - Slots [../guide/render-function.html#Slots]**

vm.$scopedSlots [#vm-scopedSlots]

> **New in 2.1.0+**

- **Type:** `{ [name: string]: props => Array<VNode> | undefined }`

- **Read only**

- **Details:**

  Used to programmatically access **scoped slots [../guide/components.html#Scoped-Slots]** . For each slot, including the `default` one, the object contains a corresponding function that returns VNodes.

  Accessing `vm.$scopedSlots` is most useful when writing a component with a **render function [../guide/render-function.html]** .

  Note: since 2.6.0+, there are two notable changes to this property:

  1. Scoped slot functions are now guaranteed to return an array of VNodes, unless the return value is invalid, in which case the function will return `undefined` .

2. All `$slots` are now also exposed on `$scopedSlots` as functions. If you work with render functions, it is now recommended to always access slots via `$scopedSlots`, whether they currently use a scope or not. This will not only make future refactors to add a scope simpler, but also ease your eventual migration to Vue 3, where all slots will be functions.

- **See also:**
  - `<slot>` Component [#slot]
  - Scoped Slots [../guide/components.html#Scoped-Slots]
  - Render Functions - Slots [../guide/render-function.html#Slots]

vm.$refs [#vm-refs]

- **Type:** `Object`
- **Read only**
- **Details:**

  An object of DOM elements and component instances, registered with `ref` attributes [#ref] .

- **See also:**
  - Child Component Refs [../guide/components.html#Child-Component-Refs]
  - Special Attributes - ref [#ref]

vm.$isServer [#vm-isServer]

- **Type:** `boolean`
- **Read only**
- **Details:**

  Whether the current Vue instance is running on the server.

- **See also:** Server-Side Rendering [../guide/ssr.html]

vm.$attrs [#vm-attrs]

> **New in 2.4.0+**

- **Type:** `{ [key: string]: string }`
- **Read only**
- **Details:**

  Contains parent-scope attribute bindings (except for `class` and `style`) that are not recognized (and extracted) as props. When a component doesn't have any declared props, this essentially contains all parent-scope bindings (except for `class` and `style`), and can be passed down to an inner component via `v-bind="$attrs"` - useful when creating higher-order components.

vm.$listeners [#vm-listeners]

> **New in 2.4.0+**

- **Type:** `{ [key: string]: Function | Array<Function> }`
- **Read only**
- **Details:**

  Contains parent-scope `v-on` event listeners (without `.native` modifiers). This can be passed down to an inner component via `v-on="$listeners"` - useful when creating transparent wrapper components.

Instance Methods / Data [#Instance-Methods-Data]

vm.$watch(expOrFn, callback, [options]) [#vm-watch]

- **Arguments:**
  - `{string | Function} expOrFn`
  - `{Function | Object} callback`
  - `{Object} [options]`
    - `{boolean} deep`
    - `{boolean} immediate`
- **Returns:** `{Function} unwatch`
- **Usage:**

  Watch an expression or a computed function on the Vue instance for changes. The callback gets called with the new value and the old value. The expression only accepts dot-delimited paths. For more complex expressions, use a function instead.

  ! Note: when mutating (rather than replacing) an Object or an Array, the old value will be the same as new value because they reference the same Object/Array. Vue doesn't keep a copy of the pre-mutate value.

- **Example:**

```
JS
// keypath
vm.$watch('a.b.c', function (newVal, oldVal) {
  // do something
})

// function
vm.$watch(
  function () {
    // every time the expression `this.a + this.b` yields a different r
    // the handler will be called. It's as if we were watching a comput
    // property without defining the computed property itself
    return this.a + this.b
  },
  function (newVal, oldVal) {
    // do something
  }
)
```

`vm.$watch` returns an unwatch function that stops firing the callback:

JS

```js
var unwatch = vm.$watch('a', cb)
// later, teardown the watcher
unwatch()
```

- **Option: deep**

  To also detect nested value changes inside Objects, you need to pass in `deep: true` in the options argument. Note that you don't need to do so to listen for Array mutations.

  JS

  ```js
  vm.$watch('someObject', callback, {
    deep: true
  })
  vm.someObject.nestedValue = 123
  // callback is fired
  ```

- **Option: immediate**

  Passing in `immediate: true` in the option will trigger the callback immediately with the current value of the expression:

  JS

  ```js
  vm.$watch('a', callback, {
    immediate: true
  })
  // `callback` is fired immediately with current value of `a`
  ```

  Note that with `immediate` option you won't be able to unwatch the given property on the first callback call.

JS

```js
// This will cause an error
var unwatch = vm.$watch(
  'value',
  function () {
    doSomething()
    unwatch()
  },
  { immediate: true }
)
```

If you still want to call an unwatch function inside the callback, you should check its availability first:

JS

```js
var unwatch = vm.$watch(
  'value',
  function () {
    doSomething()
    if (unwatch) {
      unwatch()
    }
  },
  { immediate: true }
)
```

vm.$set(target, propertyName/index, value) [#vm-set]

- **Arguments:**
  - `{Object | Array} target`
  - `{string | number} propertyName/index`
  - `{any} value`
- **Returns:** the set value.
- **Usage:**

This is the alias of the global `Vue.set` .

- **See also:** Vue.set [#Vue-set]

vm.$delete(target, propertyName/index) [#vm-delete]

- **Arguments:**
  - `{Object | Array} target`
  - `{string | number} propertyName/index`
- **Usage:**

  This is the alias of the global `Vue.delete` .

- **See also:** Vue.delete [#Vue-delete]

Instance Methods / Events [#Instance-Methods-Events]

vm.$on(event, callback) [#vm-on]

- **Arguments:**
  - `{string | Array<string>} event` (array only supported in 2.2.0+)
  - `{Function} callback`
- **Usage:**

  Listen for a custom event on the current vm. Events can be triggered by `vm.$emit` . The callback will receive all the additional arguments passed into these event-triggering methods.

- **Example:**

JS

```
vm.$on('test', function (msg) {
  console.log(msg)
})
vm.$emit('test', 'hi')
// => "hi"
```

vm.$once(event, callback) [#vm-once]

- **Arguments:**
  - `{string} event`
  - `{Function} callback`
- **Usage:**

  Listen for a custom event, but only once. The listener will be removed once it triggers for the first time.

vm.$off([event, callback]) [#vm-off]

- **Arguments:**
  - `{string | Array<string>} event` (array only supported in 2.2.2+)
  - `{Function} [callback]`
- **Usage:**

  Remove custom event listener(s).

  - If no arguments are provided, remove all event listeners;
  - If only the event is provided, remove all listeners for that event;
  - If both event and callback are given, remove the listener for that specific callback only.

vm.$emit(eventName, […args]) [#vm-emit]

- **Arguments:**
  - `{string} eventName`
  - `[...args]`

  Trigger an event on the current instance. Any additional arguments will be passed into the listener's callback function.

- **Examples:**

  Using `$emit` with only an event name:

  JS
  ```js
  Vue.component('welcome-button', {
    template: `
      <button v-on:click="$emit('welcome')">
        Click me to be welcomed
      </button>
    `
  })
  ```

  HTML
  ```html
  <div id="emit-example-simple">
    <welcome-button v-on:welcome="sayHi"></welcome-button>
  </div>
  ```

  JS
  ```js
  new Vue({
    el: '#emit-example-simple',
    methods: {
      sayHi: function () {
        alert('Hi!')
      }
    }
  })
  ```

  Click me to be welcomed

  Using `$emit` with additional arguments:

JS

```
Vue.component('magic-eight-ball', {
  data: function () {
    return {
      possibleAdvice: ['Yes', 'No', 'Maybe']
    }
  },
  methods: {
    giveAdvice: function () {
      var randomAdviceIndex = Math.floor(Math.random() * this.possibleA
      this.$emit('give-advice', this.possibleAdvice[randomAdviceIndex])
    }
  },
  template: `
    <button v-on:click="giveAdvice">
      Click me for advice
    </button>
  `
})
```

HTML

```
<div id="emit-example-argument">
  <magic-eight-ball v-on:give-advice="showAdvice"></magic-eight-ball>
</div>
```

JS

```
new Vue({
  el: '#emit-example-argument',
  methods: {
    showAdvice: function (advice) {
      alert(advice)
    }
  }
})
```

Click me for advice

Instance Methods / Lifecycle [#Instance-Methods-Lifecycle]

vm.$mount([elementOrSelector]) [#vm-mount]

- **Arguments:**
  - `{Element | string} [elementOrSelector]`
  - `{boolean} [hydrating]`
- **Returns:** `vm` - the instance itself
- **Usage:**

  If a Vue instance didn't receive the `el` option at instantiation, it will be in "unmounted" state, without an associated DOM element. `vm.$mount()` can be used to manually start the mounting of an unmounted Vue instance.

  If `elementOrSelector` argument is not provided, the template will be rendered as an off-document element, and you will have to use native DOM API to insert it into the document yourself.

  The method returns the instance itself so you can chain other instance methods after it.

- **Example:**

  JS
  ```
  var MyComponent = Vue.extend({
    template: '<div>Hello!</div>'
  })

  // create and mount to #app (will replace #app)
  new MyComponent().$mount('#app')

  // the above is the same as:
  new MyComponent({ el: '#app' })

  // or, render off-document and append afterwards:
  ```

```
var component = new MyComponent().$mount()
document.getElementById('app').appendChild(component.$el)
```

- **See also:**
  - Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]
  - Server-Side Rendering [../guide/ssr.html]

vm.$forceUpdate() [#vm-forceUpdate]

- **Usage:**

  Force the Vue instance to re-render. Note it does not affect all child components, only the instance itself and child components with inserted slot content.

vm.$nextTick([callback]) [#vm-nextTick]

- **Arguments:**
  - `{Function} [callback]`

- **Usage:**

  Defer the callback to be executed after the next DOM update cycle. Use it immediately after you've changed some data to wait for the DOM update. This is the same as the global `Vue.nextTick`, except that the callback's `this` context is automatically bound to the instance calling this method.

  > **New in 2.1.0+: returns a Promise if no callback is provided and Promise is supported in the execution environment. Please note that Vue does not come with a Promise polyfill, so if you target browsers that don't support Promises natively (looking at you, IE), you will have to provide a polyfill yourself.**

- **Example:**

JS
```
new Vue({
  // ...
  methods: {
    // ...
    example: function () {
      // modify data
      this.message = 'changed'
      // DOM is not updated yet
      this.$nextTick(function () {
        // DOM is now updated
        // `this` is bound to the current instance
        this.doSomethingElse()
      })
    }
  }
})
```

- **See also:**
  - Vue.nextTick [#Vue-nextTick]
  - Async Update Queue [../guide/reactivity.html#Async-Update-Queue]

vm.$destroy() [#vm-destroy]

- **Usage:**

  Completely destroy a vm. Clean up its connections with other existing vms, unbind all its directives, turn off all event listeners.

  Triggers the `beforeDestroy` and `destroyed` hooks.

  > ! In normal use cases you shouldn't have to call this method yourself. Prefer controlling the lifecycle of child components in a data-driven fashion using `v-if` and `v-for` .

- **See also:** Lifecycle Diagram [../guide/instance.html#Lifecycle-Diagram]

Directives [#Directives]

# v-text [#v-text]

- **Expects:** `string`
- **Details:**

  Updates the element's `textContent` . If you need to update the part of `textContent` , you should use `{{ Mustache }}` interpolations.

- **Example:**

  HTML
  ```
  <span v-text="msg"></span>
  <!-- same as -->
  <span>{{msg}}</span>
  ```

- **See also:** Data Binding Syntax - Interpolations [../guide/syntax.html#Text]

# v-html [#v-html]

- **Expects:** `string`
- **Details:**

  Updates the element's `innerHTML` . **Note that the contents are inserted as plain HTML - they will not be compiled as Vue templates.** If you find yourself trying to compose templates using `v-html` , try to rethink the solution by using components instead.

  > !
  > Dynamically rendering arbitrary HTML on your website can be very dangerous because it can easily lead to XSS attacks [https://en.wikipedia.org/wiki/Cross-site_scripting] . Only use `v-html` on trusted content and **never** on user-provided content.

> In **single-file components [../guide/single-file-components.html]** , `scoped` styles will not apply to content inside `v-html` , because that HTML is not processed by Vue's template compiler. If you want to target `v-html` content with scoped CSS, you can instead use **CSS modules [https://vue-loader.vuejs.org/en/features/css-modules.html]** or an additional, global `<style>` element with a manual scoping strategy such as BEM.

- **Example:**

  HTML
  ```html
  <div v-html="html"></div>
  ```

- **See also: Data Binding Syntax - Interpolations [../guide/syntax.html#Raw-HTML]**

# v-show [#v-show]

- **Expects:** `any`

- **Usage:**

  Toggles the element's `display` CSS property based on the truthy-ness of the expression value.

  This directive triggers transitions when its condition changes.

- **See also: Conditional Rendering - v-show [../guide/conditional.html#v-show]**

# v-if [#v-if]

- **Expects:** `any`

- **Usage:**

  Conditionally render the element based on the truthy-ness of the expression value. The element and its contained directives / components are destroyed and re-constructed during toggles. If the element is a `<template>` element, its content will be extracted as the conditional block.

This directive triggers transitions when its condition changes.

> ! When used together with v-if, v-for has a higher priority than v-if. See the list rendering guide [../guide/list.html#v-for-with-v-if] for details.

- **See also:** Conditional Rendering - v-if [../guide/conditional.html]

# v-else [#v-else]

- **Does not expect expression**
- **Restriction:** previous sibling element must have `v-if` or `v-else-if` .
- **Usage:**

  Denote the “else block” for `v-if` or a `v-if` / `v-else-if` chain.

  HTML
  ```html
  <div v-if="Math.random() > 0.5">
    Now you see me
  </div>
  <div v-else>
    Now you don't
  </div>
  ```

- **See also:** Conditional Rendering - v-else [../guide/conditional.html#v-else]

# v-else-if [#v-else-if]

> **New in 2.1.0+**

- **Expects:** `any`

- **Restriction:** previous sibling element must have `v-if` or `v-else-if` .

- **Usage:**

  Denote the “else if block” for `v-if` . Can be chained.

  HTML
  ```html
  <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>
  ```

- **See also:** Conditional Rendering - v-else-if [../guide/conditional.html#v-else-if]

# v-for [#v-for]

- **Expects:** `Array | Object | number | string | Iterable (since 2.6)`

- **Usage:**

  Render the element or template block multiple times based on the source data. The directive’s value must use the special syntax `alias in expression` to provide an alias for the current element being iterated on:

  HTML
  ```html
  <div v-for="item in items">
    {{ item.text }}
  </div>
  ```

Alternatively, you can also specify an alias for the index (or the key if used on an Object):

```html
<div v-for="(item, index) in items"></div>
<div v-for="(val, key) in object"></div>
<div v-for="(val, name, index) in object"></div>
```

The default behavior of `v-for` will try to patch the elements in-place without moving them. To force it to reorder elements, you need to provide an ordering hint with the `key` special attribute:

```html
<div v-for="item in items" :key="item.id">
  {{ item.text }}
</div>
```

In 2.6+, `v-for` can also work on values that implement the **Iterable Protocol [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#The_iterable_protocol]** , including native `Map` and `Set` . However, it should be noted that Vue 2.x currently does not support reactivity on `Map` and `Set` values, so cannot automatically detect changes.

> ! When used together with v-if, v-for has a higher priority than v-if. See the **list rendering guide [../guide/list.html#v-for-with-v-if]** for details.

The detailed usage for `v-for` is explained in the guide section linked below.

- **See also:**
  - **List Rendering [../guide/list.html]**
  - **key [../guide/list.html#key]**

# v-on [#v-on]

- **Shorthand:** `@`
- **Expects:** `Function | Inline Statement | Object`

- **Argument:** `event`

- **Modifiers:**

  - `.stop` - call `event.stopPropagation()` .
  - `.prevent` - call `event.preventDefault()` .
  - `.capture` - add event listener in capture mode.
  - `.self` - only trigger handler if event was dispatched from this element.
  - `.{keyCode | keyAlias}` - only trigger handler on certain keys.
  - `.native` - listen for a native event on the root element of component.
  - `.once` - trigger handler at most once.
  - `.left` - (2.2.0+) only trigger handler for left button mouse events.
  - `.right` - (2.2.0+) only trigger handler for right button mouse events.
  - `.middle` - (2.2.0+) only trigger handler for middle button mouse events.
  - `.passive` - (2.3.0+) attaches a DOM event with `{ passive: true }` .

- **Usage:**

  Attaches an event listener to the element. The event type is denoted by the argument. The expression can be a method name, an inline statement, or omitted if there are modifiers present.

  When used on a normal element, it listens to **native DOM events** [https://developer.mozilla.org/en-US/docs/Web/Events] only. When used on a custom element component, it listens to **custom events** emitted on that child component.

  When listening to native DOM events, the method receives the native event as the only argument. If using inline statement, the statement has access to the special `$event` property: `v-on:click="handle('ok', $event)"` .

  Starting in 2.4.0+, `v-on` also supports binding to an object of event/listener pairs without an argument. Note when using the object syntax, it does not support any modifiers.

- **Example:**

```html
HTML
<!-- method handler -->
<button v-on:click="doThis"></button>

<!-- dynamic event (2.6.0+) -->
<button v-on:[event]="doThis"></button>

<!-- inline statement -->
```

```html
<button v-on:click="doThat('hello', $event)"></button>

<!-- shorthand -->
<button @click="doThis"></button>

<!-- shorthand dynamic event (2.6.0+) -->
<button @[event]="doThis"></button>

<!-- stop propagation -->
<button @click.stop="doThis"></button>

<!-- prevent default -->
<button @click.prevent="doThis"></button>

<!-- prevent default without expression -->
<form @submit.prevent></form>

<!-- chain modifiers -->
<button @click.stop.prevent="doThis"></button>

<!-- key modifier using keyAlias -->
<input @keyup.enter="onEnter">

<!-- key modifier using keyCode -->
<input @keyup.13="onEnter">

<!-- the click event will be triggered at most once -->
<button v-on:click.once="doThis"></button>

<!-- object syntax (2.4.0+) -->
<button v-on="{ mousedown: doThis, mouseup: doThat }"></button>
```

Listening to custom events on a child component (the handler is called when “my-event” is emitted on the child):

HTML

```html
<my-component @my-event="handleThis"></my-component>

<!-- inline statement -->
```

```
<my-component @my-event="handleThis(123, $event)"></my-component>

<!-- native event on component -->
<my-component @click.native="onClick"></my-component>
```

- **See also:**
  - **Event Handling [../guide/events.html]**
  - **Components - Custom Events [../guide/components.html#Custom-Events]**

# v-bind [#v-bind]

- **Shorthand:** `:`
- **Expects:** `any (with argument) | Object (without argument)`
- **Argument:** `attrOrProp (optional)`
- **Modifiers:**
  - `.prop` - Bind as a DOM property instead of an attribute (**what's the difference? [https://stackoverflow.com/questions/6003819/properties-and-attributes-in-html#answer-6004028]**). If the tag is a component then `.prop` will set the property on the component's `$el` .
  - `.camel` - (2.1.0+) transform the kebab-case attribute name into camelCase.
  - `.sync` - (2.3.0+) a syntax sugar that expands into a `v-on` handler for updating the bound value.
- **Usage:**

  Dynamically bind one or more attributes, or a component prop to an expression.

  When used to bind the `class` or `style` attribute, it supports additional value types such as Array or Objects. See linked guide section below for more details.

  When used for prop binding, the prop must be properly declared in the child component.

  When used without an argument, can be used to bind an object containing attribute name-value pairs. Note in this mode `class` and `style` does not support Array or Objects.

- **Example:**

```html
<!-- bind an attribute -->
<img v-bind:src="imageSrc">

<!-- dynamic attribute name (2.6.0+) -->
<button v-bind:[key]="value"></button>

<!-- shorthand -->
<img :src="imageSrc">

<!-- shorthand dynamic attribute name (2.6.0+) -->
<button :[key]="value"></button>

<!-- with inline string concatenation -->
<img :src="'/path/to/images/' + fileName">

<!-- class binding -->
<div :class="{ red: isRed }"></div>
<div :class="[classA, classB]"></div>
<div :class="[classA, { classB: isB, classC: isC }]">

<!-- style binding -->
<div :style="{ fontSize: size + 'px' }"></div>
<div :style="[styleObjectA, styleObjectB]"></div>

<!-- binding an object of attributes -->
<div v-bind="{ id: someProp, 'other-attr': otherProp }"></div>

<!-- DOM attribute binding with prop modifier -->
<div v-bind:text-content.prop="text"></div>

<!-- prop binding. "prop" must be declared in my-component. -->
<my-component :prop="someThing"></my-component>

<!-- pass down parent props in common with a child component -->
<child-component v-bind="$props"></child-component>
```

```html
<!-- XLink -->
<svg><a :xlink:special="foo"></a></svg>
```

The `.camel` modifier allows camelizing a `v-bind` attribute name when using in-DOM templates, e.g. the SVG `viewBox` attribute:

```html
<svg :view-box.camel="viewBox"></svg>
```

`.camel` is not needed if you are using string templates, or compiling with `vue-loader` / `vueify` .

- **See also:**
  - Class and Style Bindings [../guide/class-and-style.html]
  - Components - Props [../guide/components.html#Props]
  - Components - `.sync` Modifier [../guide/components.html#sync-Modifier]

# v-model [#v-model]

- **Expects:** varies based on value of form inputs element or output of components
- **Limited to:**
  - `<input>`
  - `<select>`
  - `<textarea>`
  - components
- **Modifiers:**
  - `.lazy` [../guide/forms.html#lazy] - listen to `change` events instead of `input`
  - `.number` [../guide/forms.html#number] - cast valid input string to numbers
  - `.trim` [../guide/forms.html#trim] - trim input
- **Usage:**

  Create a two-way binding on a form input element or a component. For detailed usage and other notes, see the Guide section linked below.

- **See also:**

- Form Input Bindings [../guide/forms.html]
- Components - Form Input Components using Custom Events [../guide/components.html#Form-Input-Components-using-Custom-Events]

# v-slot [#v-slot]

- **Shorthand:** `#`
- **Expects:** JavaScript expression that is valid in a function argument position (supports destructuring in supported environments [../guide/components-slots.html#Slot-Props-Destructuring]). Optional - only needed if expecting props to be passed to the slot.
- **Argument:** slot name (optional, defaults to `default`)
- **Limited to:**
  - `<template>`
  - components [../guide/components-slots.html#Abbreviated-Syntax-for-Lone-Default-Slots] (for a lone default slot with props)
- **Usage:**

  Denote named slots or slots that expect to receive props.

- **Example:**

```html
<!-- Named slots -->
<base-layout>
  <template v-slot:header>
    Header content
  </template>

  Default slot content

  <template v-slot:footer>
    Footer content
  </template>
</base-layout>
```

```
<!-- Named slot that receives props -->
<infinite-scroll>
  <template v-slot:item="slotProps">
    <div class="item">
      {{ slotProps.item.text }}
    </div>
  </template>
</infinite-scroll>

<!-- Default slot that receive props, with destructuring -->
<mouse-position v-slot="{ x, y }">
  Mouse position: {{ x }}, {{ y }}
</mouse-position>
```

For more details, see the links below.

- **See also:**
  - Components - Slots [../guide/components-slots.html]
  - RFC-0001 [https://github.com/vuejs/rfcs/blob/master/active-rfcs/0001-new-slot-syntax.md]

# v-pre [#v-pre]

- **Does not expect expression**
- **Usage:**

  Skip compilation for this element and all its children. You can use this for displaying raw mustache tags. Skipping large numbers of nodes with no directives on them can also speed up compilation.

- **Example:**

```html
<span v-pre>{{ this will not be compiled }}</span>
```

# v-cloak [#v-cloak]

- **Does not expect expression**
- **Usage:**

  This directive will remain on the element until the associated Vue instance finishes compilation. Combined with CSS rules such as `[v-cloak] { display: none }`, this directive can be used to hide un-compiled mustache bindings until the Vue instance is ready.

- **Example:**

  CSS
  ```css
  [v-cloak] {
    display: none;
  }
  ```

  HTML
  ```html
  <div v-cloak>
    {{ message }}
  </div>
  ```

  The `<div>` will not be visible until the compilation is done.

# v-once [#v-once]

- **Does not expect expression**
- **Details:**

  Render the element and component **once** only. On subsequent re-renders, the element/component and all its children will be treated as static content and skipped. This can be used to optimize update performance.

  HTML
  ```html
  <!-- single element -->
  <span v-once>This will never change: {{msg}}</span>
  <!-- the element have children -->
  <div v-once>
  ```

```
    <h1>comment</h1>
    <p>{{msg}}</p>
  </div>
  <!-- component -->
  <my-component v-once :comment="msg"></my-component>
  <!-- `v-for` directive -->
  <ul>
    <li v-for="i in list" v-once>{{i}}</li>
  </ul>
```

- **See also:**
  - Data Binding Syntax - interpolations [../guide/syntax.html#Text]
  - Components - Cheap Static Components with `v-once` [../guide/components.html#Cheap-Static-Components-with-v-once]

Special Attributes [#Special-Attributes]

# key [#key]

- **Expects:** `number | string | boolean (since 2.4.2) | symbol (since 2.5.12)`

  The `key` special attribute is primarily used as a hint for Vue's virtual DOM algorithm to identify VNodes when diffing the new list of nodes against the old list. Without keys, Vue uses an algorithm that minimizes element movement and tries to patch/reuse elements of the same type in-place as much as possible. With keys, it will reorder elements based on the order change of keys, and elements with keys that are no longer present will always be removed/destroyed.

  Children of the same common parent must have **unique keys**. Duplicate keys will cause render errors.

  The most common use case is combined with `v-for`:

  HTML
  ```
  <ul>
    <li v-for="item in items" :key="item.id">...</li>
  </ul>
  ```

It can also be used to force replacement of an element/component instead of reusing it. This can be useful when you want to:

- Properly trigger lifecycle hooks of a component
- Trigger transitions

For example:

HTML
```html
<transition>
  <span :key="text">{{ text }}</span>
</transition>
```

When `text` changes, the `<span>` will always be replaced instead of patched, so a transition will be triggered.

# ref [#ref]

- **Expects:** `string`

  `ref` is used to register a reference to an element or a child component. The reference will be registered under the parent component's `$refs` object. If used on a plain DOM element, the reference will be that element; if used on a child component, the reference will be component instance:

  HTML
  ```html
  <!-- vm.$refs.p will be the DOM node -->
  <p ref="p">hello</p>

  <!-- vm.$refs.child will be the child component instance -->
  <child-component ref="child"></child-component>
  ```

  When used on elements/components with `v-for`, the registered reference will be an Array containing DOM nodes or component instances.

  An important note about the ref registration timing: because the refs themselves are created as a result of the render function, you cannot access them on the initial render - they don't exist yet! `$refs` is also non-reactive, therefore you should not attempt to use it in templates for data-binding.

- **See also:** Child Component Refs [../guide/components.html#Child-Component-Refs]

# is [#is]

- **Expects:** `string | Object (component’s options object)`

  Used for dynamic components [../guide/components.html#Dynamic-Components] and to work around limitations of in-DOM templates [../guide/components.html#DOM-Template-Parsing-Caveats] .

