Javascript (Https://Www.Sitepoint.Com/Javascript/) > September 18, 2017
> By Michael Wanyoike (https://www.sitepoint.com/author/mwanyoike/)

Redux vs MobX: Which Is Best for Your Project?

For a lot of JavaScript developers, the biggest complaint with Redux is the amount of boilerplate code needed to implement features. A better alternative is MobX which provides similar functionality but with lesser code to write.

For MobX newbies, take a quick look at this introduction (https://medium.com/@mweststrate/becoming-fully-reactive-an-in-depth-explanation-of-mobservable-55995262a254) written by MobX's creator. You can also work through this tutorial (https://www.sitepoint.com/manage-javascript-application-state-mobx/) to gain some practical experience.

The goal of this article is to help JavaScript developers decide which of these two state management solutions are best for their projects. I've migrated this CRUD Redux project (https://www.sitepoint.com/crud-app-react-redux-feathersjs/) to MobX (https://www.github.com/brandiqa/mobx-crud-example) to use as an example in this article. I'll first discuss the pros and cons of using MobX, and then I'll demonstrate actual code samples from both versions to show the difference.

The code for the projects mentioned in this article can be found on GitHub:

- Redux CRUD example (https://github.com/sitepoint-editors/redux-crud-example)
- MobX CRUD example (https://github.com/sitepoint-editors/mobx-crud-example)

If you enjoy this post, you might also like to sign up for SitePoint Premium and watch our course on working with forms using React and Redux (https://www.sitepoint.com/premium/courses/forms-with-react-and-redux-2935).

What Do Redux and MobX Have in Common?

First, let's look at what they both have in common. They:

- are open-source libraries
- provide client-side state management
- support time-travel debugging via the redux-devtools-extension (https://github.com/zalmoxisus/redux-devtools-extension)
- are not tied to a specific framework
- have extensive support for React/React Native frameworks.

4 Reasons to Use MobX

Let's now look at the main differences between Redux and MobX.

1. Easy to learn and use

For a beginner, you can learn how to use MobX in just 30 minutes. Once you learn the basics, that's it. You don't need to learn anything new. With Redux, the basics are easy too. However, once you start building more complex applications, you'll have to deal with:

- handling async actions with redux-thunk (https://github.com/gaearon/redux-thunk)
- simplifying your code with redux-saga (https://github.com/redux-saga/redux-saga)
- defining selectors to handle computed values, etc.

With MobX, all these situations are "magically" taken care of. You don't need additional libraries to handle such situations.

2. Less code to write

To implement a feature in Redux, you need to update at least four artifacts. This includes writing code for reducers, actions, containers and components. This is particularly annoying if you're working on a small project. MobX only requires you to update at least two artifacts (i.e. the store and the view component).

3. Full support for object-oriented programming

If you prefer writing object-oriented code, you'll be pleased to know you can use OOP to implement state management logic with MobX. Through the use of decorators such as `@observable` and `@observer`, you can easily make your plain JavaScript components and stores reactive. If you prefer functional programming, no problem — that's supported as well. Redux, on the other hand, is heavily geared towards functional programming principles. However, you can use the redux-connect-decorator (https://github.com/evturn/redux-connect-decorator) library if you want a class-based approach.

4. Dealing with nested data is easy

In most JavaScript applications, you'll find yourself working with relational or nested data. To be able to use it in a Redux store, you'll have to normalize (http://redux.js.org/docs/recipes/reducers/NormalizingStateShape.html) it first. Next, you have to write some more code (http://redux.js.org/docs/recipes/reducers/UpdatingNormalizedData.html) to manage tracking of references in normalized data.

In MobX, it's recommended (https://mobx.js.org/best/store.html) to store your data in a denormalized form. MobX can keep track of the relations for you, and will automatically re-render changes. By using domain objects to store your data, you can refer directly to other domain objects defined in other stores. In addition, you can use (@)computed decorators (https://mobx.js.org/refguide/computed-decorator.html) and modifiers for observables (https://mobx.js.org/refguide/modifiers.html) to easily solve complex data challenges.

3 Reasons Not to Use MobX

1. Too much freedom

Redux is a framework that provides strict guidelines on how you write state code. This means you can easily write tests and develop maintainable code. MobX is a library and has no rules on how to implement it. The danger with this is that it's very easy to take shortcuts and apply quick fixes that can lead to unmaintainable code.

