May 6, 2020

# How to create a reactive state-based UI component with vanilla JS Proxies

Yesterday, we looked at how to create a state-based UI component with vanilla JS.

Today, we're going to learn how to use JavaScript Proxies to make it *reactive*. If you haven't read yesterday's post, you should do that now, or today's article won't make much sense.

## What is *reactivity*? #

If you're not already familiar with *data reactivity*, it simply means that the UI *reacts* to changes in your component data.

Any time you update the data, the UI automatically updates to match it. Let's see how it works!

## Adding proxies to our `data` #

Here's our state-based UI component from yesterday.

```
var Rue = function (options) {
    this.elem = document.querySelector(options.selector);
    this.data = options.data;
    this.template = options.template;
};

Rue.prototype.render = function () {
    this.elem.innerHTML = this.template(this.data);
};
```

Instead of setting `this.data` to `options.data`, we're going to first convert it into a Proxy.

Let's start by adding a `handler()` function.

*(If you're not sure what this is about, check out this article on nested arrays and objects.)*

```
var handler = function () {
    return {
        get: function (obj, prop) {
            console.log('got it!');
            if (['[object Object]', '[object Array]'].indexOf(Object.
prototype.toString.call(obj[prop])) > -1) {
                return new Proxy(obj[prop], handler());
            }
            return obj[prop];
        },
        set: function (obj, prop, value) {
            console.log('set it');
            obj[prop] = value;
            return true;
        },
        deleteProperty: function (obj, prop) {
            console.log('delete it');
            delete obj[prop];
            return true;

        }
    };
};
```

Then, in our constructor function, we'll pass `options.data` into a `new Proxy()` before assigning it to `this.data`.

```
var Rue = function (options) {
    this.elem = document.querySelector(options.selector);
    this.data = new Proxy(options.data, handler());
    this.template = options.template;
};
```

Now, whenever you update a property, the `handler()` will log messages to the console.

Here's a demo.

## Making the data reactive #

To make the data reactive, we need to call the `render()` method inside the setters, getters, and `deleteProperty()` method in the `handler()`.

In order for that to work, we need access to `this` in the `handler()`. We can pass it in as an argument. Make sure to recursively pass it into the `handler()` in the `get()` method, too.

```
var handler = function (instance) {
    return {
        get: function (obj, prop) {
            if (['[object Object]', '[object Array]'].indexOf(Object.
prototype.toString.call(obj[prop])) > -1) {
                return new Proxy(obj[prop], handler(instance));
            }
            return obj[prop];
        },
        set: function (obj, prop, value) {
            obj[prop] = value;
            instance.render();
            return true;
        },
        deleteProperty: function (obj, prop) {
            delete obj[prop];
            instance.render();
            return true;

        }
    };
};

var Rue = function (options) {
    this.elem = document.querySelector(options.selector);
    this.data = new Proxy(options.data, handler(this));
    this.template = options.template;
};
```

Now we can update our data *without* calling `app.render()`. Updates to the data will *reactively* update the UI.

Here's an updated demo.

## Overwriting the entire data object #

There's one situation where this whole thing falls apart.

If someone updates the entire `data` property, it will overwrite the Proxy and the `handler()` functions won't fire.

```
// This breaks all the things
app.data = {};
```

To prevent that, we need to make the Proxy a *private variable* inside our component.

Then, we'll add setter and getter methods using `Object.defineProperty()` that create a new Proxy if someone tries to overwrite it.

```
var Rue = function (options) {

    // Variables
    this.elem = document.querySelector(options.selector);
    var _data = new Proxy(options.data, handler(this));
    this.template = options.template;

    // Define setter and getter for data
    Object.defineProperty(this, 'data', {
        get: function () {
            return _data;
        },
        set: function (data) {
            _data = new Proxy(data, handler(_this));
            return true;
        }
    });

};
```

After setting a new Proxy, we'll need to run the `render()` method to update the UI.

However, the context of `this` won't be the `Rue()` component inside the function. To get around this, we'll store `this` to a `_this` variable and use that instead.

```
var Rue = function (options) {

    // Variables
    var _this = this;
    _this.elem = document.querySelector(options.selector);
    var _data = new Proxy(options.data, handler(this));
    _this.template = options.template;

    // Define setter and getter for data
    Object.defineProperty(this, 'data', {
        get: function () {
            return _data;
        },
        set: function (data) {
            _data = new Proxy(data, handler(_this));
            _this.render();
            return true;
        }
    });

};
```

And with that, we've got a reactive, state-based UI component.

Here's a final demo.

## What's next? #

One thing that frameworks (including smaller, lightweight ones like Reef and Preact) do is *DOM diffing*.

With *DOM diffing*, instead of using `innerHTML` to replace the UI each time, the app compares the `template()`'s output to the current UI, figures out what's different, and changes only the thing that need to be updated. DOM diffing can get pretty complicated, so we *won't* be covering that in this series.

Tomorrow, we'll instead take a look at how to batch updates into a single render for better performance.