4 Essential ES2015 Features For Vue.js Development

ES2015 (aka ES6) is the current specification of the JavaScript language. If you're new to JavaScript or haven't updated your JavaScript knowledge recently, there are a number of new features in ES2015 that make development much better and more enjoyable.

If you're a Vue developer, you'd benefit from learning all these new features. But as a means of triage, you might start with those features that apply to Vue specifically.

In this article, I'll show you four ES2015 features that you'll use on a daily basis with Vue. I'll provide an explanation and brief example of each.

1. Arrow functions
2. Template literals
3. Modules
4. Destructuring and spread syntax

1. Arrow functions

Arrow functions are a new way to declare JavaScript functions. They provide a shorter syntax, but differ from regular JavaScript function in other ways, too.

```
// Regular JavaScript function
function(parameters) {
  statements
}

// Arrow function
(parameters) => {
  statements
}
```

No bound `this`

An important feature of arrow functions is that they do not bind a value for `this`. Instead, they use the `this` of the enclosing context.

Consider JavaScript array methods requiring a callback function. `Array.filter`, for example, allows you to return a new array only including those items that match the filter defined by the callback.

One of the great features of Vue.js is that you can easily access data properties, computed properties and methods as `this.vueProperty` from within the context of the Vue configuration object.

If you use a regular function for a callback, however, it will bind its own value for `this`. You can't then refer to properties of the Vue object as `this.vueProperty` from within the callback, you have to manually create them somewhere in the scope of the callback.

In the below example, `size` is a data property. In the `fitlerBySize` computed property, we need to declare a variable `size` so this value can be used in the `filter` callback:

```
new Vue({
  data: {
    size: 'large',
    items: [ { size: 'small' }, { size: 'large' } ]
  },
  computed: {
    filterBySize() {
      let size = this.size;
      return this.items.filter(function(item) {
        return item.size === size;
        // Note: this.size is undefined
      });
    }
  }
});
```

An arrow function uses the `this` object from the enclosing context. In this case, it's from the `filterBySize` computed property, which has the Vue object bound to `this`, which simplifies the `filter` callback:

```
filterBySize() {
  return this.items.filter((item) => {
    return item.size === this.size;
  });
}
```

Gotcha

While arrow functions can be used effectively in many situations, it doesn't mean we should use them all the time when developing Vue. Indeed, you should never use arrow functions as function properties on Vue configuration object as these need access to the `this` context from the Vue constructor.

```
// Regular function

var regular = new Vue({
  data: {
    val: 'Hello world'
  },
  computed: {
    upperCase() {
      return this.val.toUpperCase();
    }
  }
});

console.log(regular.upperCase); // HELLO WORLD

// Arrow function

var arrow = new Vue({
  data: {
    val: 'Hello world'
  },
  computed: {
    upperCase: () => {
      return this.val.toUpperCase();
    }
  }
});

console.log(arrow.upperCase);
// Uncaught TypeError: Cannot read property 'toUpperCase' of undefined
```

Single parameter and implicit return

You can make arrow function syntax even terser in certain scenarios. If you only have one parameter for your function, you can drop the brackets `()`. If you only have one expression in your function, you can even drop the curly braces `{}`!

Here's the array filter callback from above with those shorthands implemented:

```
filterBySize() {
  return this.items.filter(item => item.size === this.size);
}
```

Read more about arrow functions on MDN.

2. Template literals

Template literals use backticks (``` `` ```) instead of double or single quotes to define a string.

Template literals allow us to do two super-useful things in Vue.js:

- Multi-line strings (great for component templates)
- Embedded expressions (great for computed properties)

Multi-line strings

Writing a template in JavaScript code is not ideal, but sometimes we want/need to. But what if the template has a lot of content? Pre-ES2015, we have two options:

First, put it all on one line:

```
Vue.component({
  template: '<div><h1></h1><p></p></div>'
});
```

This is really hard to read when the line gets long.

Second option: make it multi-line. Due to how JavaScript strings are parsed, you'll need to break the string at the end of each line and join it up again with a `+`. This makes the template much harder to edit:

```
Vue.component({
  template: '<div>' +
            '<h1></h1>' +
            '<p></p>' +
            '</div>'
});
```

Template literals solve the problem as they allowing multi-line strings without requiring the string to be broken up:

```
Vue.component({
  template: `<div>
              <h1></h1>
              <p></p>
            </div>`
});
```

Embedded expressions

Sometimes we want a string to be dynamic i.e. include a variable. This is very common in computed properties where you may want to interpolate a string in the template which is derived from a reactive Vue.js data property.

Using regular strings, we have to break up the string to insert a variable and join it back together with +. Again, this makes the string hard to read and edit:

```
new Vue({
  data: {
    name: 'George'
  },
  computed: {
    greeting() {
      return 'Hello, ' + this.name + ', how are you?'
    }
  }
});
```

By using a placeholder `${}` in a template literal, we can insert variables and other expressions without breaking the string:

```
new Vue({
  data: {
    name: 'George'
  },
  computed: {
    greeting() {
      return `Hello, ${this.name}, how are you?`
    }
  }
});
```

Read more about template literals on MDN.

