Intro

`v-bind`

```
<div :title="message"></div>
```

- Bind this element's `title` with `message` in the Vue instance
- `:` shorthand

`v-if`, `v-else`

```
<div id="root" v-if="seen">Hello</div>
```

- If `seen` is false in root, this message will disappear

`v-for`

```
<ol>
<li v-for="todo in todos">{{ todo.text }}</li>
</ol>
```

- Display list using array data

Instances

```
var data = { a: 1 }
var vm = new Vue({
  el: '#example',
  data: data
})
```

- Upon Vue instance creation, all properties in the `data` object are added to Vue's *reactivity system*
  - properties are only reactive if they are created before the instance
- Must set initial values for properties
- `Object.freeze()` prevents mutation

```
vm.$data === data // => true
vm.$el === document.getElementById('example') // => true
```

- `$` prefix are instance properties

Templates

Data binding with text interpolation

- `{{ }}` mustache syntax

```
<div>Message: {{ msg }}</div>
<div :id="dynamicId"></div>
```

```
{{ ok ? 'yes' : 'no' }}
```

- JS support inside all data bindings - single expressions

Dynamic Arguments

```
<a :[attribute]="url">
```

- `attribute` is dynamically evaluated as a JS expression
  - ex. Vue instance has property `attribute: 'href'`, equivalent is `:href="url"`

Constraints

- Spaces, quotes, invalid
- All attribute names are converted to lowercase

Class/Style binding

```
<div :class="{ active: isActive }"></div>
```

- Presence of `active` class determined by truthiness of `isActive`

```
<div
    class="static"
    :class="{ active: isActive, 'coolness': epic }"
></div>
```

- Multiple Classes

```
data: {
  isActive: false,
  epic: true
}
```

This data renders:

```
<div class="static coolness"></div>
```

Arrays

```
<div :class="[activeClass, coolClass]"></div>
```

```
data: {
```

```
  activeClass: 'active',
  coolClass: 'epic'
}
```

```
<div class="active epic"></div>
```

Conditional Rendering

```
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

- Multiple elements with `<template>`

`key`

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username" key="username-input">
</template>
<template v-else>
  <label>Email</label>
  <input placeholder="Enter your email address" key="email-input">
</template>
```

- `key` value separates elements

`v-show`

```
<h1 v-show="ok">Hello!</h1>
```

- Toggles element's `display` css property

`v-if` and `v-show`

- `v-show` if something is toggled often
- `v-if` if condition is unlikely to change

List Rendering

- `v-for` supports an optional 2nd argument: index of current item

`v-for` with Objects

```
<ul id="v-for-obj">
  <li v-for="value in object">
    {{ value }}
  </li>
</ul>
```

- Iterate through object properties
  - Note `in` instead of `of` for object

```
<div v-for="(value, name) in object">
  {{ name }}: {{ value }}
</div>
```

```
new Vue({
  el: '#v-for-obj',
  data: {
    object: {
      title: 'cool name',
      author: 'me'
    }
  }
```

```
})
```

```
title: cool name
author: me
```

```
<div v-for="(value, name, index) in object">
  {{ index }}* {{ name }}: {{ value }}
</div>
```

```
0* title: cool name
1* author: me
```

- Optional second, third argument for property name (key) and index

Track node identity

```
<div v-for="item in items" v-bind:key="item.id">
  <!-- content -->

</div>
```

- Provide unique `key` attribute for each item

Displaying filtered/sorted arrays without mutation

```
<li v-for="n in evenNumbers">{{ n }}</li>
```

```
data: {
  numbers: [ 1, 2, 3, 4, 5 ]
},
computed: {
  evenNumbers: function () {
    return this.numbers.filter(function (number) {
```

```
      return number % 2 === 0
    })
  }
}
```

`v-for` with range bound

```
<div>
  <span v-for="n in 10">{{ n }} </span>
</div>
```

```
1 2 3 4 5 6 7 8 9 10
```

