VueJS

Build amazing Apps

while smiling (-:

MisterBit

The enter of the JS frameworks

- Web development has changed…
  - HTML is being used to build applications instead of documents
- For **large scale** projects, we just cannot get around with plain *javascript* or *jquery* any more
- We need a powerful framework to support all various aspects and life cycle of a web application

MisterBit

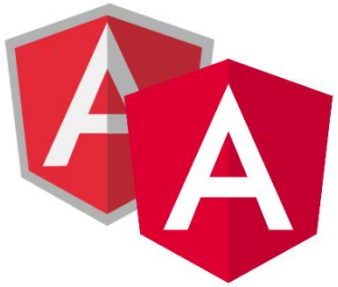
Good frameworks bring along

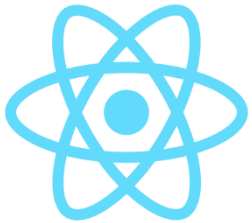
- A solid foundation so we can focus on our unique challenge
- Good separation of concerns
- Making the app easier to extend, maintain, and test

MisterBit

The enter of the JS frameworks

- We had Angular1 (now called angularJS)
  - It was great but had built-in problems
- React solved one of those problems
  - But took an idealistic approach
- New Angular has entered the scene
  - But is so complicated
- Vue is like a fresh breeze of air!





What's VueJS

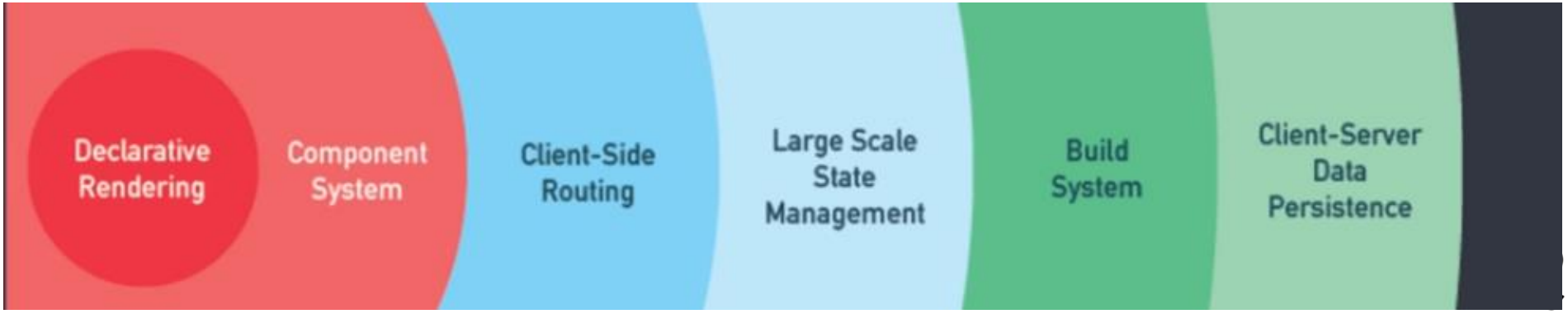
- Framework for building user interfaces.
- Easy to pick up and integrate with other libraries or existing projects.
- Capable of powering modern progressive web applications

  – Here is an Hello World fiddle.

Mister Bit

Why VueJS

- Its amazingly simple
  - It has great documentation
- Its battle tested
  - Used in large successful projects
- It can be used to build any app
  - simple to large applications
- Its small – 19KB!
  - minified and gziped



Feature Rich

Declarative Rendering

Render data to the DOM using template syntax:

```
<div id="app">
    <span>
        {{ msg}}
    </span>
</div>
```

```
var app = new Vue({
  el: '#app',
  data: {
      msg: 'Hello Vue!‘ + Date.now()
  }
})
```

Conditional Rendering

Toggle the presence of an element

```
<p v-if="shouldRender"> Best P Ever </p>
```

```
<h1 v-if="ok">Yes</h1>
<h1 v-else>No</h1>
```

MisterBit

Conditional Rendering v-if

```
<div v-if="type === 'A’”>
    Aba
</div>
<div v-else-if="type === 'B’”>
    Baba
</div>
<div v-else-if="type === ‘S’”>
    Saba
</div>
<div v-else>
    Not A/B/S
</div>
```

Loops

Looping through an array:

```
<ul>
  <li v-for="(pet, idx) in pets">
      Pet {{idx}} - {{ pet.name  }}
  </li>
</ul>
```

Handling Events

Use the v-on directive to attach event listeners that invoke methods on our Vue instances:

```
<span v-if=“toShow">
   Hello {{userName}}
</span>
<button v-on:click=“toShow = !toShow“>
    Toggle
</button>
```

Event handling

We use the v-on directive to listen to DOM events and run some JavaScript when they're triggered.

```
<button v-on:click="counter += 1">Add 1</button>
<p>The button above has been clicked {{ counter }} times.</p>
```

Usually, we will use methods:

```
<button v-on:click="greet">Greet</button>
```

```
methods: {
  greet (event) {
    // `this` inside methods points to the Vue instance
    alert('Hello ' + this.userName + '!')
    // `event` is the native DOM event
    console.log(event.target)
  }
}
```

Event handling

we can also use methods in an inline JavaScript statement:

```
<button v-on:click="say('hi')">Say hi</button>
<button v-on:click="warn('Sure?', $event)">Submit</button>

methods: {
  warn(message, event) {
    // now we have access to the native event
    if (!confirm(message)) event.preventDefault()
  },
  say(msg) {console.log(msg)}
```

Note: passing 'this' from the template will pass the window (not useful)

Two way data binding

The v-model directive makes two-way binding between form input and our model

```
<p>{{ message }}</p>
<input v-model="message">
```

Interpolations

- Using the double-curly: {{ x }} to render a value to the DOM is called Interpolation
- You can render a valid JS expression, e.g:

  {{ (isValid)? msg + '!' : 'Err at' + Date.now() }}

- Side Note: you cannot use *anything* inside interpolation
  - Only few globals are accessible.

v-bind

- Use the **v-bind** directive to bind data to properties:
  - <a v-bind:href = “book.purchaseUrl"
  - <img v-bind:title = “book.name“ v-bind:src = “book.imgUrl”

- NOTE - we cannot use interpolations inside property values (bad code: <a href="{{url}}">)

Binding HTML Classes

We can pass an object to *v-bind:class* to dynamically toggle classes:

```
<div class="static“
      v-bind:class="{ active: isActive, 'text-danger': hasError }">
          Thanks!
</div>
```

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

Renders: `<div class="static active">Thanks!</div>`

Computed

Make our UI more declarative:

computed properties are cached
based on their reactive dependencies!