2. Hard to debug

MobX's internal code "magically" handles a lot of logic to make your application reactive. There's an invisible area where your data passes between the store and your component, which makes it difficult to debug when you have a problem. If you change state directly in components, without using `@actions`, you'll have a hard time pinpointing the source of a bug.

3. There could be a better alternative to MobX

In software development, new emerging trends appear all the time. Within a few short years, current software techniques can quickly loose momentum. At the moment, there are several solutions competing with both Redux and Mobx. A few examples are Relay/Apollo & GraphQL (https://www.sitepoint.com/react-data-fetching-with-relay/), Alt.js (http://alt.js.org/) and Jumpsuit (https://jumpsuit.js.org/). Any of these technologies has the potential to become the most popular. If you really want to know which one is best for you, you'll have to try them all.

Code Comparison: Redux vs MobX

Enough theory, let's look at the code. First, we compare how each version does bootstrapping.

Bootstrapping

Redux Version:

In Redux, we first define our store and then we pass it to `App` via `Provider`. We'll also need to define `redux-thunk` and `redux-promise-middleware` to handle asynchronous functions. The `redux-devtools-extension` allows us to debug our store in time-traveling mode.

```
// src/store.js
import { applyMiddleware, createStore } from "redux";
import thunk from "redux-thunk";
import promise from "redux-promise-middleware";
import { composeWithDevTools } from 'redux-devtools-extension';
import rootReducer from "./reducers";

const middleware = composeWithDevTools(applyMiddleware(promise(), thunk));

export default createStore(rootReducer, middleware);

-------------------------------------------------------------------------------

// src/index.js
…
ReactDOM.render(
  <BrowserRouter>
    <Provider store={store}>
      <App />
    </Provider>
  </BrowserRouter>,
  document.getElementById('root')
);
```

MobX Version:

In MobX, we need to setup multiple stores. In this case, I'm using only one store, which I've placed in a collection named `allStores`. A `Provider` is then used to pass the stores collection to the `App`.

As mentioned earlier, MobX doesn't need external libraries to handle async actions, hence the fewer lines. However, we do need the `mobx-remotedev` to connect to the `redux-devtools-extension` debugging tool.

```
// src/stores/index.js
import remotedev from 'mobx-remotedev';
import Store from './store';

const contactConfig = {
  name:'Contact Store',
  global: true,
  onlyActions:true,
  filters: {
    whitelist: /fetch|update|create|Event|entity|entities|handleErrors/
  }
};

const contactStore = new Store('api/contacts');

const allStores = {
  contactStore: remotedev(contactStore, contactConfig)
};

export default allStores;

-------------------------------------------------------------------------------


// src/index.js
…
ReactDOM.render(
  <BrowserRouter>
    <Provider stores={allStores}>
      <App />
    </Provider>
  </BrowserRouter>,
  document.getElementById('root')
);
```

The amount of code here is roughly about the same in both versions. MobX has fewer import statements though.

Props injection

Redux Version:

In Redux, state and actions are passed to props using react-redux's `connect()` function.

```
// src/pages/contact-form-page.js
…
  // accessing props
  <ContactForm
    contact={this.props.contact}
    loading={this.props.loading}
    onSubmit={this.submit}
  />
…

// function for injecting state into props
function mapStateToProps(state) {
  return {
    contact: state.contactStore.contact,
    errors: state.contactStore.errors
  }
}

// injecting both state and actions into props
export default connect(mapStateToProps, { newContact,
  saveContact,
  fetchContact,
  updateContact
})(ContactFormPage);
```

MobX Version:

In MobX, we simply inject the `stores` collection. We use `@inject` at the top of a container or component class to do this. This makes `stores` available in `props`, which in turn allows us to access a specific store and pass it to a child component. Both state and actions are accessed via properties in the `store` object hence no need to pass them separately as with the case in Redux.

```
// src/pages/contact-form-page.js

…
@inject("stores") @observer // injecting store into props
class ContactFormPage extends Component {
…
  // accessing store via props
  const { contactStore:store } = this.props.stores;
  return (
      <ContactForm
        store={store}
        form={this.form}
        contact={store.entity}
      />
  )
…
}
```

The MobX version seems to be easier to read. However, we can use redux-connect-decorators (https://github.com/evturn/redux-connect-decorator) to simplify Redux code. In that case, there'll be no clear winner.

Defining stores, actions, and reducers

To keep this article lean, I'll show you a code sample for just one action.

