BADR

Knowledge Academy

Enjoy The Vue

Wednesday, Sept 13

Sessions Roadmap

1- Vue.js in Depth

Vue.js core concepts, Library Internals, Vue Instance Components.

2- REST APIs & Routing

Consuming REST APIs, Services, What's Routing, Nested Routing, More

3- State Management & Wrap up

Manage components state, Interaction between different components

Capstone Project

New Tab Dashboard

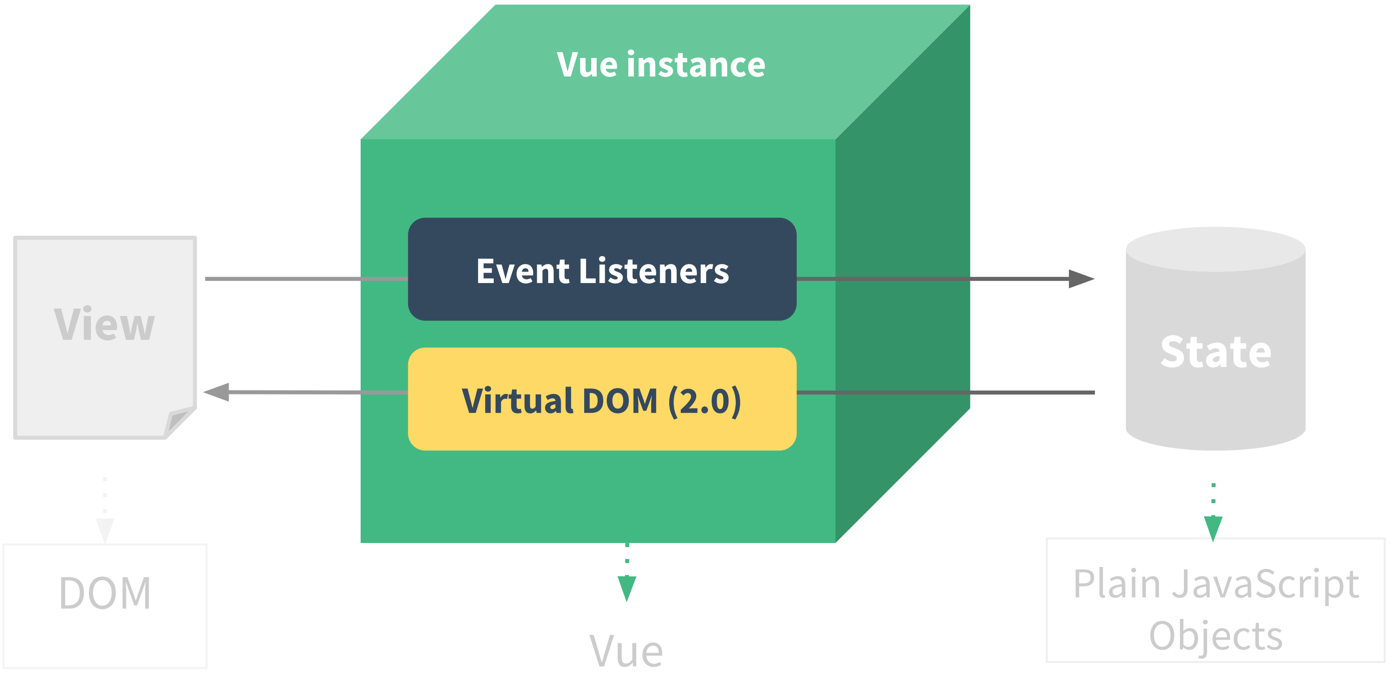
A New Tab Dashboard with a set of widgets: Weather, Todo List and New Feed Reader.

Session Agenda

- What’s Vue Instance?
- Vue Instance Components
- How does it work?
- Lifecycle hooks
- Data Binding
- Computed Values
- Class and Style Bindings
- Conditional Rendering
- List Rendering
- Form input bindings
-

-

What’s Vue Instance?



What’s Vue Instance?

Every Vue application starts by creating a new Vue instance with the Vue function:

```
var vm = new Vue({
 // options
})
```

