Build your first app with Mobx and React

Rosy Shrestha Follow
Jan 30 · 7 min read

MobX is another state management library available for React apps besides Redux and Context API. However, MobX is not just a library for React alone, it is also suitable for use with other JavaScript libraries and frameworks that power the frontend of web apps. The MobX >=5 version runs on any browser that supports ES6 proxy.

Major concepts

Here are the major concepts of mobx:

Observable

The observable allows us to turn any data structure or property into an observable state so that other things can keep track of these observable changes and values.

Action

The action allows us to change the state i.e. values of observable. The state data should not be modified outside actions to ensure code scalability.

Computed

The computed property allows us to derive some value based on the state change. The computed values are obtained by performing some sort of calculations on observables.

Observer

The observer allows us to keep track of changes in observables so that React gets notified on any change and starts re-rendering. They are provided by the mobx-react package.

Store

The store is the source of data. Unlike redux, in mobx data and functions which change the data live in the store. So a store may contain observables and actions.

> Very in a nutshell mobx works on the observable — observer flow. You declare some data to be observable and when that data changes all the observers that are using that data will be notified.

Now let's put these concepts into practice.

We are going to create a simple application where users can react to images and comment on it, similar to Facebook. Here's the link to the demo.

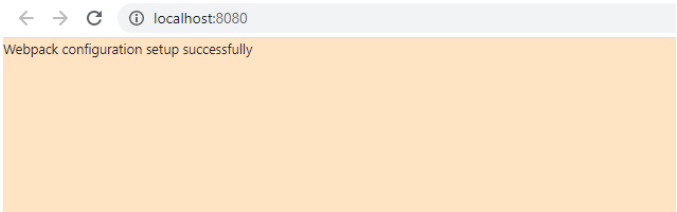
Project setup

Assuming prior knowledge of React, you need to have NPM and Node.js installed on your machine.

I am using custom webpack configuration and setting up the project to enable decorators. Don't worry, there's an alternative way to do this without decorators as well. For this example, I'm using decorators anyway since it's more concise. But I'll mention the alternatives as well. If you're using create-react-app you can skip these setup steps.

Pull the `master` branch from this repository for initial setup.

Run `yarn` to install dependencies and start the project using `yarn start`. The app will run on http://localhost:8080.



Setup for decorators

The following plugins are required to enable ESNext decorators.

```
yarn add --dev @babel/plugin-proposal-class-properties @babel/plugin-proposal-decorators
```

Then add the following configuration to `.babelrc` file.

```
"plugins": [
        [
            "@babel/plugin-proposal-decorators",
            {
                "legacy": true
            }
        ],
```

```
        [
            "@babel/plugin-proposal-class-properties",
            {
                "loose": true
            }
        ]
    ]
```