Redux Version:

In Redux, we need to define actions and reducers.

```
// src/actions/contact-actions.js
…
export function fetchContacts(){
  return dispatch => {
    dispatch({
      type: 'FETCH_CONTACTS',
      payload: client.get(url)
    })
  }
}
…

// src/reducers/contact-reducer
…
switch (action.type) {
    case 'FETCH_CONTACTS_FULFILLED': {
      return {
        ...state,
        contacts: action.payload.data.data || action.payload.data,
        loading: false,
        errors: {}
      }
    }

    case 'FETCH_CONTACTS_PENDING': {
      return {
        ...state,
        loading: true,
        errors: {}
      }
    }

    case 'FETCH_CONTACTS_REJECTED': {
      return {
        ...state,
        loading: false,
        errors: { global: action.payload.message }
      }
    }
}
…
```

MobX Version:

In MobX, the logic for the action and the reducer is done in one class. I've defined an async action that calls another action `entities fetched` after `response` has been received.

Since MobX uses the OOP style, the `Store` class defined here has been refactored to allow easy creation of multiple stores using the class constructor. Hence the code demonstrated here is base code that's not tied to a particular domain store.

```
// src/stores/store.js
…
@action
fetchAll = async() => {
  this.loading = true;
  this.errors = {};
  try {
    const response = await this.service.find({})
    runInAction('entities fetched', () => {
      this.entities = response.data;
      this.loading = false;
    });
  } catch(err) {
      this.handleErrors(err);
  }
}
…
```

Believe it or not, the logic defined in both versions do the same tasks, which are:

- update the UI loading state
- fetch data asynchronously
- catch exceptions and update state.

In Redux, we've used **33 lines of code**. In MobX, we've used about **14 lines of code** to achieve the same result! A major benefit of the MobX version is that you can reuse the base code in almost all the domain store classes with little or no modification.

That means you can build your application faster.

Other differences

To create forms in Redux, I've used redux-form (https://www.npmjs.com/package/redux-form). In MobX, I've used mobx-react-form (https://www.npmjs.com/package/mobx-react-form). Both libraries are mature and help you handle form logic easily. Personally, I prefer `mobx-react-form`, since it allows you to validate fields via plugins. With `redux-form`, you either write your own validation code or you can import a validation package to handle validation for you.

One tiny downside with MobX is that you can't directly access certain functions in observable objects since they are not really plain JavaScript objects. Luckily, they have provided the function `toJS()` which you can use to convert observable objects to plain JavaScript objects.

Recommended Courses

Conclusion

Clearly, you can see that MobX's code base is far much leaner. Using OOP style and good development practices, you can rapidly build applications. The major downside is that it's very easy to write poor, unmaintainable code.