Instead of:

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

```
<p> {{ msg }} </p>
<p> {{ reversedMsg }}  </p>
```

```
var vm = new Vue({
  el: '#example',
  data: {
    msg: 'Hello'
  },
  computed: {
    reversedMsg() {
      // `this` points to the vm instance
      return this.msg.split('').reverse().join('')
    }
  }
})
```

MisterBit

Binding HTML Classes – from data

The class object does not have to be inlined, so this reads better:

```
<div class="static” v-bind:class="classObject"></div>

data: {
  classObject: {
    active: true,
    'text-danger': false
  }
}
```

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

(nice, but even better thing is coming next slide)

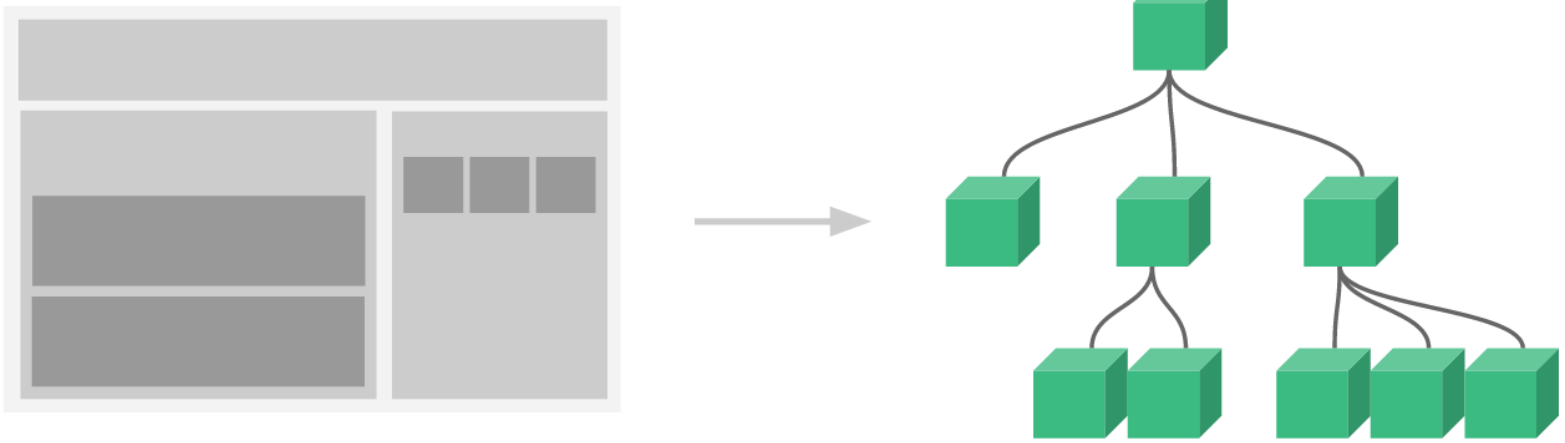
Binding HTML Classes – with computed

Best strategy is using a computed property:

```
<div v-bind:class="classObject"></div>
```

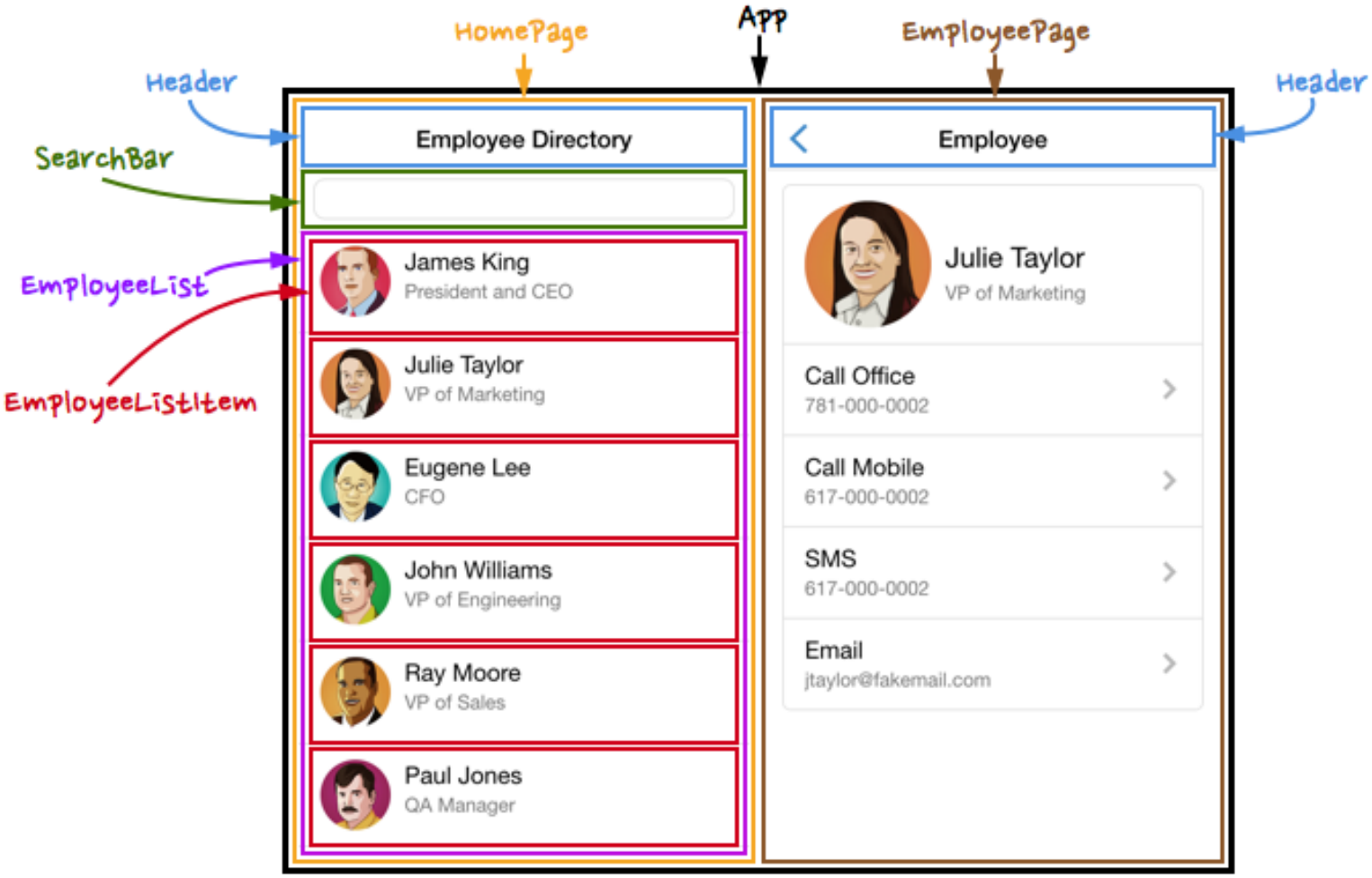
```
data: {
  isActive: true,
  error: null
},
computed: {
  classObject() {
    return {
      active: this.isActive && !this.error,
      'text-danger': this.error && this.error.type === 'fatal',
    }
  }
}
```

Components



Building UI with Components

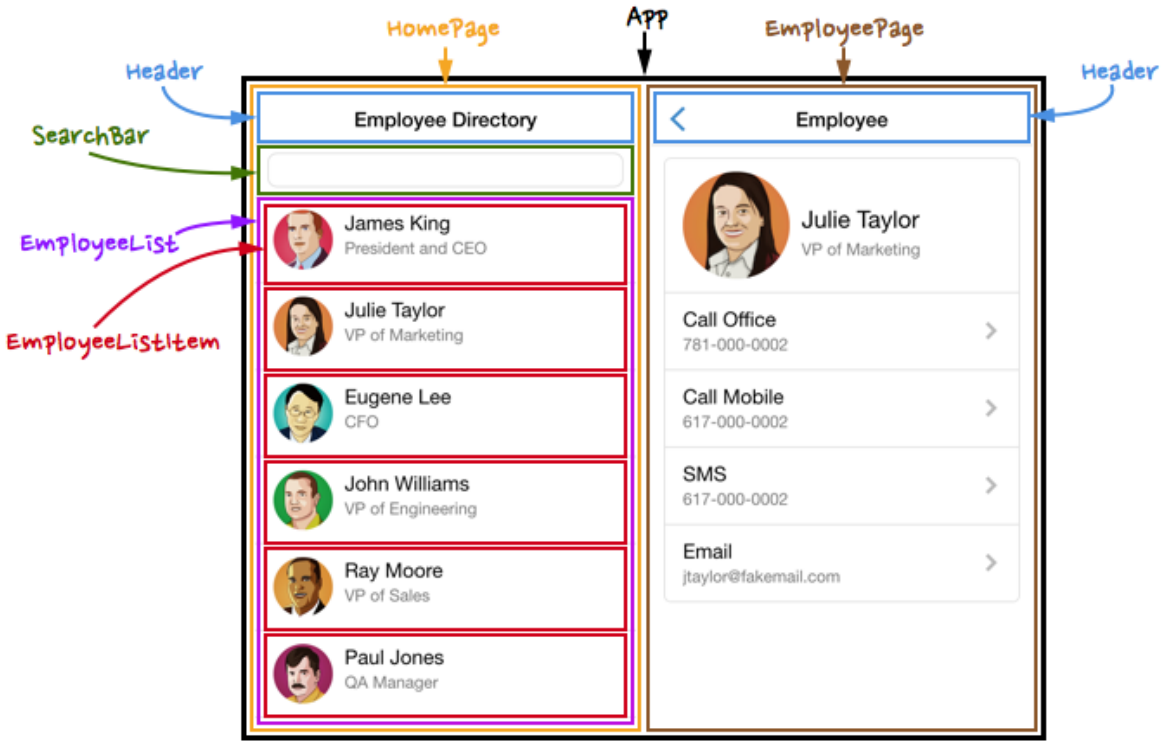
Web apps today are made out of *components:*