On creating new Vue function. You need to pass an **Options Object.**

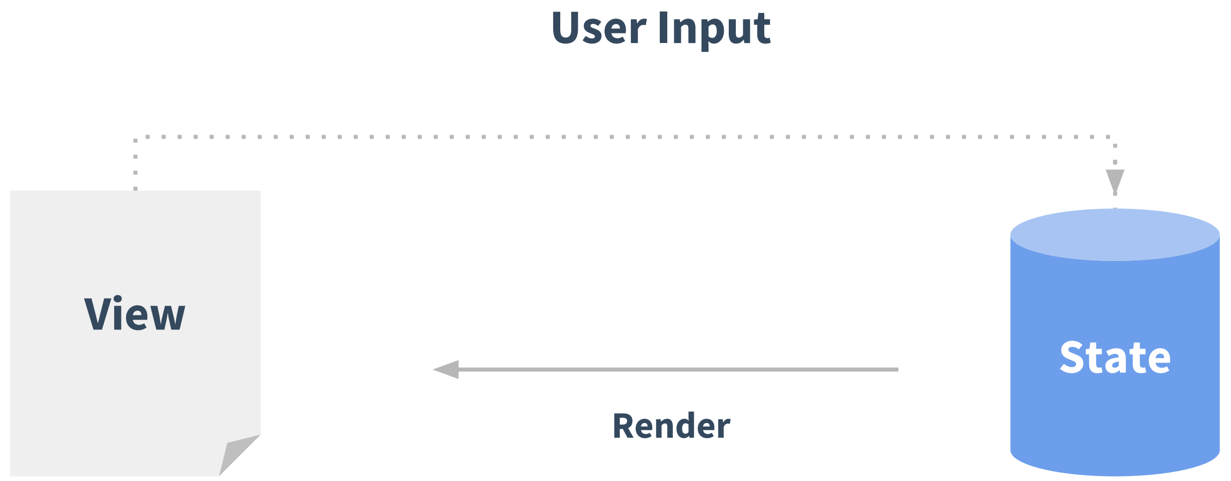
During this session we are going to explain, how you can use these options to create our desired behavior.

What’s Vue Instance?

Available Configuration you can add to the **Options object**

- **el** - the Vue instance an existing DOM element to mount on. (Not needed inside .vue files)
- **data** - The data object for the Vue instance.
- **computed** - The data object for the Vue instance.
- **watch** - An object where keys are expressions to watch and values are the corresponding callbacks.
- **methods** - The data object for the Vue instance.
- **data** - The data object for the Vue instance.
- **Lifecycle Hooks:**
    - **beforeCreate**
    - **created**
    - **beforeMount**
    - **mounted**
    - **beforeUpdate**
    - **updated**
    - **beforeDestroy**
    - **destroyed**

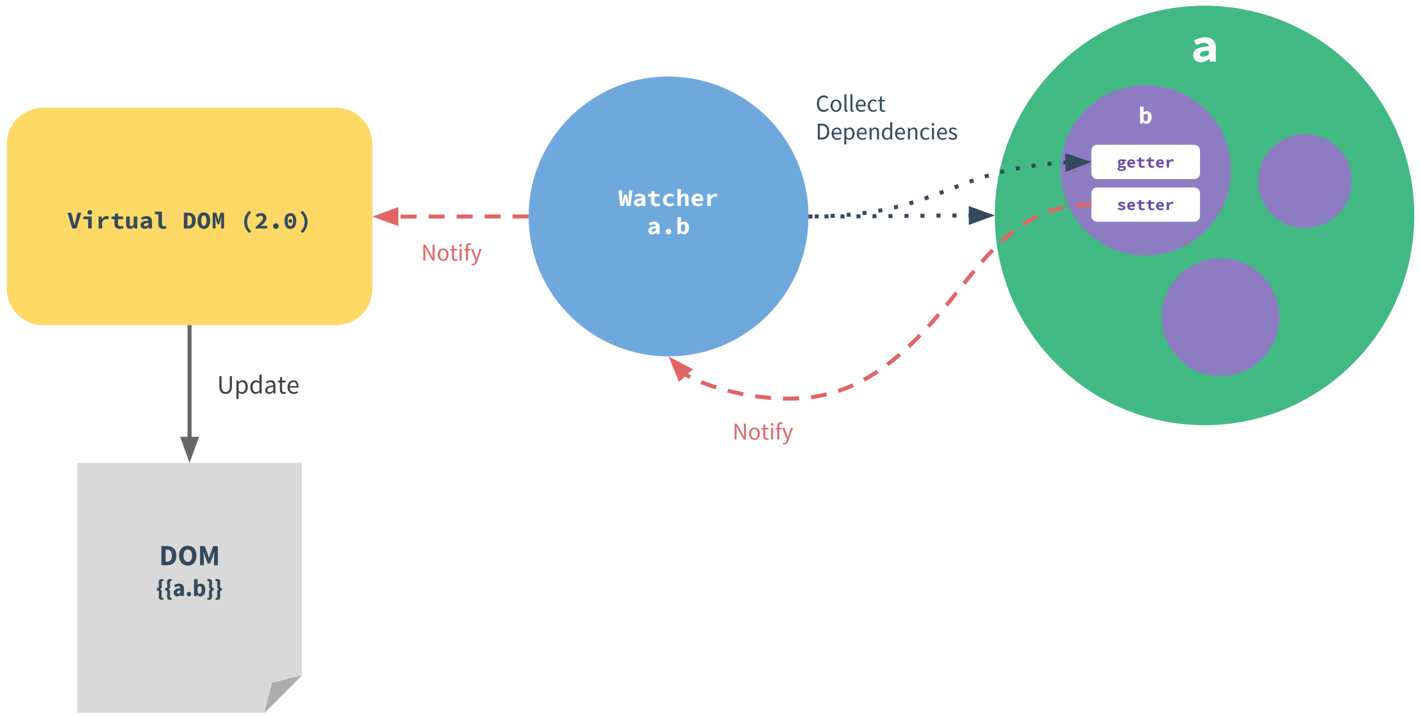
How does it work?



