Build a state management system with vanilla JavaScript

BY **ANDY BELL** ON JULY 25, 2018
⌨ JAVASCRIPT, JAVASCRIPTS

Managing state is not a new thing in software, but it's still relatively new for building software in JavaScript. Traditionally, we'd keep state within the DOM itself or even assign it to a global object in the window. Now, though we've spoiled with choices for libraries and frameworks to help us with this. Libraries like Redux, MobX and Vuex make managing cross-component state almost trivial. This is great for an application's resilience and it works really well with a state-first, reactive framework such as React or Vue.

How do these libraries work though? What would it take to write one ourselves? Turns out, it's pretty straightforward and there's an opportunity to learn some really common patterns and also learn about some useful modern APIs that are available to us.

Before we get started, it's recommended that you have an intermediary knowledge of JavaScript. You should know about data types and ideally, you should have a grasp of some more modern ES6+ JavaScript features. If not, we've got your back. It's also worth noting that I'm not saying that you should replace Redux or MobX with this. We're working on a little project to skill-up together and, hey, it could definitely power a small application if you were keeping an eye on the size of your JavaScript payload.

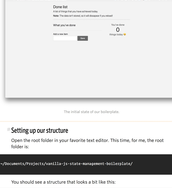
Getting started

Before we dive into code, take a look at what we're building. It's a "done list" that adds up the things you've achieved today. It's inspired by a component of the UI Lab magic — an all-in-one framework dependencies. That's not the real magic though. Behind the scenes, we've got a little state system that's sitting, waiting for instructions and maintaining a single source of truth in a predictable fashion.

VIEW DEMO VIEW REPO

Pretty cool, right? Let's do some admin first. I've put together a bit of a boilerplate so we can keep this tutorial snappy. The first thing you need to do is either clone it from GitHub, or download a ZIP archive and expand it.

Now that you've got that going, going to need to run it in a local web server. I like to use a package called http-serve for these sort of things, but you can use whatever you want. When you've got it running locally, you should see something that looks like this:



The initial state of our boilerplate.

Setting up our structure

Open the root folder in your favorite text editor. This time, for me, the root folder is:

```
~/Documents/Projects/vanilla-js-state-management-boilerplate/
```

You should see a structure that looks a bit like this:

```
./js
├── actions
├── attributes
├── LICENSE
└── README.md
```

Pub/Sub

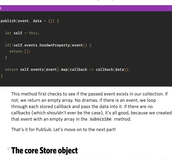
Next, open up the `.js` folder and then open up the `.js` folder that lives in there. Make a new folder called `lib`. Inside that, make a new file called `pubsub.js`.

The structure of your `.js` directory should look like this:

```
./js
└── lib
    └── pubsub.js
```

Open up `pubsub.js` because we're going to make a little Pub/Sub pattern, which is short for "Publish/Subscribe." We're creating the functionality that allows other parts of our application to subscribe to named events. Another part of the application can then publish those events, often with some sort of relevant payload.

Pub/Sub is sometimes hard to grasp, so how about an analogy? Imagine you work in a restaurant and your customers have a starter and a main course. If you've ever worked in a kitchen, you'd know that when the server clears the starters, they let the chefs know which table's starters are cleared. This is a cue to start on the main courses for that table. In a big kitchen, there are a lot of tables and a lot of starters being cleared at the same time. Imagine the cue from the server that the customers have finished their starters, so they know to do their function, which is to prepare the main course. So, you've got multiple chefs waiting on the same cue (named event) to do different functions (callback) to each other.



Hopefully thinking of it like that helps it make sense. Let's move on!

The Pub/Sub pattern loops through all of the subscriptions and fires their callbacks with that payload. It's a great way of creating a pretty elegant reactive flow for your app and we can do it with only a few lines of code.

Add the following to `pubsub.js`:

```
export class PubSub {
  constructor() {
    this.events = {};
  }
}
```

What we've got there is a fresh new class and we're setting `this.events` as a blank object by default. The `this.events` object will hold our named events.

After the constructor's closing bracket, add the following:

```
subscribe(event, callback) {

  let self = this;

  if(!self.events.hasOwnProperty(event
    self.events[event] = {};
  }

  return self.events[event].push(callback
}
```

This is our subscriber method. You pass a string `event`, which is the event's unique name and a callback function. If there's not already a matching event in our `events` collection, we create it with a blank array so we don't have to type check it later. Then, we push the callback into that collection. If it already existed, this is all the method would do. We return the length of the events collection, because it might be handy for someone to know how many events exist.

Now that we've got our subscriber method, guess what comes next? You know it's the `publish` method. Add the following after your subscribe method:

```
publish(event, data = {}) {

  let self = this;

  if(!self.events.hasOwnProperty(event
    return [];
  }

  return self.events[event].map(callback
}
```

This method first checks to see if the passed event exists in our collection. If not, we return an empty array. No drama. If there is an event, we loop through each stored callback and pass the data into it. If there are no callbacks (which shouldn't ever be the case), it's all good, because we created that event with an empty array in the `subscribe` method.

That's it for Pub/Sub. Let's move on to the next part!

The core Store object

Now that we've got our Pub/Sub module, we've got our only dependency for the main**ritters of this little application: the Store. We'll go ahead and start fleshing that out now.