.babelrc hosted with ❤ by GitHub view raw

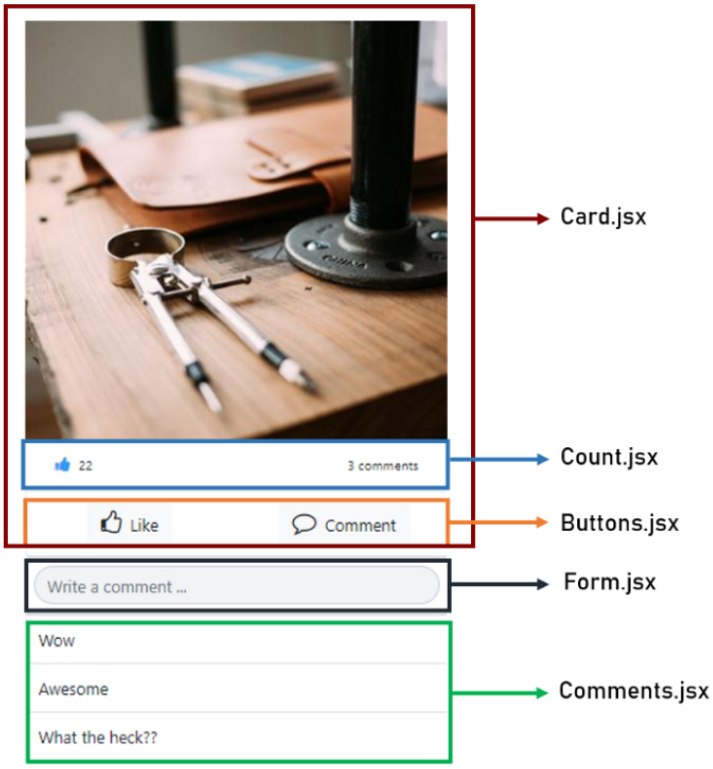
Styles

Pull design branch for styles. All the styling is under the `css` folder inside the `src` directory. Here's a visual of our app components.

- **Card component** with:

1. Randomly generated image.
2. **Count** component to keep track of the number of likes and comments.
3. **Button** component with Like and Comment buttons.

- **Form component** with the input field to post a comment.
- **Comments component** with a list of comments.



Installing Dependencies

Install `mobx` state management solution and `mobx-react` library to connect the state layer to the React view layer.

```
yarn add mobx mobx-react
```

Now we will actually start adding features using Mobx.

. . .

Store

First, we're going to create a `Store.jsx` under `store` folder.

```
import { observable, action } from 'mobx'

class Store {
    @observable likesCount = 12

    @action updateCount{
        this.likesCount++;
    }
}

const storeInstance = new Store()
export default storeInstance;
```

Store.jsx hosted with ❤ by GitHub view raw

Here we've created a Store class with `likesCount` as an observable state, `updateCount` as an action to modify the state and then exported a new instance of the Store.

If your setup doesn't support decorators the above code can be re-written as:

```
import { decorate, observable } from "mobx";

class Store {
    likesCount = 12;

    updateCount{
        this.likesCount++;
    }
}

decorate(Store, {
    likesCount: observable,
    updateCount: action
})
```

Store.jsx hosted with ❤ by GitHub view raw

Then we make the store accessible throughout the app by passing it using the Context API in `main.js`

```
import storeInstance from './store/Store'

export const StoreContext = React.createContext();

ReactDOM.render(
        <StoreContext.Provider value={storeInstance}>
            <Post />
        </StoreContext.Provider >
    , document.getElementById('app'));
```

main.js hosted with ❤ by GitHub view raw

Now we can access the store and its class properties in `Count.jsx` using `useContext`. Since we've set the initial value of `likesCount` to 12, your app will render that value.

```
import React, { useContext } from 'react';
import { StoreContext } from '../main'

export default function Count() {
    const store = useContext(StoreContext)
    return (
        <div className="row reactions-count" >
            <div className="col-sm" align="left">
```

```
                <i className="fa fa-thumbs-up" />{store.likesCount}
            </div>
            <div className="col-sm" align="right">
                3 comments
        </div>
        </div>
    )
}
```

Count.jsx hosted with ❤ by GitHub view raw

Remember that the observable state can only be modified via actions. So in order to increment `likesCount` when user clicks on Like button we are going to use `updateCount` action from the store that we've already defined. Handle `onClick` action in `Buttons.jsx`.

```
const store = useContext(StoreContext)
<button type="button" className="btn btn-light align-top" onClick={() => store.updateCount()}>
  <i className="fa fa-thumbs-o-up" />
  Like
</button>
```

Buttons.jsx hosted with ❤ by GitHub view raw

If you click on the Like button you won't see any changes.

To observe and react to changes in a functional component, we can either wrap the component in observer function or implement useObserver hook, like below. So let's update `Count.jsx` as:

```
import { useObserver } from 'mobx-react';

...
  return useObserver(() => (
        <div className="row reactions-count" >
            <div className="col-sm" align="left">
                <i className="fa fa-thumbs-up" />{store.likesCount}
            ...
            ...
        </div>
        </div>
    ))
```

Count.jsx hosted with ❤ by GitHub view raw

Now the like count updates when you click on the button.