View is just a declarative mapping from the state

State should be the single source of truth

How does it work?

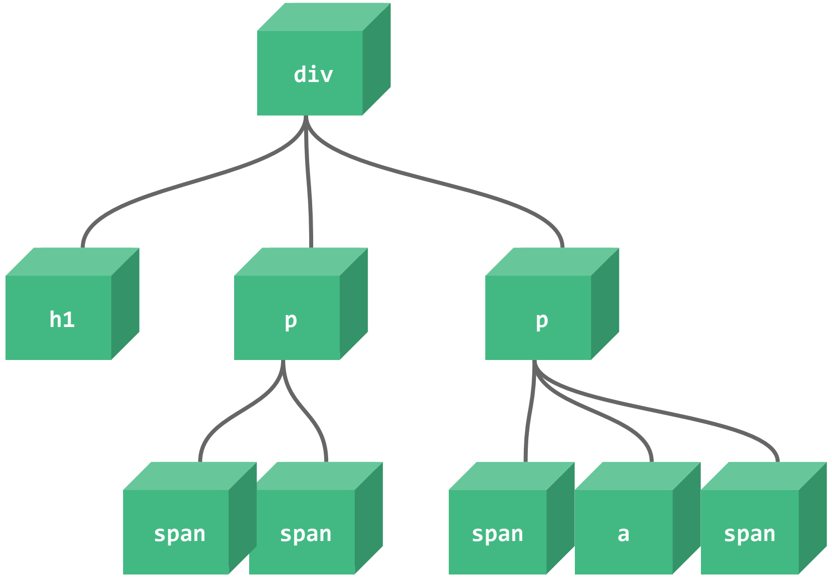


How does it work?

When passing an object to the **Vue instance** as it’s data **option**, Vue.js walks through all it’s properties and converts them to **getter/setters** using Object.defineProperty.

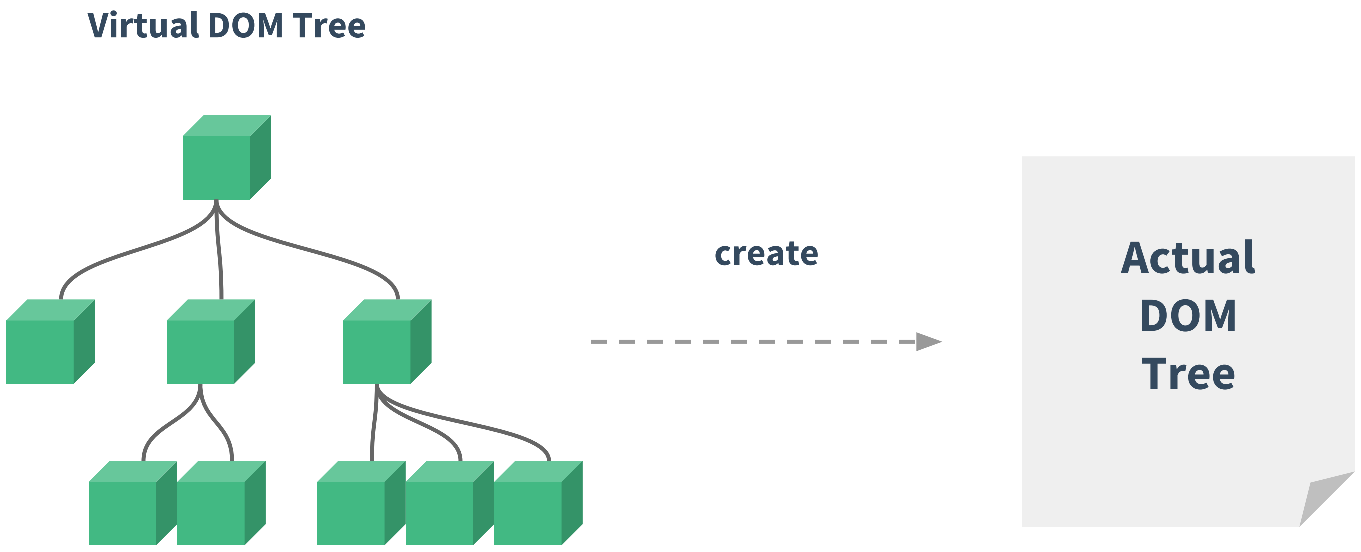
For every directive / data binding in the template, there will be a corresponding watcher object. When a dependency’s setter is called, it triggers the watcher to re-evaluate, and in turn causes its associated directive to perform DOM updates.

How does it work?