MisterBit

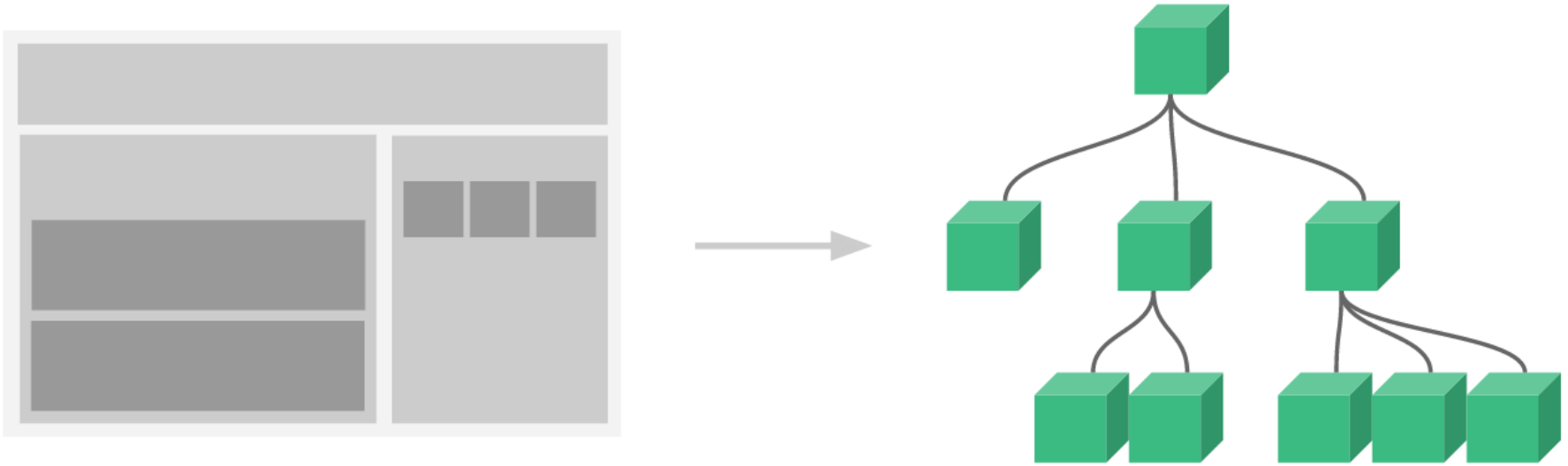
Define: Component

- A component is a small, reusable chunk of code, markup and style that is responsible for one job.
- That job is often to render some HTML



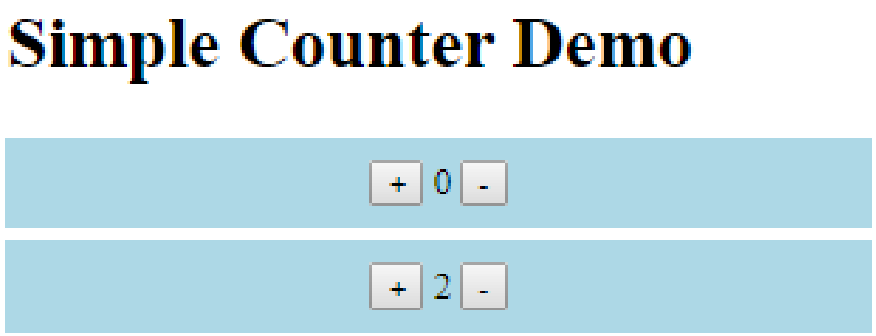
Components architecture

Building big applications from small, self-contained, reusable components.



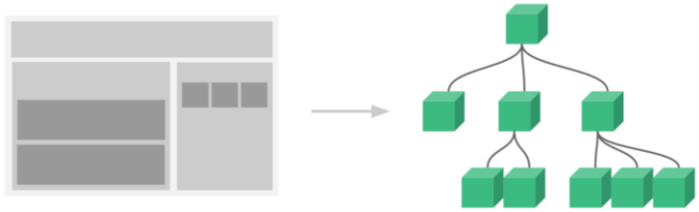
Components architecture

Example: simple-counter component



Composing with Components

The UI is broken to small pieces:



```
Vue.component('user-info', {
    template: '<section>{{user.name}}</section>'
})
Vue.component('car-details', {
    template: '<section> Car Details </section>'
})
```

components - global

This is how we register a global component:

```
// register
Vue.component('my-component', {
  template: '<div>A custom component!</div>'
})
// create a root instance
new Vue({
  el: '#example'
})
```

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

WILL RENDER:

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

It is a good practice to adhere to the W3C rules for custom tag names
(tldr : all-lowercase, must contain a hyphen)

components – data()

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 function.

```
var Comp = {
  template: '<div>A custom component!</div>‘,
  data() {
    return {};
  }
}
```

components – created()

Called (by Vue) when the component gets created:

```
var Comp = {
  template: '<div>A custom component!</div>‘,
  created() {
    console.log(‘Alive’, this);
  }
}
```

Props with binding

Similar to binding normal properties on native dom elements, we can also use **v-bind** for dynamically binding props of a component to data on the parent.

```
<input v-bind:value="myPet.name">
<h1 v-bind:title="desc">Hello</h1>

<pet-play v-bind:pet="myPet"></pet-play>


<!– Caveat: this passes down plain strings -->
<comp some-prop="value"></comp>
<comp some-prop="1"></comp>
```

Passing Props

This is how:

```
Vue.component('todo-item', {
  props: ['todo'],
  template: '<li>{{ todo.text }}</li>'
})
```

```
<ol>
  <todo-item v-for="item in myList" v-bind:todo="item"></todo-item>
</ol>
```

Using v-on with Custom Events

Every Vue instance can Trigger an event with: this.$emit(eventName)

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

See full demo here

Composing components

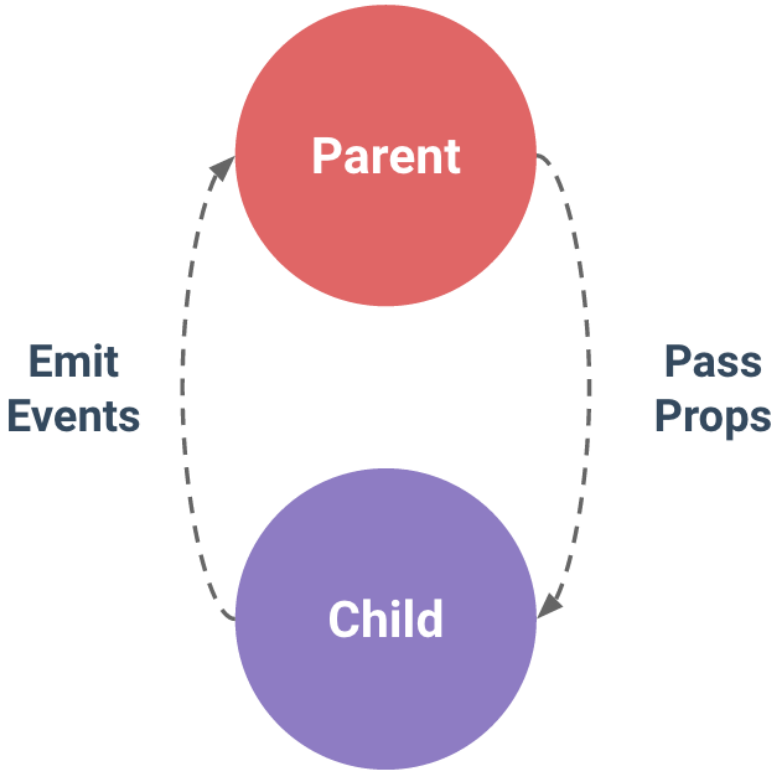
Components are meant to be used together, most commonly in parent-child relationships.

