May 7, 2020

How to batch UI rendering in a reactive, state-based UI component with vanilla JS

This week, we've looked at how to create a state-based UI component, and how to use Proxies to make it reactive.

Today is the final article of the series, and we're going to learn how to bath multiple state updates into a single render for better performance.

(*If you haven't read the first two articles in this series yet, you should. This article won't make a whole lot of sense otherwise.*)

Why does this matter?

Let's say you have a todo list component, and you're updating multiple pieces of information back-to-back.

```
todo.data.todos[3].todo = 'Bake chocolate chip cookies';
todo.data.todos[3].due = '2020/05/11';
todo.data.todos[3].complete = true;
todo.data.todos[3].alert = false;
```

With the the component we've built, `Rue()`, this would trigger four separate `render()` events. The `innerHTML` property would run four times, causing four repaints.

Not good for performance.

We want to *batch* these changes into a single `render()`. Let's look at how to make that work.

Debouncing renders with the `requestAnimationFrame()` method

We're going to implement something called *debouncing*. This is an approach to prevents a method from being run more than a specific number of times in a particular period of time.

Here's how it works:

1. Whenever a property update happens, we'll use the `requestAnimationFrame()` method to *schedule* our `render()` to run at the next animation frame event.
2. If another property update happens before it's run, we'll cancel the current `requestAnimationFrame()` and schedule a new one.
3. There is no step 3.

First, let's add a `debounce` property to the `Rue()` constructor.

```
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;
    _this.debounce = null;

    // 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;
        }
    });

};
```

Next, we'll create a function called `debounceRender()`.

We'll pass in the current instantiation as an argument (the `this` operator), and use it to store the `requestAnimationFrame()` method to the `debounce` property. We can also cancel it if needed.

```
var debounceRender = function (instance) {

    // If there's a pending render, cancel it
    if (instance.debounce) {
        window.cancelAnimationFrame(instance.debounce);
    }

    // Setup the new render to run at the next animation frame
    instance.debounce = window.requestAnimationFrame(function () {
        instance.render();
    });

};
```

Now, in our `handler()` method, any time we would call `instance.render()`, we'll instead run `debounceRender(instance)`.

```
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;
            debounceRender(instance);
            return true;
        },
        deleteProperty: function (obj, prop) {
            delete obj[prop];
            debounceRender(instance);
            return true;

        }
    };
};
```

And finally, we'll do the same thing in the `Object.defineProperty()` method, where we handle overwriting the entire `data` object.

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

Here's a demo you can play with.

Want to dig deeper into this topic?

If this way of coding seems interesting to you, you should check out ReefJS.

It works just like what we've built in these articles, but also implements DOM diffing under the hood to be less destructive to overall UI. It also has some nice extras we didn't cover in this tutorial.