`v-for` with components

- `key` and props required

```
<my-component
  v-for="(item, index) in items"
  v-bind:item="item"
  v-bind:index="index"
  v-bind:key="item.id"
></my-component>
```

Event Handling

`v-on`

- Listen to DOM and run JS method on activation
- `@` shorthand

Event Modifers

- `.stop` - prevents event bubbling
- `.prevent` - prevents default behaviour (ex. urls)
- `.capture` - uses capture mode (opposite of bubbling)
- `.self` - only trigger if event target is itself
- `.once` - runs function at most once

Key Modifiers

- Listen for keyboard events

```
<!-- only call `vm.submit()` when the `key` is `Enter` -->
<input v-on:keyup.enter="submit">
```

Form Input Bindings

`v-model`

- 2 way data bindings for form input, text area, and select elements
  - Treats Vue form instance data as truth sources

Text

`value` property and `input` event

Checkboxes/Radio

`checked` property and `change` event

Select

`value` prop and `change` event

Modifiers

- `.lazy` - syncs after `change` event
- `.number` - typecast input as a Number
- `.trim` - trim whitespace

Components

- Reusable Vue instances
- Every HTML component element is a new instance

Component `data` must be a function

- Each instance has an independent copy of the data object (scope)

Props

- Pass data into a component (like function arguments)

```
Vue.component('blog-post', {
  props: ['title'],
  template: '<h3>{{ title }}</h3>'
})
```

```
<blog-post title="big blog 1"></blog-post>
<blog-post title="big blog 2"></blog-post>
```

Important:

1. Props are passed **down** the component tree to children
2. Props are read-only

- In templates, access props using `{`
- Everywhere else in the Vue component use `this.name`

Single Root Element

- When building a component ex. `<blog-post>`, the template will eventually have more than 1 element

```
<h3>{{ title }}</h3>
<div v-html="content"></div>
```

- Error, more than 1 root element

```
<div class="blog-post">
  <h3>{{ title }}</h3>
  <div v-html="content"></div>
</div>
```

- Use a wrapper

`$emit`

- Say we want to create a custom `v-on` event

```
  <modal v-show="showModal" @close="showModal = false">

  <button @click="$emit('close')"></button>
```

- In this example, we have a modal element that we want to close
- We replace `@click` in the button with a custom `@close` that on click emits event `close` which makes the root instance update the `showModal` property

Component Registration

- Use kebab-case or PascalCase

Global Registration

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

- Can be used in any root Vue template *after* registration
- Applies to all subcomponents, even available inside each other
- If unused, unnecessarily increases amount of JS
- Local registration ideal

Local Registration

```
var ComponentA = { /* ... */ }
var ComponentB = { /* ... */ }

new Vue({
  el: '#root',
  components: {
    'component-a': ComponentA,
    'component-b': ComponentB
  }
})
```

- Unavailable in subcomponents

```
var ComponentA = { /* ... */ }

var ComponentB = {
  components: {
    'component-a': ComponentA
  },
  // ...
}
```

ex. `ComponentA` available in `ComponentB`

More Props

- Use kebab-case

Passing Static/Dynamic Props

```
<blog-post title="My journey with Vue"></blog-post>
```

- Passed by static value

```
<blog-post
  v-bind:title="post.title + ' by ' + post.author.name"
></blog-post>
```

- Dynamic assignment using `v-bind`

Passing in a Number/Boolean/Array/Object

Prop Type Validation

Slots

```
<epic>hey hey hey</epic>

template:
<h1><slot>oi oi oi</slot></h1>
```

- `<slot></slot>` is replaced by `hey hey hey`
- `oi oi oi` is default slot content

Slot Scope

- Everything in the parent template is compiled in the parent scope; Everything in the child template is compiled in the child scope;

Named Slots

```
template:
<header></header>

<section></section>

<footer></footer>
```

- Using an unnamed slot will only change default slot

```
<slot name="header"></slot>
...
```

- Use named slots

```
<div slot="header">big title</div>

<template slot="header">big title</template>
```

- `template` does not show in actual html

Danny Wu