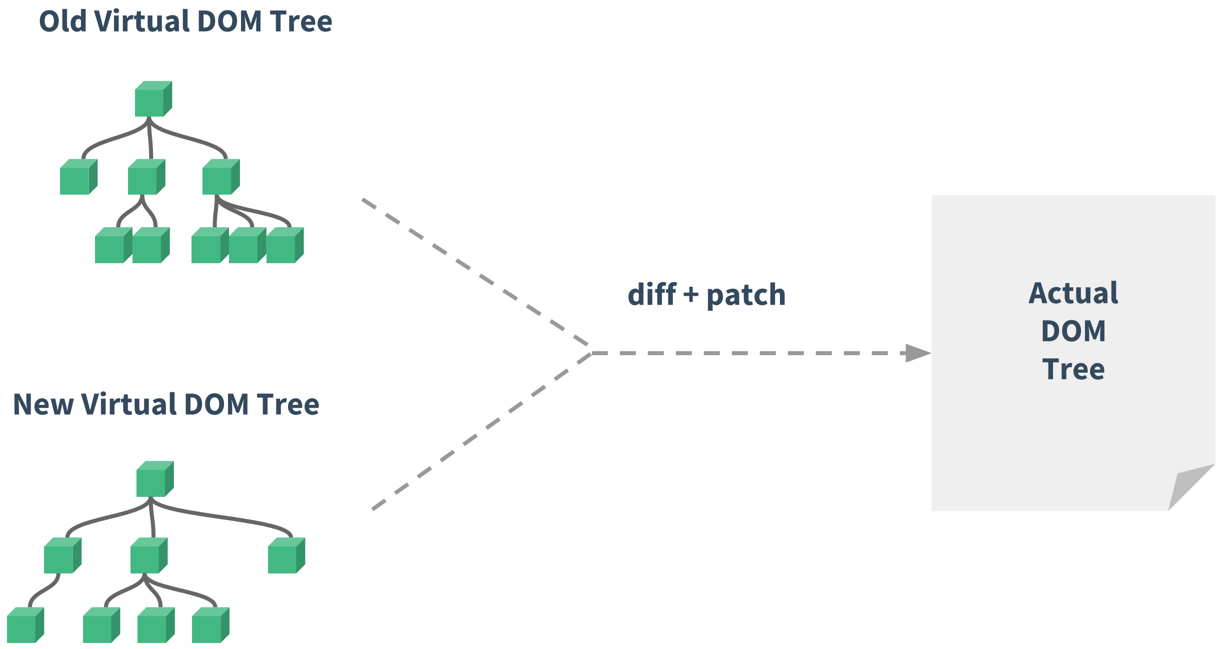
Virtual DOM

How does it work?



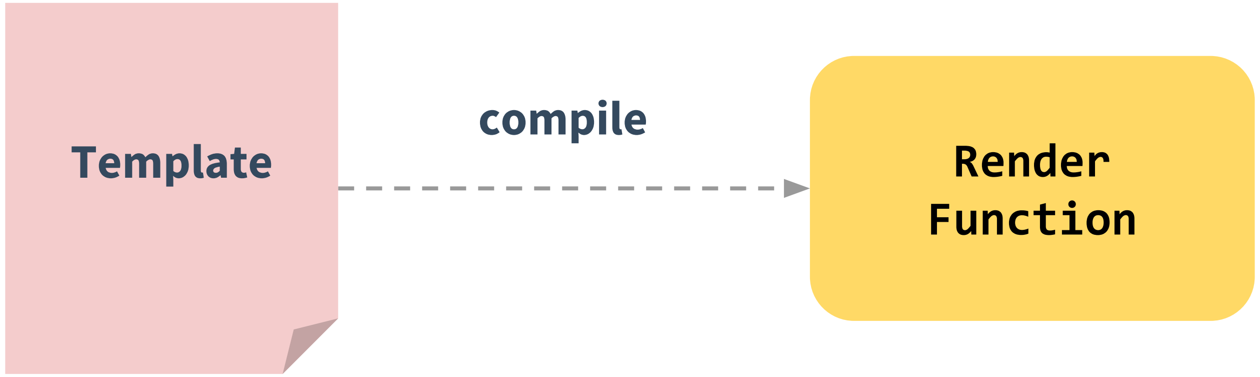
Virtual DOM initial render

How does it work?



Virtual DOM Update

How does it work?



