Creating a base component

To communicate with our store, we've got three main areas that update independently based on what's stored in it. We're going to make a list of submitted items, a visual count of those items, and another one that's visually hidden with more accurate information for screen readers. These all do different things, but they would all benefit from something shared to control their local state. We're going to make a base component class!

First up, let's create a file. In the `lib` directory, go ahead and create a file called `component.js`. The path for me is:

```
~/Documents/Projects/vanilla-js-state-management-boilerplate/src/js/lib/component.js
```

Once that file is created, open it and add the following:

```js
import Store from '../store/store.js';

export default class Component {
  constructor(props = {}) {
    let self = this;

    this.render = this.render || function() {};

    if(props.store instanceof Store) {
      props.store.events.subscribe('stateChange', () => self.render());
    }

    if(props.hasOwnProperty('element')) {
      this.element = props.element;
    }
  }
}
```

Let's talk through this chunk of code. First up, we're importing the `Store` *class*. This isn't because we want an instance of it, but more for checking one of our properties in the `constructor`. Speaking of which, in the `constructor` we're looking to see if we've got a render method. If this `Component` class is the parent of another class, then that will have likely set its own method for `render`. If there is no method set, we create an empty method that will prevent things from breaking.

After this, we do the check against the `Store` class like I mentioned above. We do this to make sure that the `store` prop is a `Store` class instance so we can confidently use its methods and properties. Speaking of which, we're subscribing to the global `stateChange` event so our object can ***react***. This is calling the `render` function each time the state changes.

That's all we need to write for that class. It'll be used as a parent class that other components classes will `extend`. Let's crack on with those!

Creating our components

Like I said earlier, we've got three components to make and their all going to `extend` the base `Component` class. Let's start off with the biggest one: the list of items!

In your `js` directory, create a new folder called `components` and in there create a new file called `list.js`. For me the path is:

```
~/Documents/Projects/vanilla-js-state-management-boilerplate/src/js/components/list.js
```

Open up that file and paste this whole chunk of code in there:

```js
import Component from '../lib/component.js';
import store from '../store/index.js';

export default class List extends Component {

  constructor() {
    super({
      store,
      element: document.querySelector('.js-items')
    });
  }

  render() {
    let self = this;

    if(store.state.items.length === 0) {
      self.element.innerHTML = `<p class="no-items">You've done nothing yet &#x1f622;</
      return;
    }

    self.element.innerHTML = `
      <ul class="app__items">
        ${store.state.items.map(item => {
          return `
            <li>${item}<button aria-label="Delete this item">×</button></li>
          `
        }).join('')}
      </ul>
    `;

    self.element.querySelectorAll('button').forEach((button, index) => {
      button.addEventListener('click', () => {
        store.dispatch('clearItem', { index });
      });
    });
  }
};
```

I hope that code is pretty self-explanatory after what we've learned earlier in this tutorial, but let's skim through it anyway. We start off by passing our `Store` instance up to the `Component` parent class that we are extending. This is the `Component` class that we've just written.

After that, we declare our render method that gets called each time the `stateChange` Pub/Sub event happens. In this `render` method we put out either a list of items, or a little notice if there are no items. You'll also notice that each button has an event attached to it and they dispatch and action within our store. This action doesn't exist yet, but we'll get to it soon.

Next up, create two more files. These are two new components, but they're tiny — so we're just going to paste some code in them and move on.

First, create `count.js` in your `component` directory and paste the following in it:

```js
import Component from '../lib/component.js';
import store from '../store/index.js';

export default class Count extends Component {
  constructor() {
    super({
      store,
      element: document.querySelector('.js-count')
    });
  }

  render() {
    let suffix = store.state.items.length !== 1 ? 's' : '';
    let emoji = store.state.items.length > 0 ? '&#x1f64c;' : '&#x1f622;';

    this.element.innerHTML = `
      <small>You've done</small>
      ${store.state.items.length}
      <small>thing${suffix} today ${emoji}</small>
    `;
  }
}
```

Looks pretty similar to list, huh? There's nothing in here that we haven't already covered, so let's add another file. In the same `components` directory add a `status.js` file and paste the following in it:

```js
import Component from '../lib/component.js';
import store from '../store/index.js';

export default class Status extends Component {
  constructor() {
    super({
      store,
      element: document.querySelector('.js-status')
    });
  }

  render() {
    let self = this;
    let suffix = store.state.items.length !== 1 ? 's' : '';

    self.element.innerHTML = `${store.state.items.length} item${suffix}`;
  }
}
```

Again, we've covered everything in there, but you can see how handy it is having a base `Component` to work with, right? That's one of the many benefits of Object-orientated Programming, which is what most of this tutorial is based on.