. . .

Comments

Let's start working on the comments section.

An array data structure can be observable as well. Let's create an observable `comments` field. Add the following in `Store.jsx`.

```
@observable comments = ["Wow", "awesome"]
```

Then access the comments property of Store class from `Comments.jsx` like we did before in `Count.jsx` using `useContext`. The Comments component will now render the comments from the store.

```
import React, { useContext } from 'react';
import { StoreContext } from '../main';
```

```jsx
export default function Comments() {
    const store = useContext(StoreContext)
    return (
        <table className="table">
            <tbody>
                {
                    store.comments.map((comment, index) => {
                        return (
                            <tr key={index}>
                                <td>
                                    {comment}
                                </td>
                            </tr>
                        )

                    })
                }
            </tbody>
        </table>
    )
}
```

Comments.jsx hosted with ❤ by GitHub view raw

We also need to allow the user to add comments from the form.

First, let's create an action called `postComment` in the store that simply pushes the new comment into the previous array of comments. Add the following lines of code in `Store.jsx`.

```
@action postComment(comment){
        this.comments.push(comment)
    }
```

Then update the `Form.jsx` component as:

```jsx
import React, { useContext } from 'react';
import { StoreContext } from '../main';

export default class Form extends React.Component {

    handleSubmit = (e, store) => {
        e.preventDefault();
        store.postComment(this.comment.value);
        this.comment.value = "";
    }

    render() {
        return (
            <StoreContext.Consumer>
                {
                    store => (

                        <form onSubmit={(e) => this.handleSubmit(e, store)}>
                            <div>
                                <input type="text" id={'comment'} className="form-control" place
                                    this.comment = node;
                                }} />
                            </div>
                        </form>
                    )
                }
            </StoreContext.Consumer>
        )
    }
}
```

Form.jsx hosted with ❤ by GitHub view raw

Here we've simply created a function that calls the store's `postComment` action when the user submits the comment and set the input field to empty after submit.

To update the comments component when a new comment is added, we

need to make the Comments component an observer as we did with Count. So in `Comments.jsx` wrap the content to be returned with `useObserver`. Also, don't forget to import `useObserver`.

```
return useObserver(() => (
        <table className="table">
            <tbody>
                {
                    store.comments.map((comment, index) => {
                       ...
                       ...
                }
            </tbody>
        </table>
    )
    )
```

Comments.jsx hosted with ❤ by GitHub view raw

Now if you write any comment and hit enter your list of comments will automatically update.

Let's focus on the input field when you click on the comment button. We can simply use HTML DOM focus( ) method. But first, let's give the input field an id.

```
<input type="text" id={'comment'} className="form-control" placeholder={"Write a comment ..."}
ref={node => {this.comment = node;}} />
```

Input.jsx hosted with ❤ by GitHub view raw

Then add focus method on `onClick` handler of comment button in `Buttons.jsx` component.

```
<button type="button" className="btn btn-light" onClick={() => document.getElementById('comment')
  <i className="fa fa-comment-o" />
  Comment
</button>
```

Buttons.jsx hosted with ❤ by GitHub view raw

Now when you click on the comment button the comment field is focused.

Computed

Now in order to get the count of the comments, we are going to create a `commentsCount` getter function that computes the observable `comments` array's length. MobX will ensure `commentsCount` updates automatically whenever `comments` array changes. In `Store.jsx` add the following:

```
@computed get commentsCount(){
        return this.comments.length;
    }
```

Then simply update the following lines in `Count.jsx`.

```
<div className="col-sm" align="right">
  {store.commentsCount} comments
</div>
```

You'll also notice that when you add a comment, the count gets updated as well.

. . .

Services / API call

Making an API call and asynchronous codes are frequent in applications. Since this is a custom webpack configuration to enable async/await update the `.babelrc` file with the following.

```
"presets": [
        ["@babel/preset-env",
        {
            "targets": {
              "node": "10"
            }
          }
        ],
        "@babel/preset-react"
    ],
```

.babelrc hosted with ