Components

What are Components?

Components are one of the most powerful features of Vue. They help you extend basic HTML elements to encapsulate reusable code. At a high level, components are custom elements that Vue's compiler attaches behavior to. In some cases, they may also appear as a native HTML element extended with the special `is` attribute.

Using Components

Registration

We've learned in the previous sections that we can create a new Vue instance with:

```js
new Vue({
  el: '#some-element',
  // options
})
```

To register a global component, you can use `Vue.component(tagName, options)`. For example:

```js
Vue.component('my-component', {
  // options
})
```

Once registered, a component can be used in an instance's template as a custom element, `<my-component></my-component>`. Make sure the component is registered **before** you instantiate the root Vue instance. Here's the full example:

```html
<div id="example">
  <my-component></my-component>
</div>
```

```js
// register
Vue.component('my-component', {
  template: '<div>A custom component!</div>'
})

// create a root instance
new Vue({
  el: '#example'
})
```

Which will render:

```html
<div id="example">
  <div>A custom component!</div>
</div>
```

Local Registration

You don’t have to register every component globally. You can make a component available only in the scope of another instance/component by registering it with the `components` instance option:

```js
var Child = {
  template: '<div>A custom component!</div>'
}

new Vue({
  // ...
  components: {
    // <my-component> will only be available in parent's template
    'my-component': Child
  }
})
```

The same encapsulation applies for other registerable Vue features, such as directives.

DOM Template Parsing Caveats

When using the DOM as your template (e.g. using the `el` option to mount an element with existing content), you will be subject to some restrictions that are inherent to how HTML works, because Vue can only retrieve the template content **after** the browser has parsed and normalized it. Most notably, some elements such as `<ul>` , `<ol>` , `<table>` and `<select>` have restrictions on what elements can appear inside them, and some elements such as `<option>` can only appear inside certain other elements.

This will lead to issues when using custom components with elements that have such restrictions, for example:

```html
<table>
  <my-row>...</my-row>
</table>
```

The custom component `<my-row>` will be hoisted out as invalid content, thus causing errors in the eventual rendered output. A workaround is to use the `is` special attribute:

```html
<table>
  <tr is="my-row"></tr>
</table>
```

It should be noted that these limitations do not apply if you are using string templates from one of the following sources:

- `<script type="text/x-template">`
- JavaScript inline template strings
- `.vue` components

Therefore, prefer using string templates whenever possible.

`data` Must Be a Function

Most of the options that can be passed into the Vue constructor can be used in a component, with one special case: `data` must be a function. In fact, if you try this:

```js
Vue.component('my-component', {
  template: '<span>{{ message }}</span>',
  data: {
    message: 'hello'
  }
})
```

Then Vue will halt and emit warnings in the console, telling you that `data` must be a function for component instances. It's good to understand why the rules exist though, so let's cheat.

```html
<div id="example-2">
  <simple-counter></simple-counter>
  <simple-counter></simple-counter>
  <simple-counter></simple-counter>
</div>
```

```js
var data = { counter: 0 }

Vue.component('simple-counter', {
  template: '<button v-on:click="counter += 1">{{ counter }}</button>',
  // data is technically a function, so Vue won't
  // complain, but we return the same object
  // reference for each component instance
  data: function () {
    return data
  }
})

new Vue({
  el: '#example-2'
})
```

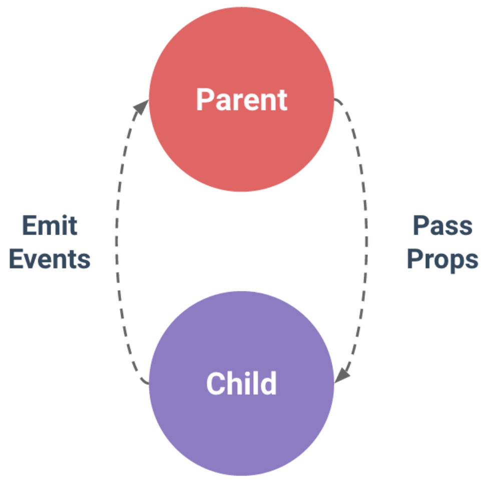
Since all three component instances share the same `data` object, incrementing one counter increments them all! Ouch. Let's fix this by instead returning a fresh data object:

```js
data: function () {
  return {
    counter: 0
  }
}
```

Composing Components

Components are meant to be used together, most commonly in parent-child relationships: component A may use component B in its own template. They inevitably need to communicate to one another: the parent may need to pass data down to the child, and the child may need to inform the parent of something that happened in the child. However, it is also very important to keep the parent and the child as decoupled as possible via a clearly-defined interface. This ensures each component's code can be written and reasoned about in relative isolation, thus making them more maintainable and potentially easier to reuse.

In Vue, the parent-child component relationship can be summarized as **props down, events up**. The parent passes data down to the child via **props**, and the child sends messages to the parent via **events**. Let's see how they work next.



Props

Passing Data with Props

Every component instance has its own **isolated scope**. This means you cannot (and should not) directly reference parent data in a child component's template. Data can be passed down to child components using **props**.

A prop is a custom attribute for passing information from parent components. A child component needs to explicitly declare the props it expects to receive using the `props` option:

```js
Vue.component('child', {
  // declare the props
  props: ['message'],
  // like data, the prop can be used inside templates and
  // is also made available in the vm as this.message
  template: '<span>{{ message }}</span>'
})
```

Then we can pass a plain string to it like so:

```html
<child message="hello!"></child>
```

camelCase vs. kebab-case

HTML attributes are case-insensitive, so when using non-string templates, camelCased prop names need to use their kebab-case (hyphen-delimited) equivalents:

```js
Vue.component('child', {
  // camelCase in JavaScript
  props: ['myMessage'],
  template: '<span>{{ myMessage }}</span>'
})
```

```html
<!-- kebab-case in HTML -->
<child my-message="hello!"></child>
```

Again, if you’re using string templates, then this limitation does not apply.

Dynamic Props

Similar to binding a normal attribute to an expression, we can also use `v-bind` for dynamically binding props to data on the parent. Whenever the data is updated in the parent, it will also flow down to the child:

```html
<div>
  <input v-model="parentMsg">
  <br>
  <child v-bind:my-message="parentMsg"></child>
</div>
```

You can also use the shorthand syntax for `v-bind` :

```html
<child :my-message="parentMsg"></child>
```

If you want to pass all the properties in an object as props, you can use `v-bind` without an argument (`v-bind` instead of `v-bind:prop-name`). For example, given a `todo` object:

```js
todo: {
  text: 'Learn Vue',
  isComplete: false
}
```

Then:

```html
<todo-item v-bind="todo"></todo-item>
```

Will be equivalent to:

```html
<todo-item
  v-bind:text="todo.text"
  v-bind:is-complete="todo.isComplete"
></todo-item>
```

Literal vs. Dynamic

A common mistake beginners tend to make is attempting to pass down a number using the literal syntax:

```html
<!-- this passes down a plain string "1" -->
```

```html
<comp some-prop="1"></comp>
```

However, since this is a literal prop, its value is passed down as a plain string `"1"` instead of an actual number. If we want to pass down an actual JavaScript number, we need to use `v-bind` so that its value is evaluated as a JavaScript expression:

```html
<!-- this passes down an actual number -->
<comp v-bind:some-prop="1"></comp>
```

One-Way Data Flow

All props form a **one-way-down** binding between the child property and the parent one: when the parent property updates, it will flow down to the child, but not the other way around. This prevents child components from accidentally mutating the parent's state, which can make your app's data flow harder to understand.

In addition, every time the parent component is updated, all props in the child component will be refreshed with the latest value. This means you should **not** attempt to mutate a prop inside a child component. If you do, Vue will warn you in the console.

There are usually two cases where it's tempting to mutate a prop:

1. The prop is used to pass in an initial value; the child component wants to use it as a local data property afterwards.

2. The prop is passed in as a raw value that needs to be transformed.

The proper answer to these use cases are:

1. Define a local data property that uses the prop's initial value as its initial value:

   ```js
   props: ['initialCounter'],
   data: function () {
     return { counter: this.initialCounter }
   }
   ```

2. Define a computed property that is computed from the prop's value:

   ```js
   props: ['size'],
   computed: {
     normalizedSize: function () {
       return this.size.trim().toLowerCase()
     }
   }
   ```

Prop Validation

It is possible for a component to specify requirements for the props it is receiving. If a requirement is not met, Vue will emit warnings. This is especially useful when you are authoring a component that is intended to be used by others.

Instead of defining the props as an array of strings, you can use an object with validation requirements:

```js
Vue.component('example', {
  props: {
    // basic type check (`null` means accept any type)
    propA: Number,
    // multiple possible types
    propB: [String, Number],
    // a required string
    propC: {
      type: String,
      required: true
    },
    // a number with default value
    propD: {
      type: Number,
      default: 100
    },
    // object/array defaults should be returned from a
    // factory function
    propE: {
      type: Object,
      default: function () {
        return { message: 'hello' }
      }
    },
    // custom validator function
    propF: {
      validator: function (value) {
        return value > 10
      }
    }
  }
})
```

The `type` can be one of the following native constructors:

- String
- Number
- Boolean
- Function
- Object
- Array
- Symbol

In addition, `type` can also be a custom constructor function and the assertion will be made with an `instanceof` check.

When prop validation fails, Vue will produce a console warning (if using the development build). Note that props are validated **before** a component instance is created, so within `default` or `validator` functions, instance properties such as from `data`, `computed`, or `methods` will not be available.

Non-Prop Attributes

A non-prop attribute is an attribute that is passed to a component, but does not have a corresponding prop defined.

While explicitly defined props are preferred for passing information to a child component, authors of component libraries can't always foresee the contexts in which their components might be used. That's why components can accept arbitrary attributes, which are added to the component's root element.

For example, imagine we're using a 3rd-party `bs-date-input` component with a Bootstrap plugin that requires a `data-3d-date-picker` attribute on the `input`. We can add this attribute to our component instance:

```html
<bs-date-input data-3d-date-picker="true"></bs-date-input>
```

And the `data-3d-date-picker="true"` attribute will automatically be added to the root element of `bs-date-input`.

Replacing/Merging with Existing Attributes

Imagine this is the template for `bs-date-input`:

```html
<input type="date" class="form-control">
```

To specify a theme for our date picker plugin, we might need to add a specific class, like this:

```html
<bs-date-input
  data-3d-date-picker="true"
  class="date-picker-theme-dark"
></bs-date-input>
```

In this case, two different values for `class` are defined:

- `form-control`, which is set by the component in its template
- `date-picker-theme-dark`, which is passed to the component by its parent

For most attributes, the value provided to the component will replace the value set by the component. So for example, passing `type="large"` will replace `type="date"` and probably break it! Fortunately, the `class` and `style` attributes are a little smarter, so both values are merged, making the final value: `form-control date-picker-theme-dark`.

Custom Events

We have learned that the parent can pass data down to the child using props, but how do we communicate back to the parent when something happens? This is where Vue's custom event system comes in.

Using `v-on` with Custom Events

Every Vue instance implements an **events interface**, which means it can:

- Listen to an event using `$on(eventName)`
- Trigger an event using `$emit(eventName)`

> ! Note that Vue's event system is different from the browser's **EventTarget API**. Though they work similarly, `$on` and `$emit` are **not** aliases for `addEventListener` and `dispatchEvent` .

In addition, a parent component can listen to the events emitted from a child component using `v-on` directly in the template where the child component is used.

> ! You cannot use `$on` to listen to events emitted by children. You must use `v-on` directly in the template, as in the example below.

Here's an example:

```html
<div id="counter-event-example">
  <p>{{ total }}</p>
  <button-counter v-on:increment="incrementTotal"></button-counter>
  <button-counter v-on:increment="incrementTotal"></button-counter>
</div>
```

```js
Vue.component('button-counter', {
  template: '<button v-on:click="incrementCounter">{{ counter }}</button>',
  data: function () {
    return {
      counter: 0
    }
  },
  methods: {
    incrementCounter: function () {
      this.counter += 1
      this.$emit('increment')
    }
  },
})

new Vue({
  el: '#counter-event-example',
  data: {
    total: 0
  },
  methods: {
    incrementTotal: function () {
      this.total += 1
    }
  }
})
```

In this example, it's important to note that the child component is still completely decoupled from what happens outside of it. All it does is report information about its own activity, just in case a parent component might care.

Binding Native Events to Components

There may be times when you want to listen for a native event on the root element of a component. In these cases, you can use the `.native` modifier for `v-on`. For example:

```html
<my-component v-on:click.native="doTheThing"></my-component>
```

`.sync` Modifier

2.3.0+

In some cases we may need "two-way binding" for a prop - in fact, in Vue 1.x this is exactly what the `.sync` modifier provided. When a child component mutates a prop that has `.sync`, the value change will be reflected in the parent. This is convenient, however it leads to maintenance issues in the long run because it breaks the one-way data flow assumption: the code that mutates child props are implicitly affecting parent state.

This is why we removed the `.sync` modifier when 2.0 was released. However, we've found that there are indeed cases where it could be useful, especially when shipping reusable components. What we need to change is **making the code in the child that affects parent state more consistent and explicit.**

In 2.3.0+ we re-introduced the `.sync` modifier for props, but this time it is only syntax sugar that automatically expands into an additional `v-on` listener:

The following

```html
<comp :foo.sync="bar"></comp>
```

is expanded into:

```html
<comp :foo="bar" @update:foo="val => bar = val"></comp>
```

`

For the child component to update `foo` ‘s value, it needs to explicitly emit an event instead of mutating the prop:

```js
this.$emit('update:foo', newValue)
```

Form Input Components using Custom Events

Custom events can also be used to create custom inputs that work with `v-model`. Remember:

```html
<input v-model="something">
```

is syntactic sugar for:

```html
<input
  v-bind:value="something"
  v-on:input="something = $event.target.value">
```

When used with a component, it instead simplifies to:

```html
<custom-input
  :value="something"
  @input="value => { something = value }">
</custom-input>
```

So for a component to work with `v-model`, it should (these can be configured in 2.2.0+):

- accept a `value` prop
- emit an `input` event with the new value

Let’s see it in action with a simple currency input:

```html
<currency-input v-model="price"></currency-input>
```

```js
Vue.component('currency-input', {
  template: '\
    <span>\
      $\
      <input\
        ref="input"\
        v-bind:value="value"\
        v-on:input="updateValue($event.target.value)">\
    </span>\
  ',
  props: ['value'],
  methods: {
    // Instead of updating the value directly, this
    // method is used to format and place constraints
    // on the input's value
    updateValue: function (value) {
      var formattedValue = value
        // Remove whitespace on either side
        .trim()
        // Shorten to 2 decimal places
        .slice(
          0,
          value.indexOf('.') === -1
            ? value.length
            : value.indexOf('.') + 3
        )
      // If the value was not already normalized,
      // manually override it to conform
      if (formattedValue !== value) {
        this.$refs.input.value = formattedValue
      }
      // Emit the number value through the input event
      this.$emit('input', Number(formattedValue))
    }
  }
})
```

Customizing Component `v-model`

> **New in 2.2.0+**

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:

```js
Vue.component('my-checkbox', {
  model: {
    prop: 'checked',
    event: 'change'
  },
  props: {
    checked: Boolean,
    // this allows using the `value` prop for a different purpose
    value: String
  },
  // ...
})
```

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

The above will be equivalent to:

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

> ! Note that you still have to declare the `checked` prop explicitly.

Non Parent-Child Communication

Sometimes two components may need to communicate with one-another but they are not parent/child to each other. In simple scenarios, you can use an empty Vue instance as a central event bus:

```js
var bus = new Vue()
```

```js
// in component A's method
bus.$emit('id-selected', 1)
```

```js
// in component B's created hook
bus.$on('id-selected', function (id) {
  // ...
})
```

In more complex cases, you should consider employing a dedicated **state-management pattern**.

Content Distribution with Slots

When using components, it is often desired to compose them like this:

```html
<app>
  <app-header></app-header>
  <app-footer></app-footer>
</app>
```

There are two things to note here:

1. The `<app>` component does not know what content it will receive. It is decided by the component using `<app>`.

2. The `<app>` component very likely has its own template.

To make the composition work, we need a way to interweave the parent "content" and the component's own template. This is a process called **content distribution** (or "transclusion" if you are familiar with Angular). Vue.js implements a content distribution API that is modeled after the current **Web Components spec draft**, using the special `<slot>` element to serve as distribution outlets for the original content.

Compilation Scope

Before we dig into the API, let's first clarify which scope the contents are compiled in. Imagine a template like this:

```html
<child-component>
  {{ message }}
</child-component>
```

Should the `message` be bound to the parent's data or the child data? The answer is the parent. A simple rule of thumb for component scope is:

> **Everything in the parent template is compiled in parent scope; everything in the child template is compiled in child scope.**

A common mistake is trying to bind a directive to a child property/method in the parent template:

```html
<!-- does NOT work -->
<child-component v-show="someChildProperty"></child-component>
```

Assuming `someChildProperty` is a property on the child component, the example above would not work. The parent's template is not aware of the state of a child component.

If you need to bind child-scope directives on a component root node, you should do so in the child component's own template:

```js
Vue.component('child-component', {
  // this does work, because we are in the right scope
  template: '<div v-show="someChildProperty">Child</div>',
  data: function () {
    return {
      someChildProperty: true
    }
  }
})
```

Similarly, distributed content will be compiled in the parent scope.

Single Slot

Parent content will be **discarded** unless the child component template contains at least one `<slot>` outlet. When there is only one slot with no attributes, the entire content fragment will be inserted at its position in the DOM, replacing the slot itself.

Anything originally inside the `<slot>` tags is considered **fallback content**. Fallback content is compiled in the child scope and will only be displayed if the hosting element is empty and has no content to be inserted.

Suppose we have a component called `my-component` with the following template:

```html
<div>
  <h2>I'm the child title</h2>
  <slot>
    This will only be displayed if there is no content
    to be distributed.
  </slot>
</div>
```

And a parent that uses the component:

```html
<div>
  <h1>I'm the parent title</h1>
  <my-component>
    <p>This is some original content</p>
    <p>This is some more original content</p>
  </my-component>
</div>
```

The rendered result will be:

```html
<div>
  <h1>I'm the parent title</h1>
  <div>
    <h2>I'm the child title</h2>
    <p>This is some original content</p>
    <p>This is some more original content</p>
  </div>
</div>
```

Named Slots

`<slot>` elements have a special attribute, `name`, which can be used to further customize how content should be distributed. You can have multiple slots with different names. A named slot will match any element that has a corresponding `slot` attribute in the content fragment.

There can still be one unnamed slot, which is the **default slot** that serves as a catch-all outlet for any unmatched content. If there is no default slot, unmatched content will be discarded.

For example, suppose we have an `app-layout` component with the following template:

```html
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

Parent markup:

```html
<app-layout>
  <h1 slot="header">Here might be a page title</h1>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <p slot="footer">Here's some contact info</p>
```

```
</app-layout>
```

The rendered result will be:

```html
<div class="container">
  <header>
    <h1>Here might be a page title</h1>
  </header>
  <main>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </main>
  <footer>
    <p>Here's some contact info</p>
  </footer>
</div>
```

The content distribution API is a very useful mechanism when designing components that are meant to be composed together.

Scoped Slots

New in 2.1.0+

A scoped slot is a special type of slot that functions as a reusable template (that can be passed data to) instead of already-rendered-elements.

In a child component, pass data into a slot as if you are passing props to a component:

```html
<div class="child">
  <slot text="hello from child"></slot>
</div>
```

In the parent, a `<template>` element with a special attribute `scope` must exist, indicating that it is a template for a scoped slot. The value of `scope` is the name of a temporary variable that holds the props object passed from the child:

```html
<div class="parent">
  <child>
    <template scope="props">
      <span>hello from parent</span>
      <span>{{ props.text }}</span>
    </template>
  </child>
</div>
```

If we render the above, the output will be:

```html
<div class="parent">
  <div class="child">
```

```
    <span>hello from parent</span>
    <span>hello from child</span>
  </div>
</div>
```

A more typical use case for scoped slots would be a list component that allows the component consumer to customize how each item in the list should be rendered:

```html
<my-awesome-list :items="items">
  <!-- scoped slot can be named too -->
  <template slot="item" scope="props">
    <li class="my-fancy-item">{{ props.text }}</li>
  </template>
</my-awesome-list>
```

And the template for the list component:

```html
<ul>
  <slot name="item"
    v-for="item in items"
    :text="item.text">
    <!-- fallback content here -->
  </slot>
</ul>
```

Dynamic Components

You can use the same mount point and dynamically switch between multiple components using the reserved `<component>` element and dynamically bind to its `is` attribute:

```js
var vm = new Vue({
  el: '#example',
  data: {
    currentView: 'home'
  },
  components: {
    home: { /* ... */ },
    posts: { /* ... */ },
    archive: { /* ... */ }
  }
})
```

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

If you prefer, you can also bind directly to component objects:

```js
var Home = {
  template: '<p>Welcome home!</p>'
}

var vm = new Vue({
  el: '#example',
  data: {
    currentView: Home
  }
})
```

`keep-alive`

If you want to keep the switched-out components in memory so that you can preserve their state or avoid re-rendering, you can wrap a dynamic component in a `<keep-alive>` element:

```HTML
<keep-alive>
  <component :is="currentView">
    <!-- inactive components will be cached! -->
  </component>
</keep-alive>
```

Check out more details on `<keep-alive>` in the **API reference**.

Misc

Authoring Reusable Components

When authoring components, it's good to keep in mind whether you intend to reuse it somewhere else later. It's OK for one-off components to be tightly coupled, but reusable components should define a clean public interface and make no assumptions about the context it's used in.

The API for a Vue component comes in three parts - props, events, and slots:

- **Props** allow the external environment to pass data into the component
- **Events** allow the component to trigger side effects in the external environment
- **Slots** allow the external environment to compose the component with extra content.

With the dedicated shorthand syntaxes for `v-bind` and `v-on`, the intents can be clearly and succinctly conveyed in the template:

```HTML
<my-component
  :foo="baz"
  :bar="qux"
  @event-a="doThis"
  @event-b="doThat"
>
  <img slot="icon" src="...">
  <p slot="main-text">Hello!</p>
</my-component>
```

Child Component Refs

Despite the existence of props and events, sometimes you might still need to directly access a child component in JavaScript. To achieve this you have to assign a reference ID to the child component using `ref` . For example:

```html
<div id="parent">
  <user-profile ref="profile"></user-profile>
</div>
```

```js
var parent = new Vue({ el: '#parent' })
// access child component instance
var child = parent.$refs.profile
```

When `ref` is used together with `v-for` , the ref you get will be an array containing the child components mirroring the data source.

> ! `$refs` are only populated after the component has been rendered, and it is not reactive. It is only meant as an escape hatch for direct child manipulation - you should avoid using `$refs` in templates or computed properties.

Async Components

In large applications, we may need to divide the app into smaller chunks and only load a component from the server when it's actually needed. To make that easier, Vue allows you to define your component as a factory function that asynchronously resolves your component definition. Vue will only trigger the factory function when the component actually needs to be rendered and will cache the result for future re-renders. For example:

```js
Vue.component('async-example', function (resolve, reject) {
  setTimeout(function () {
    // Pass the component definition to the resolve callback
    resolve({
      template: '<div>I am async!</div>'
    })
  }, 1000)
})
```

The factory function receives a `resolve` callback, which should be called when you have retrieved your component definition from the server. You can also call `reject(reason)` to indicate the load has failed. The `setTimeout` here is for demonstration; how to retrieve the component is up to you. One recommended approach is to use async components together with **Webpack's code-splitting feature**:

```js
Vue.component('async-webpack-example', function (resolve) {
  // This special require syntax will instruct Webpack to
  // automatically split your built code into bundles which
  // are loaded over Ajax requests.
  require(['./my-async-component'], resolve)
})
```

You can also return a `Promise` in the factory function, so with Webpack 2 + ES2015 syntax you can do:

```js
Vue.component(
  'async-webpack-example',
  () => import('./my-async-component')
)
```

When using **local registration**, you can also directly provide a function that returns a `Promise` :

```js
new Vue({
  // ...
  components: {
    'my-component': () => import('./my-async-component')
  }
})
```

Advanced Async Components

New in 2.3.0+

Starting in 2.3.0+ the async component factory can also return an object of the following format:

```js
const AsyncComp = () => ({
  // The component to load. Should be a Promise
  component: import('./MyComp.vue'),
  // A component to use while the async component is loading
  loading: LoadingComp,
  // A component to use if the load fails
  error: ErrorComp,
  // Delay before showing the loading component. Default: 200ms.
  delay: 200,
  // The error component will be displayed if a timeout is
  // provided and exceeded. Default: Infinity.
  timeout: 3000
})
```

Note that when used as a route component in `vue-router` , these properties will be ignored because async components are resolved upfront before the route navigation happens. You also need to use `vue-router` 2.4.0+ if you wish to use the above syntax for route components.

Component Naming Conventions

When registering components (or props), you can use kebab-case, camelCase, or PascalCase.

```js
// in a component definition
components: {
  // register using kebab-case
  'kebab-cased-component': { /* ... */ },
  // register using camelCase
  'camelCasedComponent': { /* ... */ },
  // register using PascalCase
  'PascalCasedComponent': { /* ... */ }
}
```

Within HTML templates though, you have to use the kebab-case equivalents:

```html
<!-- always use kebab-case in HTML templates -->
<kebab-cased-component></kebab-cased-component>
<camel-cased-component></camel-cased-component>
<pascal-cased-component></pascal-cased-component>
```

When using *string* templates however, we're not bound by HTML's case-insensitive restrictions. That means even in the template, you can reference your components using:

- kebab-case
- camelCase or kebab-case if the component has been defined using camelCase
- kebab-case, camelCase or PascalCase if the component has been defined using PascalCase

```js
components: {
  'kebab-cased-component': { /* ... */ },
  camelCasedComponent: { /* ... */ },
  PascalCasedComponent: { /* ... */ }
}
```

```html
<kebab-cased-component></kebab-cased-component>

<camel-cased-component></camel-cased-component>
<camelCasedComponent></camelCasedComponent>