Finally, let's check that your `js` directory is looking right. This is the structure of where we're currently at:

```
/src
├── js
│   ├── components
│   │   ├── count.js
│   │   ├── list.js
│   │   └── status.js
│   ├──lib
│   │   ├──component.js
│   │   └──pubsub.js
└───── store
        └──store.js
        └──main.js
```

Let's wire it up

Now that we've got our front-end components and our main `Store`, all we've got to do is wire it all up.

We've got our store system and the components to render and interact with its data. Let's now wrap up by hooking up the two separate ends of the app and make the whole thing work together. We'll need to add an initial state, some `actions` and some `mutations`. In your `store` directory, add a new file called `state.js`. For me it's like this:

```
~/Documents/Projects/vanilla-js-state-management-boilerplate/src/js/store/state.js
```

Open up that file and add the following:

```js
export default {
  items: [
    'I made this',
    'Another thing'
  ]
};
```

This is pretty self-explanatory. We're adding a default set of items so that on first-load, our little app will be fully interactive. Let's move on to some `actions`. In your `store` directory, create a new file called `actions.js` and add the following to it:

```js
export default {
  addItem(context, payload) {
    context.commit('addItem', payload);
  },
  clearItem(context, payload) {
    context.commit('clearItem', payload);
  }
};
```

The actions in this app are pretty minimal. Essentially, each action is passing a payload to a mutation, which in turn, commits the data to store. The `context`, as we learned earlier, is the instance of the `Store` class and the `payload` is passed in by whatever dispatches the action. Speaking of mutations, let's add some. In this same directory add a new file called `mutations.js`. Open it up and add the following:

```js
export default {
  addItem(state, payload) {
    state.items.push(payload);

    return state;
  },
  clearItem(state, payload) {
    state.items.splice(payload.index, 1);

    return state;
  }
};
```

Like the actions, these mutations are minimal. In my opinion, your mutations should always be simple because they have one job: mutate the store's state. As a result, these examples are as complex as they should ever be. Any proper logic should happen in your `actions`. As you can see for this system, we return the new version of the state so that the `` Store`'s <code>commit `` method can do its magic and update everything. With that, the main elements of the store system are in place. Let's glue them together with an index file.

In the same directory, create a new file called `index.js`. Open it up and add the following:

```js
import actions from './actions.js';
import mutations from './mutations.js';
import state from './state.js';
import Store from './store.js';

export default new Store({
  actions,
  mutations,
  state
});
```

All this file is doing is importing all of our store pieces and glueing them all together as one succinct `Store` instance. Job done!

The final piece of the puzzle

The last thing we need to put together is the `main.js` file that we included in our `index.html` page *waaaay* up at the start of this tutorial. Once we get this sorted, we'll be able to fire up our browsers and enjoy our hard work! Create a new file called `main.js` at the root of your `js` directory. This is how it looks for me:

```
~/Documents/Projects/vanilla-js-state-management-boilerplate/src/js/main.js
```

Open it up and add the following:

```js
import store from './store/index.js';

import Count from './components/count.js';
import List from './components/list.js';
import Status from './components/status.js';

const formElement = document.querySelector('.js-form');
const inputElement = document.querySelector('#new-item-field');
```

So far, all we're doing is pulling in dependencies that we need. We've got our `Store`, our front-end components and a couple of DOM elements to work with. Let's add this next bit to make the form interactive, straight under that code:

```js
formElement.addEventListener('submit', evt => {
  evt.preventDefault();

  let value = inputElement.value.trim();

  if(value.length) {
    store.dispatch('addItem', value);
    inputElement.value = '';
    inputElement.focus();
  }
});
```

What we're doing here is adding an event listener to the form and preventing it from submitting. We then grab the value of the textbox and trim any whitespace off it. We do this because we want to check if there's actually any content to pass to the store next. Finally, if there's content, we dispatch our `addItem` action with that content and let our shiny new `store` deal with it for us.

Let's add some more code to `main.js`. Under the event listener, add the following:

```js
const countInstance = new Count();
const listInstance = new List();
const statusInstance = new Status();

countInstance.render();
listInstance.render();
statusInstance.render();
```

All we're doing here is creating new instances of our components and calling each of their `render` methods so that we get our initial state on the page.

With that final addition, we are done!

