Reactivity in Depth

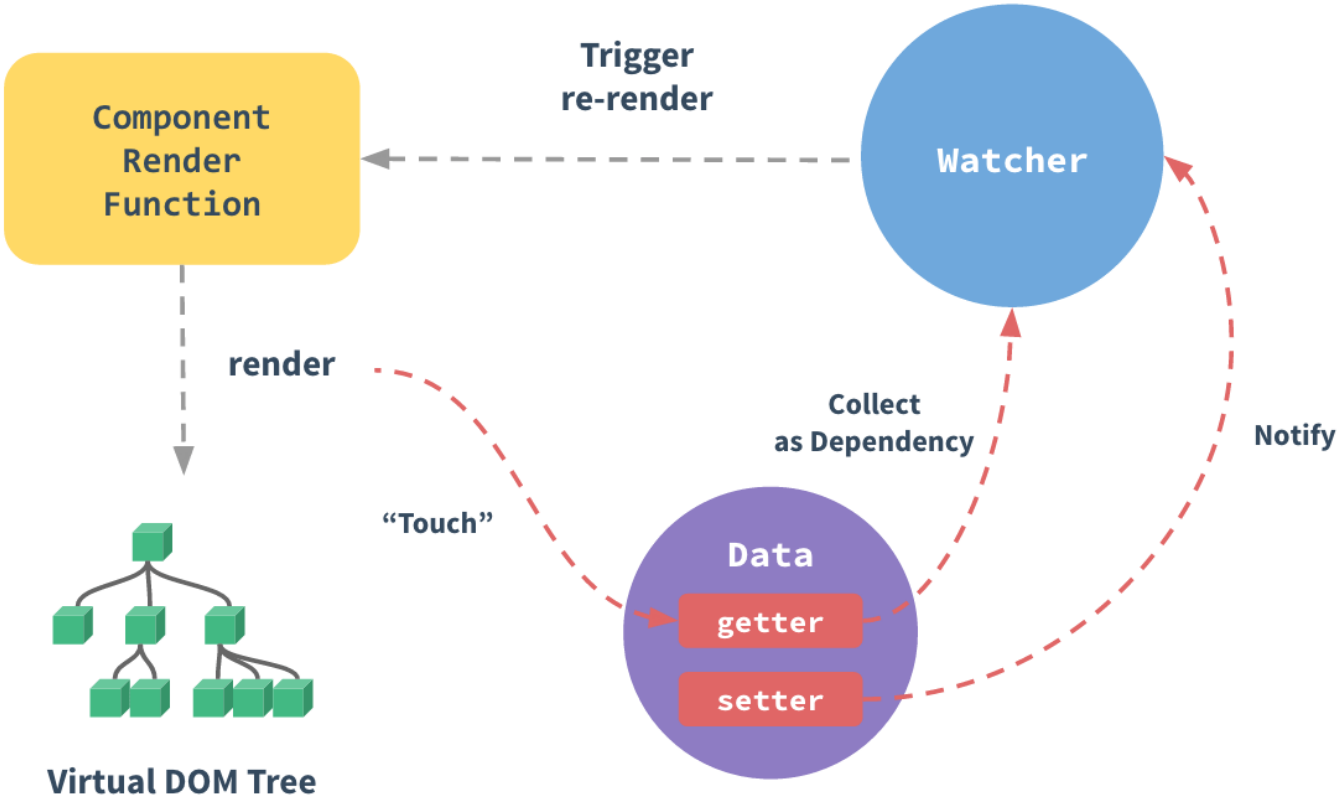
Now it's time to take a deep dive! One of Vue's most distinct features is the unobtrusive reactivity system. Models are just plain JavaScript objects. When you modify them, the view updates. It makes state management simple and intuitive, but it's also important to understand how it works to avoid some common gotchas. In this section, we are going to dig into some of the lower-level details of Vue's reactivity system.

How Changes Are Tracked

When you pass a plain JavaScript object to a Vue instance as its `data` option, Vue will walk through all of its properties and convert them to getter/setters using **Object.defineProperty**. This is an ES5-only and un-shimmable feature, which is why Vue doesn't support IE8 and below.

The getter/setters are invisible to the user, but under the hood they enable Vue to perform dependency-tracking and change-notification when properties are accessed or modified. One caveat is that browser consoles format getter/setters differently when converted data objects are logged, so you may want to install **vue-devtools** for a more inspection-friendly interface.