<pascal-cased-component></pascal-cased-component>
<pascalCasedComponent></pascalCasedComponent>
<PascalCasedComponent></PascalCasedComponent>
```

This means that the PascalCase is the most universal *declaration convention* and kebab-case is the most universal *usage convention*.

If your component isn't passed content via `slot` elements, you can even make it self-closing with a `/` after the name:

```html
<my-component/>
```

Again, this *only* works within string templates, as self-closing custom elements are not valid HTML and your browser's native parser will not understand them.

Recursive Components

Components can recursively invoke themselves in their own template. However, they can only do so with the `name` option:

```js
name: 'unique-name-of-my-component'
```

When you register a component globally using `Vue.component`, the global ID is automatically set as the component's `name` option.

```js
Vue.component('unique-name-of-my-component', {
  // ...
})
```

If you're not careful, recursive components can also lead to infinite loops:

```js
name: 'stack-overflow',
template: '<div><stack-overflow></stack-overflow></div>'
```

A component like the above will result in a "max stack size exceeded" error, so make sure recursive invocation is conditional (i.e. uses a `v-if` that will eventually be `false`).

Circular References Between Components

Let's say you're building a file directory tree, like in Finder or File Explorer. You might have a `tree-folder` component with this template:

```html
<p>
  <span>{{ folder.name }}</span>
  <tree-folder-contents :children="folder.children"/>
</p>
```

Then a `tree-folder-contents` component with this template:

```html
<ul>
  <li v-for="child in children">
    <tree-folder v-if="child.children" :folder="child"/>
    <span v-else>{{ child.name }}</span>
  </li>
</ul>
```

When you look closely, you'll see that these components will actually be each other's descendent *and* ancestor in the render tree - a paradox! When registering components globally with `Vue.component`, this paradox is resolved for you automatically. If that's you, you can stop reading here.

However, if you're requiring/importing components using a **module system**, e.g. via Webpack or Browserify, you'll get an error:

```
Failed to mount component: template or render function not defined.
```

To explain what's happening, let's call our components A and B. The module system sees that it needs A, but first A needs B, but B needs A, but A needs B, etc, etc. It's stuck in a loop, not knowing how to fully resolve either component without first resolving the other. To fix this, we need to give the module system a point at which it can say, "A needs B *eventually*, but there's no need to resolve B first."

In our case, let's make that point the `tree-folder` component. We know the child that creates the paradox is the `tree-folder-contents` component, so we'll wait until the `beforeCreate` lifecycle hook to register it:

```js
beforeCreate: function () {
  this.$options.components.TreeFolderContents = require('./tree-folder-contents.vue')
}
```

Problem solved!

Inline Templates

When the `inline-template` special attribute is present on a child component, the component will use its inner content as its template, rather than treating it as distributed content. This allows more flexible template-authoring.

```html
<my-component inline-template>
  <div>
    <p>These are compiled as the component's own template.</p>
    <p>Not parent's transclusion content.</p>
  </div>
</my-component>
```

However, `inline-template` makes the scope of your templates harder to reason about. As a best practice, prefer defining templates inside the component using the `template` option or in a `template` element in a `.vue` file.

X-Templates

Another way to define templates is inside of a script element with the type `text/x-template`, then referencing the template by an id. For example:

```html
<script type="text/x-template" id="hello-world-template">
  <p>Hello hello hello</p>
</script>
```

```js
Vue.component('hello-world', {
  template: '#hello-world-template'
})
```

These can be useful for demos with large templates or in extremely small applications, but should otherwise be avoided, because they separate templates from the rest of the component definition.

Cheap Static Components with `v-once`

Rendering plain HTML elements is very fast in Vue, but sometimes you might have a component that contains **a lot** of static content. In these cases, you can ensure that it's only evaluated once and then cached by adding the `v-once` directive to the root element, like this:

```js
Vue.component('terms-of-service', {
  template: '\
    <div v-once>\
      <h1>Terms of Service</h1>\
      ... a lot of static content ...\
    </div>\
  '
})
```