Template Compilation

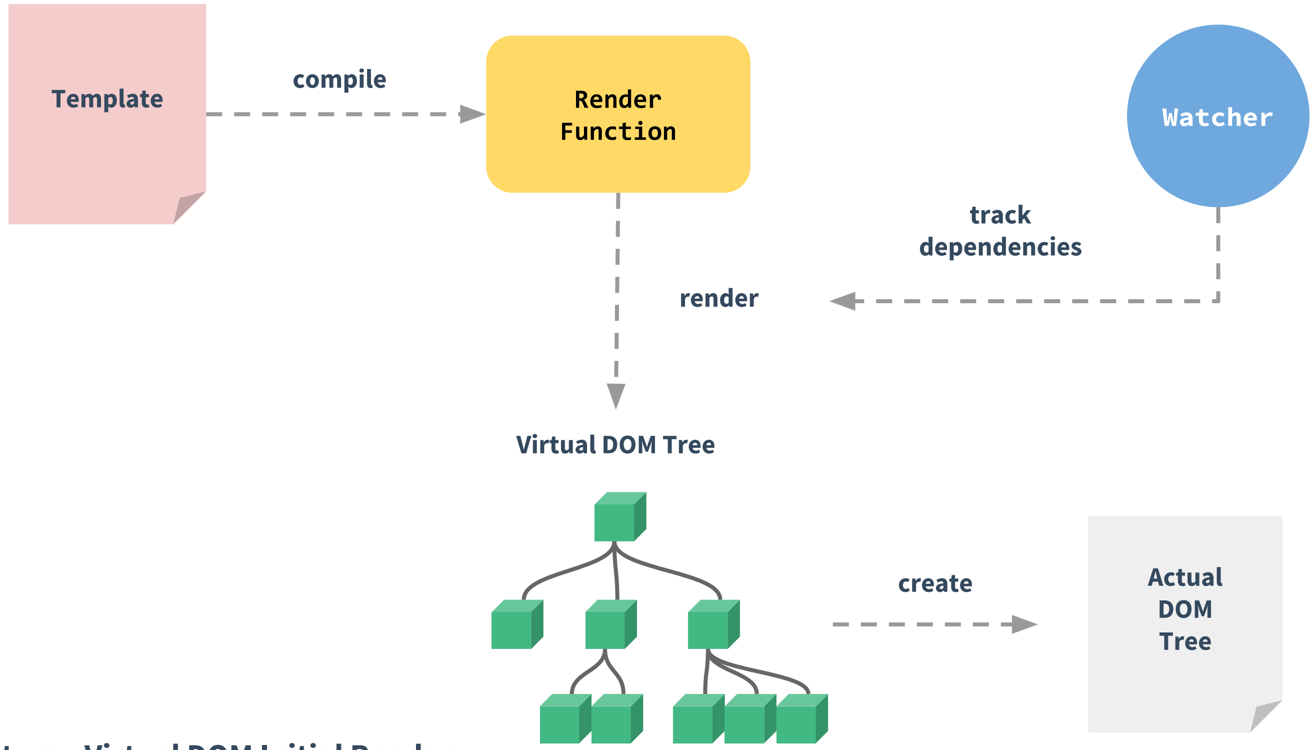
How does it work?

```
<div id="demo">
  <h1>{{msg}}</h1>
</div>
```

```
render (createElement) {
   return createElement(
      'div',
      { attrs: { id: 'demo' }},
      [createElement('h1', this.msg)]
  )
}
```