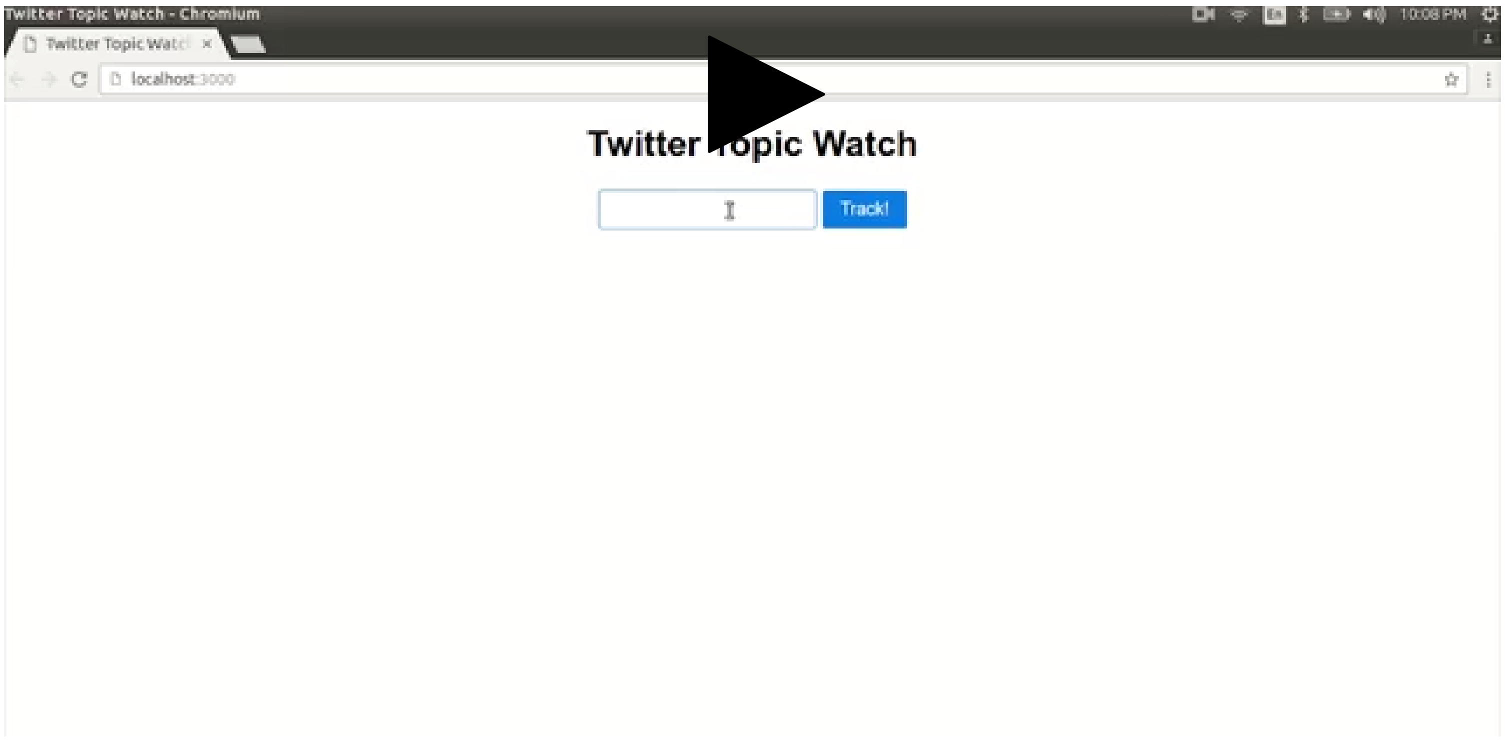
Redux, on the other hand, is more popular and well suited for building large and complex projects (https://www.sitepoint.com/premium/courses/redux-design-issues-and-testing-2962). It's a strict framework with safeguards ensuring every developer writes code that's easy to test and maintain. However, it's not well suited to small projects.

Despite MobX's drawbacks, you can still build large projects if you follow good practices. In the words of Albert Einstein, "Make everything simple as possible, but not simpler".

I hope I've provided enough information to make a clear case whether to migrate to MobX or stick with Redux. Ultimately, the decision depends on the type of project you're working on, and the resources available to you.

This article was peer reviewed by [Dominic Myers (https://github.com/annoyingmouse)](https://github.com/annoyingmouse) *and* [Vildan Softic (https://www.sitepoint.com/author/vildansoftic/)](https://www.sitepoint.com/author/vildansoftic/)*. Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!*

If you're looking to up your Redux game, sign up for SitePoint Premium and enroll in our course** [Redux Design Issues and Testing (https://www.sitepoint.com/premium/courses/redux-design-issues-and-testing-2962)](https://www.sitepoint.com/premium/courses/redux-design-issues-and-testing-2962)**. In this course, you'll build a Redux application that receives tweets, organized by topic, through a websocket connection. To give you a taster of what's in store, check out the free lesson below.



(https://www.sitepoint.com/)

OFFER ENDS IN **21** HRS **15** MINS **39** SECS

Start Learning For $9 (/Premium/L/Join?Ref_source=Sitepoint&Ref_medium=Noticebar&Ref_campaign=Redux-Vs-Mobx-Which-Is-Best&Ref_content=Javascript)

(htt

Meet the author

Michael Wanyoike (https://www.sitepoint.com/author/mwanyoike/) (https://twitter.com/myxsys) (https://www.linkedin.com/in/mikewanyoike/) (https://github.com/brandiqa)

I write clean, readable and modular code. I love learning new technologies that bring efficiencies and increased productivity to my workflow. Currently am into React & Javascript stuff.

Stuff We Do

- Premium (/premium/)
- Versioning (/versioning/)
- Themes (/themes/)
- Forums (/community/)
- References (/html-css/css/)

About

- Our Story (/about-us/)
- Press Room (/press/)

Contact

- Contact Us (/contact-us/)
- FAQ (https://sitepoint.zendesk.com/hc/en-us)
- Write for Us (/write-for-us/)
- Advertise (/advertise/)

Legals

- Terms of Use (/legals/)
- Privacy Policy (/legals/#privacy)

Connect