May 5, 2020

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

Over the last couple of days, we learned about JavaScript Proxies and how to use them with nested arrays and objects.

Today, we're going to use vanilla JS to build a really simple state-based UI component. Then, tomorrow, we'll make it reactive with JS Proxies.

What is state-based UI?

If you've ever heard the term *state* before and weren't sure what it means, you're not alone.

State is just data.

So why do they call it state instead of data? Because there's a time-bound aspect to it. State is data at a particular moment in time. It's the present "state" of your data.

With *State-based UI*, you use your *state* or *data* to create your UI.

Rather than trying to target and manipulate elements in the DOM when the user does things, you update your data object. Then, in a template, you say, "If the data looks like this, do this. If it looks like that, do that instead."

Let me show you how it works.

How to create UI based on your data or state

Let's say we have an `#app` element, and we want to render a heading and a list of items into it.

```
<div id="app"></div>
```

Our data looks like this.

```
var data = {
    heading: 'My Todos',
    todos: ['Swim', 'Climb', 'Jump', 'Play']
};
```

We can create a template function that will accept the `data` as an argument, and use it to create an HTML string for our UI.

(*I'm using template literals for this example. They don't work in IE, so you can use old-school string concatenation or transpile them with Babel if you want.*)

```
var template = function (props) {
    return `
        <h1>${props.heading}</h1>
        <ul>
            ${props.todos.map(function (todo) {
                return `<li>${todo}</li>`;
            }).join('')}
        </ul>`;
};

// Returns an HTML string
template(data);
```

Then, you can render the HTML string into the UI using the `innerHTML` property.

(*This approach can be dangerous with third-party data. Here are some techniques you can use to protect yourself from cross-site scripting attacks.*)

```
var app = document.querySelector('#app');
app.innerHTML = template(data);
```

Here's a demo.

Creating a state-based UI component

The approach above works, but it would be nice to have a component we can use over-and-over again with different elements, data, and templates.

Let's start by creating a new component using a constructor pattern. We'll call it `Rue` (constructor functions are always capitalized).

```
var Rue = function () {
    // Awesome stuff will happen...
};
```

In our constructor function, we'll accept a selector for the element to render our template into, our data, and our template.

That's a lot, so let's have it all come in as an object of `options`. We'll save each option as a property on our constructor using the `this` keyword.

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

Now, we can *instantiate* a new version of our component by using the `new` operator and passing in our options.

Using the example from above, we would do this.

```
var app = new Rue({
    selector: '#app',
    data: {
        heading: 'My Todos',
        todos: ['Swim', 'Climb', 'Jump', 'Play']
    },
    template: function (props) {
        return `
            <h1>${props.heading}</h1>
            <ul>
                ${props.todos.map(function (todo) {
                    return `<li>${todo}</li>`;
                }).join('')}
            </ul>`;
    }
});
```

Now that we have a component with all of our information, we need a way to render it.

Rendering our state-based UI component

Because we used a constructor pattern, we can add a `render()` method to the `Rue.prototype`.

```
Rue.prototype.render = function () {
    // Render the UI
};
```

This gives us access to all of the properties set to `this` *inside* the `render()` function itself.

We'll use those properties to pass our `data` into the `template()` function, and inject the resulting HTML into `elem`.

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

Any time you want to render your UI, you can run the `render()` method on your specific instance, like this.

```
app.render();
```

Let's say you wanted to add a new item to the `todos` array. You can do this.

```
// Add a new item to the data
app.data.todos.push('Take a nap... zzzzz');

// Render an updated UI
app.render();
```

Here's a demo of the state-based UI component.

And that's it! Now we have a simple, reusable state-based UI component.

Tomorrow, we'll look at how to tie in our lessons with JS proxies to make it *reactive*.