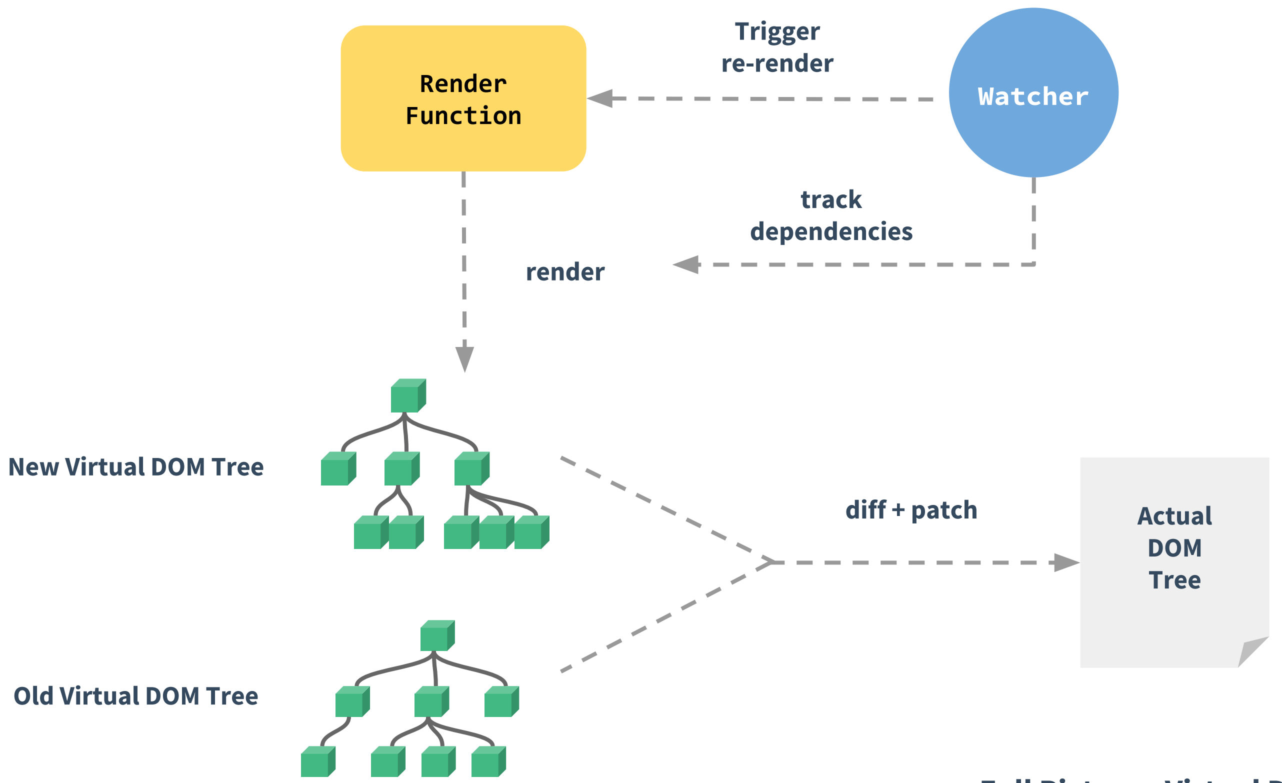
Template Compilation

How does it work?



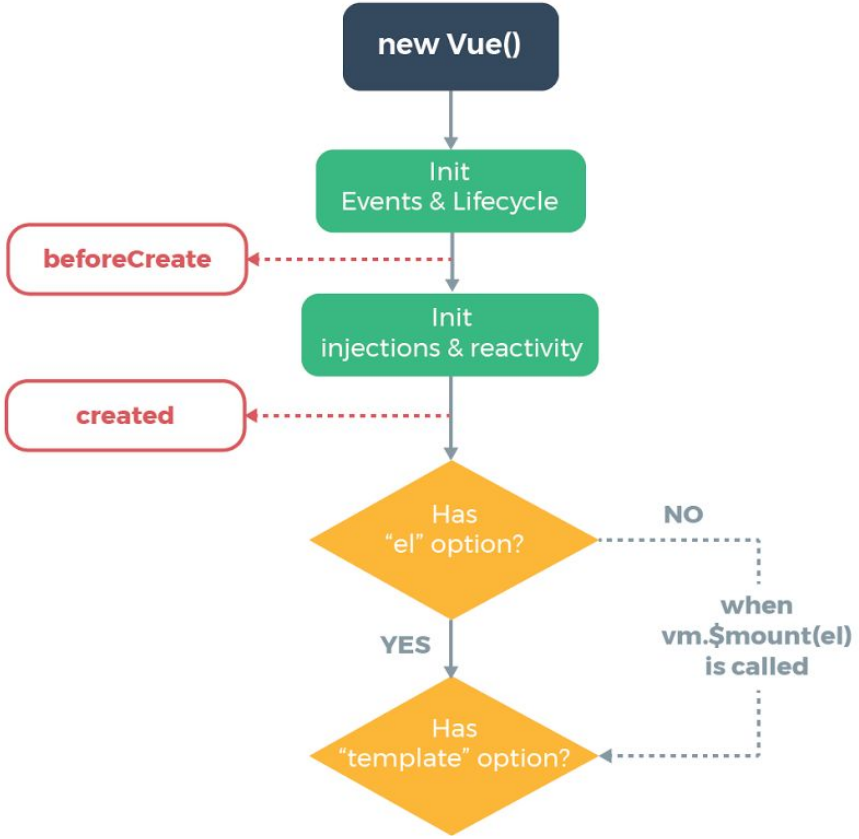
Full Picture - Virtual DOM Initial Render

How does it work?