Let's first outline what this does.

The Store is our central object. Each time you see `@Event` inside `Store/_CONTAINER.js`, you'll be pulling in the object that we're going to write. It'll contain a `state` object that, in turn, contains our application state, a `commit` method that will call our **mutations**, and lastly, a `dispatch` function that will call our **actions**. Amongst this and inside the `Store` object, there will be a Proxy-based system that will monitor and broadcast state changes with our `PubSub` module.

Start off by creating a new directory in your `.js` directory called `store`. In there, create a new file called `store.js`. Your `.js` directory should now look like this:

```
./js
└── lib
    ├── pubsub.js
    └── store
        └── store.js
```

Open up `store.js` and import our Pub/Sub module. To do that, add the following right at the top of the file:

```
import PubSub from '../lib/pubsub.js';
```

For those who work with ES6 regularly, this will be very recognizable. Running this sort of code without a bundler will probably be less recognizable though. There's a heck of a lot of support already for this approach, too!

Next, let's start building out our object. Straight after the import, add the following to `store.js`:

```
export default class Store {
  constructor(params) {
    let self = this;
  }
}
```

This is all pretty self-explanatory, so let's add the next bit. We're going to add default objects for `state`, `actions`, and `mutations`. We're also adding a `status` element that we'll use to determine what the object is doing at any given time. This goes right after `let self = this;`:

```
self.actions = {};
self.mutations = {};
self.state = {};
self.status = 'resting';
```

Straight after that, we'll create a new `PubSub` instance that will be attached the `Store` as an `events` element.

```
self.events = new PubSub();
```

Next, we're going to search the passed `params` object to see if any `actions` or `mutations` were passed in. When the `Store` object is instantiated, we can pass in an object of data. Included in that can be a collection of `actions` and `mutations` that control the flow of data in our store. The following code comes next right after the last line of code:

```
if(params.hasOwnProperty('actions')) {
  self.actions = params.actions;
}

if(params.hasOwnProperty('mutations')) {
  self.mutations = params.mutations;
}
```

That's all of our defaults set and nearly all of our potential params set. Let's take a look at how our `Store` object keeps track of all of the changes. We're going to use a Proxy to do this. What the Proxy does is essentially work on behalf of our state object. If we add a `get` trap, we can monitor every time a `get` request is made. With a `set` trap, we can do the same. We can then pass on changes that are made to the object. This is the main part we're interested in today. Add the following straight after the last lines that you added and we'll discuss what it's doing:

```
self.state = new Proxy(params.state || {}, {
  set: (target, key, value) {

    state.key = value;

    console.log(`Setting ${key} to ${value}`);

    self.events.publish('stateChange', self.state);

    if(self.status === 'mutation') {
      console.warn('You should use a mutation to set ${key}');
    }

    self.status = 'resting';

    return true;
  },
});
```

What's happening here is we're trapping the state object `set` operations. That means that when a mutation runs something like `state.name = 'Fido'`, this trap catches it before it can be set and provides us an opportunity to do our work. We're outputting information about what's happening, making sure we're setting the change and then logging the change to the console. We're then publishing a `stateChange` event with our `PubSub` module. Anything subscribed to that event's callback will be called. Lastly, we're checking the status of `Store`. If it's not currently running a `mutation`, it probably means that the state was updated manually. We add a little warning in the console for that to give the developer a little telling off.

There's a lot going on there, but I hope you're starting to see how this is all coming together and importantly, how we're able to maintain state centrally, thanks to Proxy and Pub/Sub.

Dispatch and commit

Now that we've added our core elements of the `Store`, let's add two methods. One that will call our `actions` named `dispatch` and another that will call our `mutations` called `commit`. Let's start with `dispatch` by adding this method after your `constructor` in `store.js`:

```
dispatch(actionKey, payload) {

  let self = this;

  if(typeof self.actions[actionKey] === 'function') {
    console.error('Action "${actionKey}" doesn't exist.');
    return false;
  }

  console.group(`Action triggered: "${actionKey}"`);

  self.status = 'action';

  self.actions[actionKey](self, payload);

  console.groupEnd();

  return true;
}
```

The process here is look for an action and, if it exists, set a status and call the action while creating a logging group that keeps all our logs nice and neat. Anything that is logged (like a mutation or Proxy log) will be kept in the group that we define. If no action is set, it'll log an error and fail. That was pretty straightforward, and the `commit` method is even more straightforward.

Add this after your `dispatch` method:

```
commit(mutationKey, payload) {

  let self = this;

  if(typeof self.mutations[mutationKey] === 'function') {
    console.log(`Mutation "${mutationKey}" doesn't exist`);
    return false;
  }

  self.status = 'mutation';

  let newState = self.mutations[mutationKey](self.state, payload);

  self.state = Object.assign(self.state, newState);

  return true;
}
```

This method is pretty similar, but let's run through the process anyway. If the mutation can be found, we run it and get our new state from its return value. We then merge that into our current state and merge it with our existing state to create an up-to-date version of our state.

With those methods added, our `Store` object is pretty much complete. You could actually modularize this application now if you wanted because we've included most of the bits that we need. You could also add some tools to check that everything run as expected. But I'm not going to leave you hanging like that. Let's make it all actually do what we set out to do and continue with our little app!