Every component instance has a corresponding **watcher** instance, which records any properties "touched" during the component's render as dependencies. Later on when a dependency's setter is triggered, it notifies the watcher, which in turn causes the component to re-render.



Change Detection Caveats

Due to the limitations of modern JavaScript (and the abandonment of `Object.observe`), Vue **cannot detect property addition or deletion**. Since Vue performs the getter/setter conversion process during instance initialization, a property must be present in the `data` object in order for Vue to convert it and make it reactive. For example:

```js
var vm = new Vue({
  data: {
    a: 1
  }
})
// `vm.a` is now reactive

vm.b = 2
// `vm.b` is NOT reactive
```

Vue does not allow dynamically adding new root-level reactive properties to an already created instance. However, it's possible to add reactive properties to a nested object using the `Vue.set(object, key, value)` method:

```js
Vue.set(vm.someObject, 'b', 2)
```

You can also use the `vm.$set` instance method, which is an alias to the global `Vue.set`:

```js
this.$set(this.someObject, 'b', 2)
```

`

Sometimes you may want to assign a number of properties to an existing object, for example using `Object.assign()` or `_.extend()`. However, new properties added to the object will not trigger changes. In such cases, create a fresh object with properties from both the original object and the mixin object:

```
// instead of `Object.assign(this.someObject, { a: 1, b: 2 })`
this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 })
```
JS

There are also a few array-related caveats, which were discussed earlier in the **list rendering section**.

Declaring Reactive Properties

Since Vue doesn't allow dynamically adding root-level reactive properties, you have to initialize Vue instances by declaring all root-level reactive data properties upfront, even with an empty value:

```
var vm = new Vue({
  data: {
    // declare message with an empty value
    message: ''
  },
  template: '<div>{{ message }}</div>'
})
// set `message` later
vm.message = 'Hello!'
```
JS

If you don't declare `message` in the data option, Vue will warn you that the render function is trying to access a property that doesn't exist.

There are technical reasons behind this restriction - it eliminates a class of edge cases in the dependency tracking system, and also makes Vue instances play nicer with type checking systems. But there is also an important consideration in terms of code maintainability: the `data` object is like the schema for your component's state. Declaring all reactive properties upfront makes the component code easier to understand when revisited later or read by another developer.

Async Update Queue

In case you haven't noticed yet, Vue performs DOM updates **asynchronously**. Whenever a data change is observed, it will open a queue and buffer all the data changes that happen in the same event loop. If the same watcher is triggered multiple times, it will be pushed into the queue only once. This buffered de-duplication is important in avoiding unnecessary calculations and DOM manipulations. Then, in the next event loop "tick", Vue flushes the queue and performs the actual (already de-duped) work. Internally Vue tries native `Promise.then` and `MutationObserver` for the asynchronous queuing and falls back to `setTimeout(fn, 0)`.

For example, when you set `vm.someData = 'new value'`, the component will not re-render immediately. It will update in the next “tick”, when the queue is flushed. Most of the time we don’t need to care about this, but it can be tricky when you want to do something that depends on the post-update DOM state. Although Vue.js generally encourages developers to think in a “data-driven” fashion and avoid touching the DOM directly, sometimes it might be necessary to get your hands dirty. In order to wait until Vue.js has finished updating the DOM after a data change, you can use `Vue.nextTick(callback)` immediately after the data is changed. The callback will be called after the DOM has been updated. For example:

```HTML
<div id="example">{{ message }}</div>
```

```JS
var vm = new Vue({
  el: '#example',
  data: {
    message: '123'
  }
})
vm.message = 'new message' // change data
vm.$el.textContent === 'new message' // false
Vue.nextTick(function () {
  vm.$el.textContent === 'new message' // true
})
```

There is also the `vm.$nextTick()` instance method, which is especially handy inside components, because it doesn’t need global `Vue` and its callback’s `this` context will be automatically bound to the current Vue instance:

```JS
Vue.component('example', {
  template: '<span>{{ message }}</span>',
  data: function () {
    return {
      message: 'not updated'
    }
  },
  methods: {
    updateMessage: function () {
      this.message = 'updated'
      console.log(this.$el.textContent) // => 'not updated'
      this.$nextTick(function () {
        console.log(this.$el.textContent) // => 'updated'
      })
    }
  }
})
```