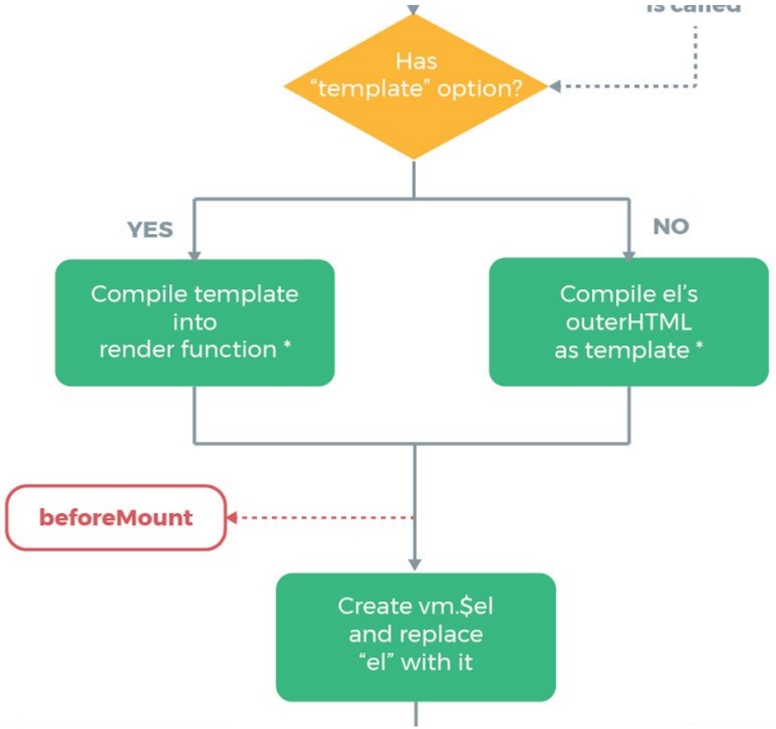


Full Picture - Virtual DOM update

Lifecycle Hooks

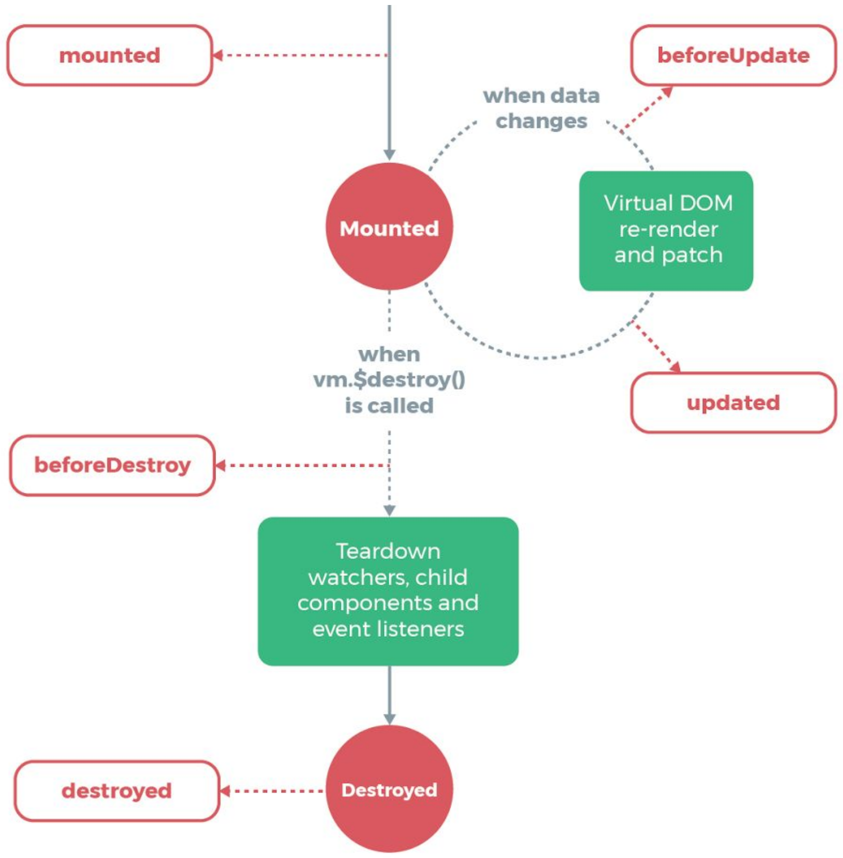


Lifecycle Hooks



is called
Has "template" option?
YES
NO
Compile template into render function *
Compile el's outerHTML as template *
beforeMount
Create vm.$el and replace "el" with it

Lifecycle Hooks


mounted
beforeUpdate
when data changes
Mounted
Virtual DOM re-render and patch
when vm.$destroy() is called
updated
beforeDestroy
Teardown watchers, child components and event listeners
destroyed
Destroyed

Data Binding

Vue.js uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying Vue instance's data.

Interpolation

Text interpolation

```
<span>Message: {{ msg }}</span>
```

One time interpolation

```
<h1 v-once>
    This won't change: {{title }}
</h1>
```

Raw HTML interpolation with triple brackets

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

JavaScript expressions

```
{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}
```

Data Binding

Directives are special attributes with the v- prefix. A directive’s job is to reactively apply side effects to the DOM when the value of its expression changes.

Directives

Renders the paragraph if greeting evaluates to true

```
<p v-if="greeting">Hello!</p>
```

Renders the paragraph if greeting evaluates to true

```
<p v-show="isActive()">Active!</p>
```

Puts the value url into the href attribute

```
<a v-bind:href="url">Open Link</a>
<a :href="url">Open Link</a>
```

Puts the value url into the href attribute

```
<a v-on:click="doSomething">
<a @click="doSomething">
```

Computed Values

In-template expressions are very convenient, but they are really only meant for simple operations. For any more complex logic you should use a **computed property.**

Template

```
<div id="example">
  <p>Original message: "{{ message }}"</p>
  <p>
      Computed reversed message: "{{reversedMessage}}"
  </p>
</div>
```

J[illegible]ipt expressions

```
data: {
  message: 'Hello'
},
computed: {
    // a computed getter
    reversedMessage: function () {
      // `this` points to the vm instance
      return
this.message.split('').reverse().join('')
    }
  }
```

Result

Original Message: “Hello”
Computed reversed message: “olleH”

Computed Values

In-template expressions are very convenient, but they are really only meant for simple operations. For any more complex logic you should use a **computed property.**

Template

```
<div id="example">
 <p>Original message: "{{ message }}"</p>
 <p>
    Computed reversed message: "{{reversedMessage}}"
 </p>
</div>
```