  For example:

  ```html
  <!-- component changes when currentView changes -->
  <component v-bind:is="currentView"></component>

  <!-- necessary because `<my-row>` would be invalid inside -->
  <!-- a `<table>` element and so would be hoisted out      -->
  <table>
    <tr is="my-row"></tr>
  </table>
  ```

  For detailed usage, follow the links in the description above.

- **See also:**
  - Dynamic Components [../guide/components.html#Dynamic-Components]
  - DOM Template Parsing Caveats [../guide/components.html#DOM-Template-Parsing-Caveats]

slot deprecated [#slot-deprecated]

Prefer v-slot [#v-slot] in 2.6.0+.

- **Expects:** `string`

Used on content inserted into child components to indicate which named slot the content belongs to.

- **See also:** Named Slots with `slot` [../guide/components.html#Named-Slots-with-slot]

slot-scope deprecated [#slot-scope-deprecated]

Prefer v-slot [#v-slot] in 2.6.0+.

- **Expects:** `function argument expression`
- **Usage:**

  Used to denote an element or component as a scoped slot. The attribute's value should be a valid JavaScript expression that can appear in the argument position of a function signature. This means in supported environments you can also use ES2015 destructuring in the expression. Serves as a replacement for `scope` [#scope-replaced] in 2.5.0+.

  This attribute does not support dynamic binding.

- **See also:** Scoped Slots with `slot-scope` [../guide/components.html#Scoped-Slots-with-slot-scope]

scope removed [#scope-removed]

Replaced by slot-scope [#slot-scope] in 2.5.0+. Prefer v-slot [#v-slot] in 2.6.0+.

Used to denote a `<template>` element as a scoped slot.

- **Usage:**

  Same as `slot-scope` [#slot-scope] except that `scope` can only be used on `<template>` elements.

Built-In Components [#Built-In-Components]

# component [#component]

- **Props:**
  - `is` - string | ComponentDefinition | ComponentConstructor
  - `inline-template` - boolean
- **Usage:**

  A “meta component” for rendering dynamic components. The actual component to render is determined by the `is` prop:

  HTML
  ```html
  <!-- a dynamic component controlled by -->
  <!-- the `componentId` property on the vm -->
  <component :is="componentId"></component>

  <!-- can also render registered component or component passed as prop -
  <component :is="$options.components.child"></component>
  ```

- **See also:** Dynamic Components [../guide/components.html#Dynamic-Components]

# transition [#transition]

- **Props:**
  - `name` - string, Used to automatically generate transition CSS class names. e.g. `name: 'fade'` will auto expand to `.fade-enter` , `.fade-enter-active` , etc. Defaults to `"v"` .
  - `appear` - boolean, Whether to apply transition on initial render. Defaults to `false` .
  - `css` - boolean, Whether to apply CSS transition classes. Defaults to `true` . If set to `false` , will only trigger JavaScript hooks registered via component events.
  - `type` - string, Specifies the type of transition events to wait for to determine transition end timing. Available values are `"transition"` and `"animation"` . By default, it will automatically detect the type that has a longer duration.
  - `mode` - string, Controls the timing sequence of leaving/entering transitions. Available modes are `"out-in"` and `"in-out"` ; defaults to simultaneous.

- `duration` - number | { `enter` : number, `leave` : number }, Specifies the duration of transition. By default, Vue waits for the first `transitionend` or `animationend` event on the root transition element.
- `enter-class` - string
- `leave-class` - string
- `appear-class` - string
- `enter-to-class` - string
- `leave-to-class` - string
- `appear-to-class` - string
- `enter-active-class` - string
- `leave-active-class` - string
- `appear-active-class` - string

- **Events:**

  - `before-enter`
  - `before-leave`
  - `before-appear`
  - `enter`
  - `leave`
  - `appear`
  - `after-enter`
  - `after-leave`
  - `after-appear`
  - `enter-cancelled`
  - `leave-cancelled` (`v-show` only)
  - `appear-cancelled`

- **Usage:**

  `<transition>` serve as transition effects for **single** element/component. The `<transition>` only applies the transition behavior to the wrapped content inside; it doesn't render an extra DOM element, or show up in the inspected component hierarchy.