Open up your browser, refresh and bask in the glory of your new state managed app. Go ahead and add something like "*Finished this awesome tutorial*" in there. Pretty neat, huh?

Next steps

There's a lot of stuff you could do with this little system that we've put together. Here are some ideas for taking it further on your own:

- You could implement some local storage to maintain state, even when you reload
- You could pull out the front-end of this and have a little state system for your projects
- You could continue to develop the front-end of this app and make it look awesome. (I'd be really interested to see your work, so please share!)
- You could work with some remote data and maybe even an API
- You could take what you've learned about `Proxy` and the Pub/Sub pattern and develop those transferable skills further

Wrapping up

Thanks for learning about how these state systems work with me. The big, popular ones are much more complex and smarter that what we've done — but it's still useful to get an idea of how these systems work and unravel the mystery behind them. It's also useful to learn how powerful JavaScript can be with no frameworks whatsoever.

If you want a finished version of this little system, check out this GitHub repository. You can also see a demo here.

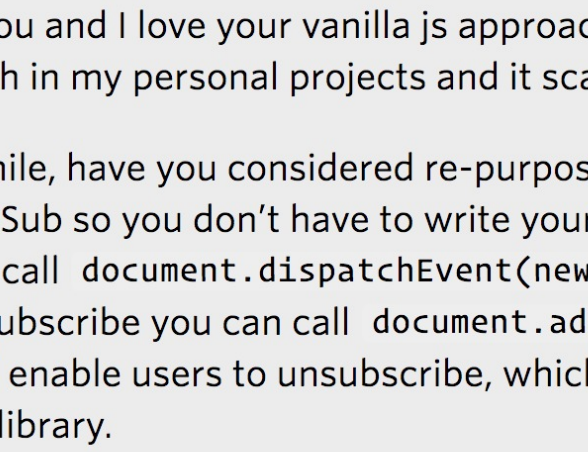
If you develop on this further, I'd love to see it, so hit me up on Twitter or post in the comments below if you do!

Share: Facebook Twitter

Related

When Does a Project Need React?

You know when a project needs HTML and CSS, because it's all of them. When you reach for



Learning Gutenberg: Series Introduction

Hey CSS-Tricksters! 👋 We have a special long-form series we're kicking off here totally dedicated

2017/2018 JavaScript

There has been a lot of research on the landscape this year! Here are a few snippets from a bunch

Comments

Chuanqi Sun (@chuanqi_sun)

JULY 25, 2018

Reply ↓

Thank you and I love your vanilla js approach! I've used a similar approach in my personal projects and it scaled pretty well!

Meanwhile, have you considered re-purposing the DOM event API as a PubSub so you don't have to write your own library? To publish, you can call `document.dispatchEvent(new CustomEvent('xxx'))` and to subscribe you can call `document.addEventListener(...)`. This will enable users to unsubscribe, which isn't part of your PubSub library.

Reference: https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Creating_and_triggering_events

Andy Bell

JULY 25, 2018

That would work for sure. Heydon Pickering has done something like that on his *tiny* and very smart approach.

Oren Mizrahi (@orenmizr)

JULY 25, 2018

Reply ↓

very interesting read. i am a big fan of vuex and state stores in general. kudos andy. i hope there will more javascript content like this.

Andy Bell

JULY 26, 2018

Thanks Oren!

Qamar Ali

JULY 27, 2018

Reply ↓

Hey! nice approach achieving state management system. One thing which is confusing me is PubSub. Isn't pubsub is same as observer pattern, as the observer pattern is also publish/subscriber pattern, or it is different.

Andy Bell

JULY 30, 2018

Pub/Sub is short for publish/subscribe so you're bang on the money :)

Pensuke

JULY 30, 2018

Actually, there is a minor difference between pub/sub and observer: in the observer pattern the publisher and subscriber are aware of each other and communicate directly. In the pub/sub pattern, the publisher and subscriber communicate indirectly via an event bus, decoupling the two entities.

Yunseop

AUGUST 2, 2018

Reply ↓

Thank you for your posting. If possible, I would like to translate this article and share it with Korean developers, can I translate it?

Geoff Graham

AUGUST 2, 2018

For sure, use anything you'd like. :)

The CSS-Tricks License

SUPER IMPORTANT LEGAL DOCUMENT ———————— I don't give two hoots what you do with any of the design or code you find here. Actually, I do. I hope you take it and use it, uncredited, on a super commercial website and get wicked rich off it. I hope you use it at work and your … Continue reading

CSS-Tricks

0

Andy Bell

AUGUST 2, 2018

Awesome! Thanks :)