This is achieved via: **props down, events up**

```
Vue.component('child', {
  props: [msg'],
  template: '<span>{{ msg }}</span>'
})
```

Using it:

```
<child msg="hello!"></child>
```



List Rendering v-for

Rendering component in a loop and passing inputs to props:

```
<item-preview
        v-for="(item, index) in items"
        v-bind:item="item"
        v-bind:index="index">
</item-preview>
```

Shorthands

Vue.js provides special shorthand for two of the most often used directives, v-bind and v-on

```
<!-- full syntax -->
<a v-bind:href="url"></a>

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

```
<!-- full syntax -->
<a v-on:click="doSomething"></a>

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

No worries - these chars are syntactically valid in HTML and they do not appear in the final rendered markup anyways.

Vue devtools

Install vue-devtools extension

- See selected: $vm0

Enough for today....

MisterBit

VueJS

Forms

MisterBit

V-model them all

Building forms is possible using the v-model directive that creates two-way data bindings:

```
<p style="white-space: pre">{{ message }}</p>
<textarea v-model="message" placeholder="add multiple lines"></textarea>

<input type="checkbox" value="Jack" v-model="checkedNames“ />  Jack
<input type="checkbox" value="John" v-model="checkedNames“ /> John

<input type="radio" value="One" v-model="picked"> One
<input type="radio" value="Two" v-model="picked"> Two

<select v-model="selected" multiple>
<option>A</option>
<option>B</option>
<option>C</option>
</select>

<select v-model="selected">
<!-- inline object literal -->
<option v-bind:value="{ number: 123 }">Something</option>
</select>

<button type=“submit" :disabled=“isValid”>Save</button>
```

MisterBit

v-model modifiers

the v-model directive to create two-way data bindings:

```
<!-- synced after "change" instead of "input" -->
<input v-model.lazy="msg" >

<!-- typecast as a number -->
<input v-model.number="age" type="number">

<input v-model.trim="msg">
```

VueJS

More About Event Handling

MisterBit

Event handling - Event Modifiers

These are event modifiers used with v-on:

- **.stop**
- **.prevent**
- .capture
- .self
- .once

```
<!-- the click event's propagation will be stopped -->
<a v-on:click.stop="doThis"></a>

<!-- the submit event will no longer reload the page -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- modifiers can be chained -->
<a v-on:click.stop.prevent="doThat"></a>

<!-- just the modifier -->
<form v-on:submit.prevent></form>

<!-- use capture mode when adding the event listener -->
<div v-on:click.capture="doThis">...</div>
```

Most are exclusive to native DOM events (.once is not)

```
<!-- only trigger handler if event.target is the element itself -->
<!-- i.e. not from a child element -->
<div v-on:click.self="doThat">...</div>

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

Event handling - Key Modifiers

Key modifiers for v-on when listening for key events:

- .delete (captures both “Delete” and “Backspace” keys)
- .enter
- .tab
- .esc
- .space
- .up
- .down
- .left
- .right

```
<!-- only call vm.submit() when the keyCode is 13 -->
<input v-on:keyup.13="submit">

<!-- same as above -->
<input v-on:keyup.enter="submit">

<!-- also works for shorthand -->
<input @keyup.enter="submit">
```

You can also define custom key modifier aliases via the global config.keyCodes object:

```
// enable v-on:keyup.f1
Vue.config.keyCodes.f1 = 112
```

Event handling – Modifiers Keys

You can use the following modifiers to trigger mouse or keyboard event listeners only when the corresponding modifier key is pressed:

- .ctrl
- .alt
- .shift
- .meta (⌘ **Mac /** ⊞ **windows, etc**)

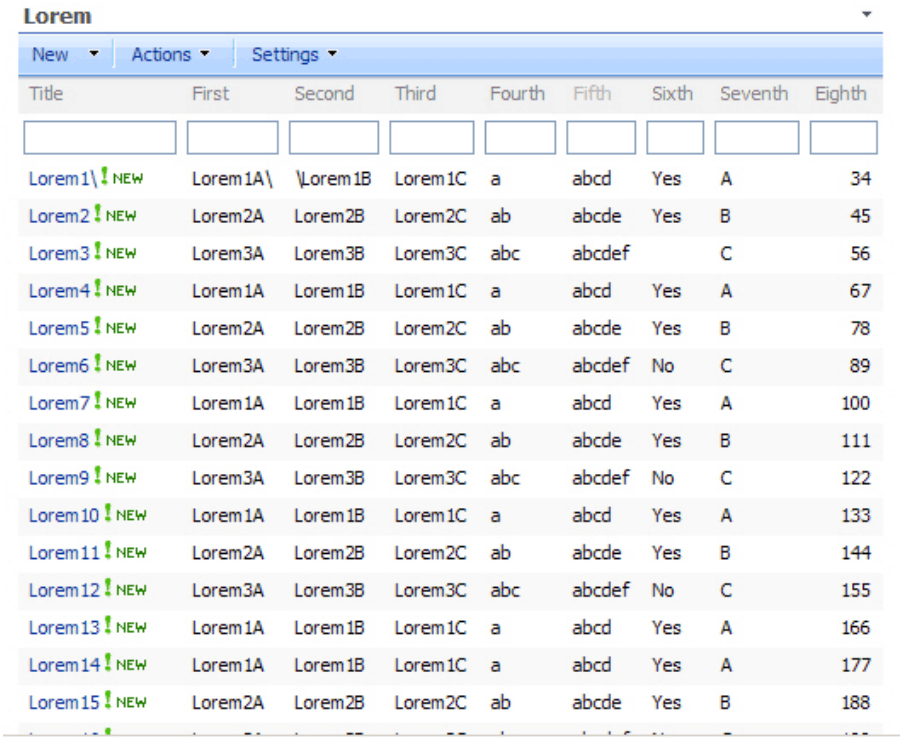
```
<!-- Alt + C -->
<input @keyup.alt.67="clear">