3. Modules

How do you load a JavaScript object from one file into another? There was no native way to do it pre-ES2015. Using JavaScript *modules*, we can do it with *export* and *import* syntax:

file1.js

```
export default {
  myVal: 'Hello'
}
```

file2.js

```
import obj from './file1.js';
console.log(obj.myVal); // Hello
```

Modules offer two key benefits:

1. We can split our JavaScript app up into multiple files
2. We can make certain code reusable across projects

Component modules

One great use case for module files is component. Pre-ES2015, we'd need to put all our component definitions in the main file including our Vue instance e.g.

app.js

```
Vue.component('component1', { ... });
Vue.component('component2', { ... });
Vue.component('component3', { ... });

new Vue({ ... });
```

If we keep doing this, our *app.js* file will get very large and complicated. Using modules, we can put our component definitions in separate files and achieve better organization, e.g.:

component1.js

```
export default {
  // component definition
};
```

We can now import the component definition object now in our main file:

app.js

```
import component1 from './component1.js';
Vue.component('component1', component1);

...
```

An even better option for modularizing your components is to utilize Single-File Components. These make use of JavaScript modules, but also require a build tool like Webpack. Refer to this article for more info.

And to read more about JavaScript modules, start here with the import feature.

4. Destructuring and spread syntax

Objects are an essential part of Vue.js development. ES2015 makes it easier to work with object properties through some new syntax features.

Destructuring assignment

Destructuring allows us to unpack object properties and assign them to distinct variables. Take the following object `myObj`. To assign its properties to new variables, we use the `.` notation:

```
let myObj = {
  prop1: 'Hello',
  prop2: 'World'
};
```

```
const prop1 = myObj.prop1;
const prop2 = myObj.prop2;
```

Using destructuring assignment, we can do this more succinctly:

```
let myObj = {
  prop1: 'Hello',
  prop2: 'World'
};

const { prop1, prop2 } = myObj;

console.log(prop1);
// Output: Hello
```

Destructuring is useful in Vuex actions. Actions receive a `context` object which includes properties for the `state` object and the `commit` API method:

```
actions: {
  increment (context) {
   // context.state
   // context.commit(...)
  }
}
```

It's common, though, that you don't need the `state` property in an action, and only want to use the `commit` API. By using a destructuring assignment in the function profile, you can create a `commit` parameter for use in the body, reducing the verbosity of this function:

```
actions: {
  increment ({ commit }) {
    commit(...);
  }
}
```

Spread syntax

Spread syntax allows us to expand an object into a place where multiple key/value pairs are expected. To copy information from one object to another pre-2015, we'd have to do it like this:

```
let myObj = {
  prop1: 'Hello',
  prop2: 'World'
};

let newObj = {
  name: 'George',
  prop1: myObj.prop1,
  prop2: myObj.prop2
};

console.log(newObj.prop1); // Hello
```

Using the spread operator `...`, we can do this more succinctly:

```
let newObj = {
  name: 'George',
  ...myObj
};

console.log(newObj.prop1); // Hello
```

Taking an example from Vuex again, we often want to use our Vuex state properties as computed properties. Pre-ES2015, we'd have to replicate each one manually. For example:

store.js

```
new Vuex.Store({
  state: {
    prop1: ...,
    prop2: ...,
    prop3: ...
  }
});
```

app.js

```
new Vue({
  computed: {
    prop1() {
      return store.state.prop1;
    },
    prop2() {
      return store.state.prop2;
    }
    ...
  }
});
```

Vuex provides the `mapState` function which returns an object with all the Vuex state properties that you specify by providing their keys:

```
import { mapState } from 'vuex';

var state = mapState(['prop1', 'prop2', 'prop3']);
console.log(state.prop1) // { ... }
```

Using `mapState` in conjunction with the spread operator, we can combine local computed properties with those from Vuex in a very succinct way:

app.js

```
import { mapState } from 'vuex';

new Vue({
  computed: {
    someLocalComputedProp() { ... },
    ...mapState(['prop1', 'prop2', 'prop3'])
  }
});
```

That’s cool! What else?

The above are ES2015 features you’ll use straight away in a Vue project. There are, of course, many other ES2015 features that are useful in Vue.js programming. If you want to keep learning from here, I’d suggest these two as your next topics:

1. Promises. These help with asynchronous programming and can be used in conjunction with Async Components, as well as Vuex Actions.
2. `Object.assign`. This is not something you’ll directly need very often, but it will help you understand how Vue’s reactivity system works. Vue.js 3.x will likely use the new Proxies feature, so check that out too!

— ABOUT ANTHONY GORE

I'm Anthony and I'm a web developer from Sydney, Australia (though I'm often traveling abroad and working remotely). I'm the author of *Full-Stack Vue.js 2 and Laravel 5* (2017, Packt Publishing), the *Ultimate Vue.js Developers video course*, and curator of the *Vue.js Developers Newsletter*.

@ANTHONYGORE