Result

```
Original Message: “Hello”
Computed reversed message: “olleH”
```

JavaScript expressions

```
data: {
  message: 'Hello'
},
computed: {
   // a computed getter
   reversedMessage: function () {
     // `this` points to the vm instance
     return this.message.split('').reverse().join('')
   }
 }
```

Computed Values

Computed Values vs Methods

```
<p>Reversed message: "{{ reverseMessage() }}"</p>
```

```
// in component
methods: {
 reverseMessage: function () {
   return this.message.split('').reverse().join('')
 }
}
```

Computed Values

Computed Values vs Watched Property

```
var vm = new Vue({
 el: '#demo',
 data: {
   firstName: 'Foo',
   lastName: 'Bar',
   fullName: 'Foo Bar'
 },
 watch: {
   firstName: function (val) {
     this.fullName = val + ' ' + this.lastName
   },
   lastName: function (val) {
     this.fullName = this.firstName + ' ' + val
   }
 }
})
```

```
var vm = new Vue({
 el: '#demo',
 data: {
   firstName: 'Foo',
   lastName: 'Bar'
 },
 computed: {
   fullName: function () {
     return this.firstName + ' ' + this.lastName
   }
 }
})
```

Computed Values

Computed Setter

Computed properties are by default getter-only, but you can also provide a setter when you need it

```
computed: {
 fullName: {
   // getter
   get: function () {
     return this.firstName + ' ' + this.lastName
   },
   // setter
   set: function (newValue) {
     var names = newValue.split(' ')
     this.firstName = names[0]
     this.lastName = names[names.length - 1]
   }
 }
}
```

Now when you run `vm.fullName = 'John Doe'`, the setter will be invoked and `vm.firstName` and `vm.lastName` will be updated accordingly.

Watchers

Computed Setter

Computed properties are by default getter-only, but you can also provide a setter when you need it

```
computed: {
 fullName: {
   // getter
   get: function () {
     return this.firstName + ' ' + this.lastName
   },
   // setter
   set: function (newValue) {
     var names = newValue.split(' ')
     this.firstName = names[0]
     this.lastName = names[names.length - 1]
   }
 }
}
```

Now when you run `vm.fullName = 'John Doe'`, the setter will be invoked and `vm.firstName` and `vm.lastName` will be updated accordingly.

Class and Style Bindings

```
<div class="static" :class="{ 'class-a': isA, 'class-b': isB }"></div>

<!-- Given that data = { isA: true, isB: false } this will render -->
<div class="static class-a"></div>



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

<!-- Where
      data: {
          classObject: {
            'class-a': true,
            'class-b': false
          }
      }
    Will render the same result. This can be used with computed properties. -->
```

Class and Style Bindings

Array syntax

```
<div v-bind:class="[classA, isB ? classB : '']">
<!-- This will always add the `classA` class, but `classB` will only be added if `isB` is true -->
```

also

```
<!-- You can mix array syntax with object syntax -->
<div v-bind:class="[classA, { classB: isB, classC: isC }]">
```

Class and Style Bindings

Style bindings

```
<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
<!-- Works mostly the same as with classes -->
```

Additionally

When you use a CSS property that requires vendor prefixes in v-bind:style, for example transform, Vue.js will automatically detect and add appropriate prefixes to the applied styles.

Conditional Rendering

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

The v-else element will render only if `ok` evaluates to false.

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

The final rendered result will not include the<template> element.

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

Another option for conditionally displaying an element is the `v-show` directive. The usage is largely the same

List rendering

```
var example1 = new Vue({
  el: '#example-1',
  data: {
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
  }
})
```

```
<ul id="example-1">
  <li v-for="item in items">
    {{ $index }} – {{ item.message }}
  </li>
</ul>

<!-- Where $index is the index
of the current item -->
```

or

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

<!-- Where the <template> element
won’t render -->
```

Methods and Event Handling

```
<button @click="greet('Developers', $event)">Greet</button>
```

```
var vm = new Vue({
  el: '#example',
  data: {
    name: 'Vue.js'
  },

  // define methods under the `methods` object
  methods: {
    greet: function (user, event) {
      // `this` inside methods point to the Vue instance
      alert('Hello ' + user + ' this is ' + this.name + '!')
      // `event` gives us access to original DOM event
    }
  }
})
```

Methods and Event Handling

```
<button @click="greet('Developers', $event)">Greet</button>
```

```
var vm = new Vue({
  el: '#example',
  data: {
    name: 'Vue.js'
  },

  // define methods under the `methods` object
  methods: {
    greet: function (user, event) {
      // `this` inside methods point to the Vue instance
      alert('Hello ' + user + ' this is ' + this.name + '!')
      // `event` gives us access to original DOM event
    }
  }
})
```

Event Modifiers

```
<!-- 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>

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

<!-- Support for key aliases as modifiers -->
<input @keyup.enter="submit">
```

That’s All for now

See you next Monday (18 Sept.)