<!-- Ctrl + Click -->
<div @click.ctrl="doSomething">Do something</div>
```

Play here

Displaying Filtered/Sorted Results

- Its common to display a filtered or sorted version of an array without actually mutating or resetting the original data.
- Use a computed property that returns the filtered or sorted array.



Lorem

New ▾ Actions ▾ Settings ▾

Title	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Eighth
Lorem1\ ! NEW	Lorem1A\	\Lorem1B	Lorem1C	a	abcd	Yes	A	34
Lorem2 ! NEW	Lorem2A	Lorem2B	Lorem2C	ab	abcde	Yes	B	45
Lorem3 ! NEW	Lorem3A	Lorem3B	Lorem3C	abc	abcdef		C	56
Lorem4 ! NEW	Lorem1A	Lorem1B	Lorem1C	a	abcd	Yes	A	67
Lorem5 ! NEW	Lorem2A	Lorem2B	Lorem2C	ab	abcde	Yes	B	78
Lorem6 ! NEW	Lorem3A	Lorem3B	Lorem3C	abc	abcdef	No	C	89
Lorem7 ! NEW	Lorem1A	Lorem1B	Lorem1C	a	abcd	Yes	A	100
Lorem8 ! NEW	Lorem2A	Lorem2B	Lorem2C	ab	abcde	Yes	B	111
Lorem9 ! NEW	Lorem3A	Lorem3B	Lorem3C	abc	abcdef	No	C	122
Lorem10 ! NEW	Lorem1A	Lorem1B	Lorem1C	a	abcd	Yes	A	133
Lorem11 ! NEW	Lorem2A	Lorem2B	Lorem2C	ab	abcde	Yes	B	144
Lorem12 ! NEW	Lorem3A	Lorem3B	Lorem3C	abc	abcdef	No	C	155
Lorem13 ! NEW	Lorem1A	Lorem1B	Lorem1C	a	abcd	Yes	A	166
Lorem14 ! NEW	Lorem1A	Lorem1B	Lorem1C	a	abcd	Yes	A	177
Lorem15 ! NEW	Lorem2A	Lorem2B	Lorem2C	ab	abcde	Yes	B	188

components - local

It is common to make a component available only in the scope of another instance/component:

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

Component Naming Conventions

When registering components (or props), you can use kebab-case, camelCase, or TitleCase. Vue doesn't care.

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

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

```
<!-- alway use kebab-case in HTML templates -->
<kebab-cased-component></kebab-cased-component>
<camel-cased-component></camel-cased-component>
<title-cased-component></title-cased-component>
```

Mutating Props

Sometimes, it's tempting to try and mutate a prop:

1. The prop is used to pass in an **initial value**, the child component simply 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 tactics for these use cases are:

```
props: ['initialCounter‘, ‘petName’],
data() {
   return { counter: this.initialCounter }
}

computed: {
  normalizedName: () {
    return this.petName.trim().toLowerCase()
  }
}
```

Caveat: Native Events on Custom Elements

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:

```
<my-component @click.native="doTheThing"></my-component>
```

VueJS

Routing

Going different places

MisterBit

Routing in a VueJS Application

- download vue-router
- main.js:
  - import routes from './routes.js‘
  - const router = new VueRouter({routes})
  - add router to root Vue
- routes.js:
  - export const routes = [{path: '/car', component: CarDetails}]
- App.js
  - <router-link>
  - <router-view>

- Routing Modes (Hash vs History)
  const router = new VueRouter({routes, mode: 'history'})

MisterBit

Routing in a VueJS Application

- Navigating from Code (Imperative Navigation):
  this.$router.push('/');
- Setting Up Route Parameters
  Add route: /car/:id
- Fetching and Using Route Parameters
  const carId = this.$route.params.id;

MisterBit

Routing in a VueJS Application

- Navigating with Router Links
  - Instead of <a> use: <router-link to='/car'>
  - Note it renders an <a> eventually
- Styling Active Links
  - .router-link-active {color: green}

  - ITP: Show a bootstrap nav component as a demo for: tag="li" active-class="active" exact

MisterBit

Routing in a VueJS Application

- Navigating from Code (Imperative Navigation):
  this.$router.push('/');
- Setting Up Route Parameters
  Add route: /car/:id
- Fetching and Using Route Parameters
  const id = this.$route.params.id

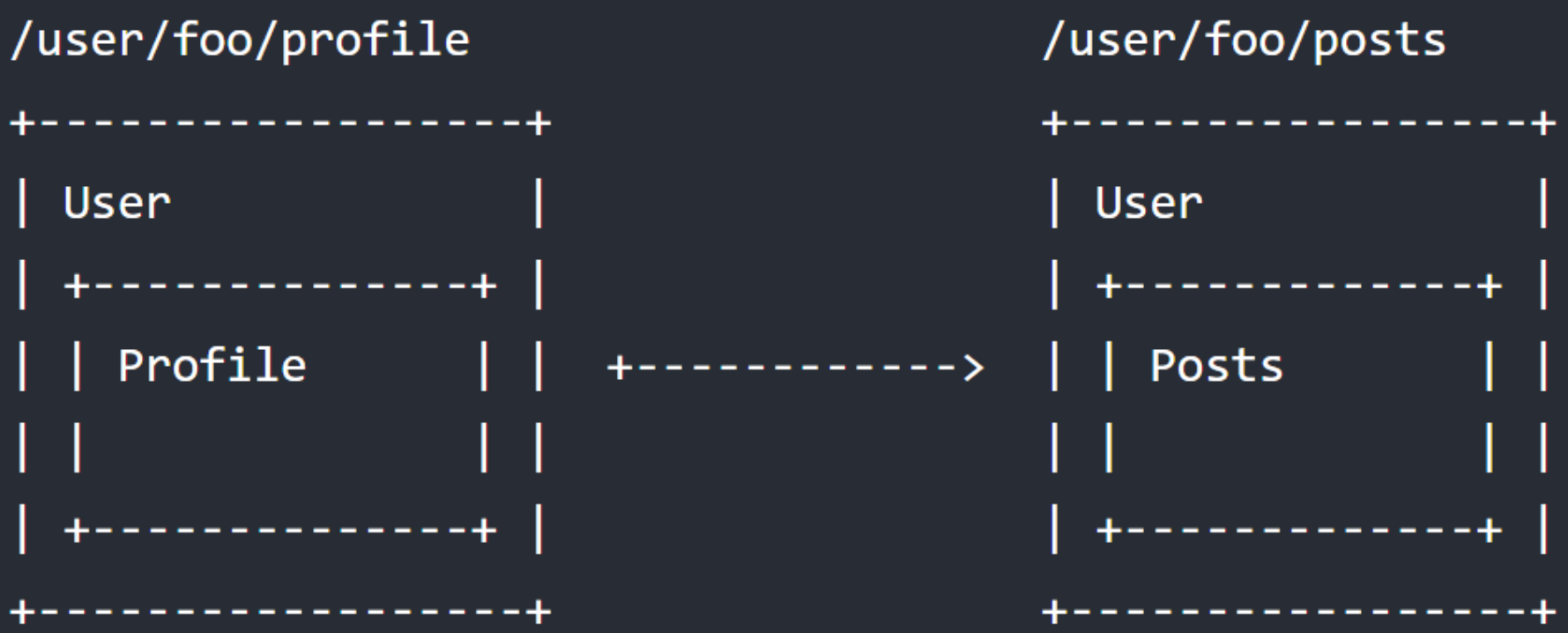
- Reacting to Changes
  in Route Parameters:

```
export default {
    data() {
      return {
          id: this.$route.params.id
      }
    },
    watch: {
        '$route'(to, from) {
            this.id = to.params.id;
        }
    }
    ,
    methods: {
        navigateToHome() {
            this.$router.push('/');
        }
    }
}
```

Nested Routes

- Real app UIs are usually composed of components that are nested multiple levels deep.
- It is also very common that the segments of a URL corresponds to a certain structure of nested components

Code Sample Here



Moving to Async Data

Most services work in an async way, returning a promise:

```
IN THE SERVICE:
function query() {
    return Promise.resolve(books);
}


IN THE COMPONENT:
data() {
        return {
            books: []
        }
},
created() {
   booksService.query()
        .then(books => {
            this.books = books;
        })
}
```

Non Parent-Child Communication

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

```
var bus = new Vue()
// in component A's method
bus.$emit('id-selected', 1)

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

In more complex cases, you should consider employing a dedicated state-management pattern (Vuex - Discussed later)

Animations

Vue provides a variety of ways to apply transition effects when items are inserted, updated, or removed from the DOM.
This includes tools to:

- Automatically apply classes for CSS transitions and animations
- integrate 3rd-party CSS animation libraries, such as Animate.css
- use JavaScript to directly manipulate the DOM during transition hooks
- integrate 3rd-party JavaScript animation libraries, such as Velocity.js
- Here is a simple example

Using Refs

- Sometimes you might still need to directly access an element
- Note that refs are available only after it is mounted to DOM, use the *mounted*() function
- Example:

```
<div id="parent">
    <div id=“map“ ref=“theMap"></div>
</div>

var elMap = this.$refs.theMap
```

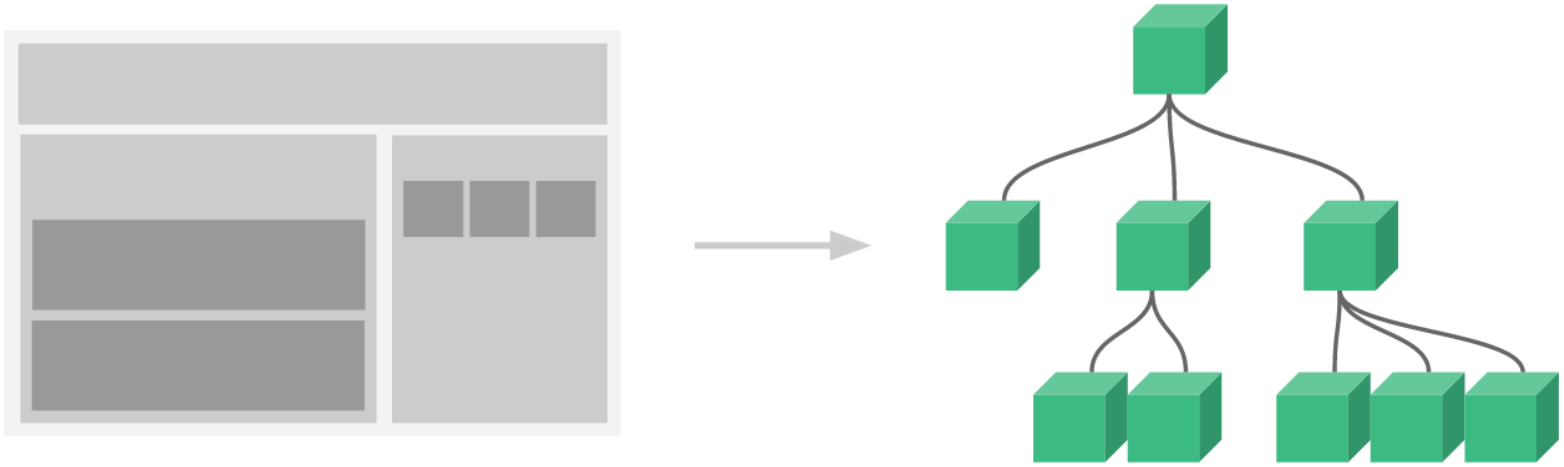
Using Refs

Another Example - you need to set the focus to an input, when some button is clicked:

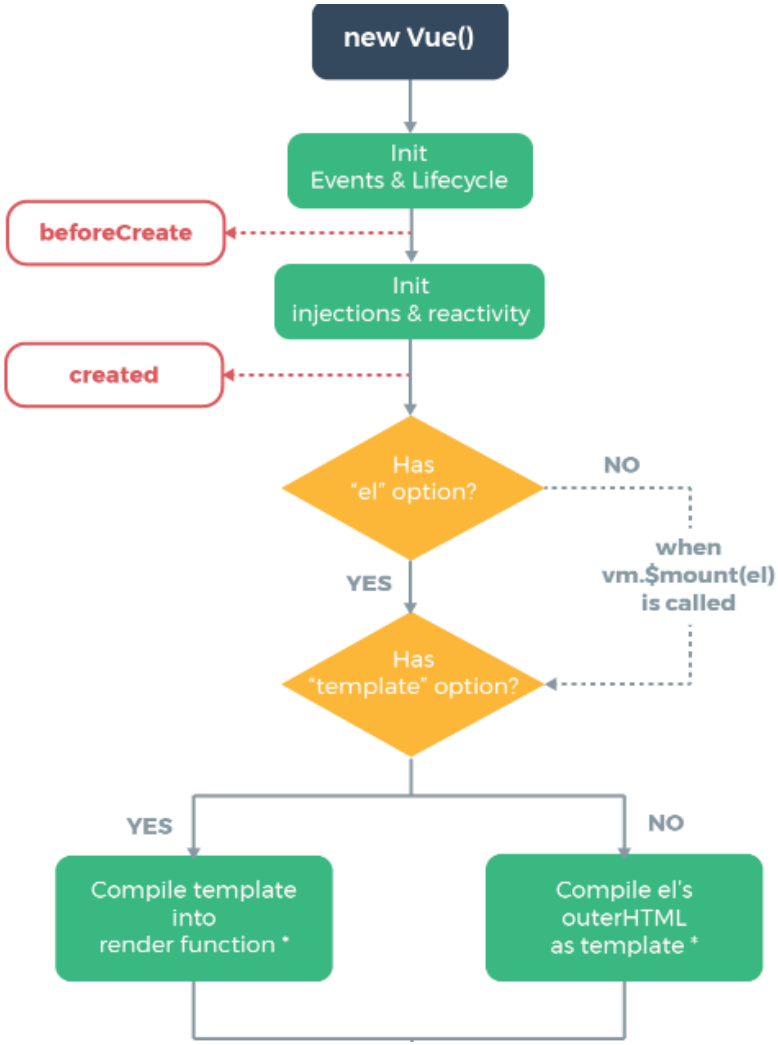
this.$refs['myInput'].focus()
this.$refs.myInput.focus()

Don't over use refs, try to relay on your model whenever possible

Going Deeper



Instance lifecycle



See it in action: https://jsfiddle.net/vyaron/jjL72zj4/

components – data()

Remember that **data must be a function?**

```
var Comp = {
  template: '<div>A custom component!</div>',
  data() {
    return {};
  }
}
```

See what happens If we cheat here

Binding Inline Styles

for the CSS property names,

You can use either camelCase or kebab-case
(use quotes with kebab-case)

```
<div :style="{‘font-style’: fStyle, fontSize: fSize + 'px' }">
   Thanks!
</div>
```

```
data: {
   fStyle: ‘italic',
   fSize: 30
}
```

Binding Inline Styles

It is often a good idea to bind to a style object so that the template is cleaner:

```
<div :style="styleObject"></div>
```

```
computed: {
  styleObject() {
     return {
        color: (this.isHappy)? ‘yellow’ : ‘gray’,
        fontSize: '13px'
     }
  }
}
```

Play here

Conditional Group Rendering - template

Achieve Conditional Groups with v-if on <template>:

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

Helps to apply v-if on multiple elements without the need to wrap them in a div

Binding HTML Classes – Array syntax

We can pass an array to *v-bind:class* to apply a list of classes:

```
<div v-bind:class="[activeClass, errorClass]">

<div v-bind:class="[isActive ? activeClass : ‘ ', errorClass]">

<div v-bind:class="[{ active: isActive }, errorClass]">
```

```
data: {
    isActive: true,
    activeClass: 'active',
    errorClass: 'text-danger'
}
```

List Rendering v-for

We can use the v-for directive to render a list of items based on an array:

```
<li v-for="(item, idx) in items">
    {{idx}} - {{ item.name }}
</li>
```

Similar to template v-if, you can also use a <template> tag with v-for to render a block of multiple elements

```
<template v-for="item in items">
  <li>{{ item.name }}</li>
  <li class="divider"></li>
</template>
```

List Rendering v-for

You can also:

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

v-for on Object

You can also use v-for to iterate through the properties of an object:

```
<li v-for="(value, key, idx) in person">
    {{ value }}
</li>

data: {
  person: {
    firstName: 'John',
    lastName: 'Doe',
    age: 30
  }
}
```

the key

It is recommended to provide a key with v-for whenever possible

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

This helps vuejs effectively correlate our model and dom, and optimize dom elements reuse

The *Key* is a generic mechanism for Vue to identify nodes, which has other uses that are not specifically tied to v-for.

Watchers on the data

- there are times when a custom watcher is necessary
- You want to perform an asynchronous action in response to changing data

Exmple:

```
Ask a yes/no question:
<input v-model="question">
<p>{{ answer }}</p>

watch: {
  question(newQuestion) {
    this.answer = 'Finding your answer...'
    this.getAnswer().then(ans => this.answer = ans)
  }
}
```

https://jsfiddle.net/vyaron/1ek3d48u/3/

MisterBit

Watchers on the Route

- Watching the route is needed when a component stayed alive, but need to update its data when route is changing
- i.e.: changed from /product/123 to /product/124

```
watch: {
      '$route.params.id’: function(id)
                                { ... }
}
```

- When needed, there is also an imperative API

Conditional Rendering v-show

v-show simply toggles the display CSS property of the element

- Note that v-show **doesn't** support the <template> syntax
- **Nor** does it work with v-else

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

Conditional Rendering v-show

```
<section v-show="isActivated">…</section>

<section v-if    =" isActivated ">…</section>
```

Usage Considerations:

v-if has higher toggle costs while v-show has higher initial render costs.

- Prefer v-show if you need to toggle something very often
- prefer v-if if the condition is unlikely to change at runtime.

VueJS

Components

Dynamic Components

MisterBit

Dynamic Components

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

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

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

Dynamic Components

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

Dynamic Component has 2 extra Lifecycle Hooks: activated(), dectivated()

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:

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