HTML

```html
<!-- simple element -->
<transition>
  <div v-if="ok">toggled content</div>
</transition>

<!-- dynamic component -->
<transition name="fade" mode="out-in" appear>
  <component :is="view"></component>
```

```
</transition>

<!-- event hooking -->
<div id="transition-demo">
  <transition @after-enter="transitionComplete">
    <div v-show="ok">toggled content</div>
  </transition>
</div>
```

JS

```
new Vue({
  ...
  methods: {
    transitionComplete: function (el) {
      // for passed 'el' that DOM element as the argument, something ..
    }
  }
  ...
}).$mount('#transition-demo')
```

- **See also: Transitions: Entering, Leaving, and Lists [../guide/transitions.html]**

# transition-group [#transition-group]

- **Props:**
  - `tag` - string, defaults to `span` .
  - `move-class` - overwrite CSS class applied during moving transition.
  - exposes the same props as `<transition>` except `mode` .
- **Events:**
  - exposes the same events as `<transition>` .
- **Usage:**

  `<transition-group>` serve as transition effects for **multiple** elements/components. The `<transition-group>` renders a real DOM element. By default it renders a

`<span>` , and you can configure what element it should render via the `tag` attribute.

Note that every child in a `<transition-group>` must be **uniquely keyed** for the animations to work properly.

`<transition-group>` supports moving transitions via CSS transform. When a child's position on screen has changed after an update, it will get applied a moving CSS class (auto generated from the `name` attribute or configured with the `move-class` attribute). If the CSS `transform` property is "transition-able" when the moving class is applied, the element will be smoothly animated to its destination using the **FLIP technique [https://aerotwist.com/blog/flip-your-animations/]** .

HTML

```html
<transition-group tag="ul" name="slide">
  <li v-for="item in items" :key="item.id">
    {{ item.text }}
  </li>
</transition-group>
```

- **See also: Transitions: Entering, Leaving, and Lists [../guide/transitions.html]**

# keep-alive [#keep-alive]

- **Props:**
  - `include` - string or RegExp or Array. Only components with matching names will be cached.
  - `exclude` - string or RegExp or Array. Any component with a matching name will not be cached.
  - `max` - number. The maximum number of component instances to cache.

- **Usage:**

When wrapped around a dynamic component, `<keep-alive>` caches the inactive component instances without destroying them. Similar to `<transition>` , `<keep-alive>` is an abstract component: it doesn't render a DOM element itself, and doesn't show up in the component parent chain.

When a component is toggled inside `<keep-alive>` , its `activated` and `deactivated` lifecycle hooks will be invoked accordingly.

> **In 2.2.0+ and above, `activated` and `deactivated` will fire for all nested components inside a `<keep-alive>` tree.**

Primarily used to preserve component state or avoid re-rendering.

```html
<!-- basic -->
<keep-alive>
  <component :is="view"></component>
</keep-alive>

<!-- multiple conditional children -->
<keep-alive>
  <comp-a v-if="a > 1"></comp-a>
  <comp-b v-else></comp-b>
</keep-alive>

<!-- used together with `<transition>` -->
<transition>
  <keep-alive>
    <component :is="view"></component>
  </keep-alive>
</transition>
```

Note, `<keep-alive>` is designed for the case where it has one direct child component that is being toggled. It does not work if you have `v-for` inside it. When there are multiple conditional children, as above, `<keep-alive>` requires that only one child is rendered at a time.

- **`include` and `exclude`**

> **New in 2.1.0+**

The `include` and `exclude` props allow components to be conditionally cached. Both props can be a comma-delimited string, a RegExp or an Array:

```html
<!-- comma-delimited string -->
<keep-alive include="a,b">
  <component :is="view"></component>
```

```
</keep-alive>

<!-- regex (use `v-bind`) -->
<keep-alive :include="/a|b/">
  <component :is="view"></component>
</keep-alive>

<!-- Array (use `v-bind`) -->
<keep-alive :include="['a', 'b']">
  <component :is="view"></component>
</keep-alive>
```

The match is first checked on the component's own `name` option, then its local registration name (the key in the parent's `components` option) if the `name` option is not available. Anonymous components cannot be matched against.

- `max`

> **New in 2.5.0+**

The maximum number of component instances to cache. Once this number is reached, the cached component instance that was least recently accessed will be destroyed before creating a new instance.

```html
<keep-alive :max="10">
  <component :is="view"></component>
</keep-alive>
```

> ! `<keep-alive>` does not work with functional components because they do not have instances to be cached.

- **See also:** Dynamic Components - keep-alive [../guide/components.html#keep-alive]

# slot [#slot]

- **Props:**
  - `name` - string, Used for named slot.

- **Usage:**

  `<slot>` serve as content distribution outlets in component templates. `<slot>` itself will be replaced.

  For detailed usage, see the guide section linked below.

- **See also: Content Distribution with Slots [../guide/components.html#Content-Distribution-with-Slots]**

VNode Interface [#VNode-Interface]

- Please refer to the VNode class declaration [https://github.com/vuejs/vue/blob/dev/src/core/vdom/vnode.js] .

Server-Side Rendering [#Server-Side-Rendering]

- Please refer to the vue-server-renderer package documentation [https://github.com/vuejs/vue/tree/dev/packages/vue-server-renderer] .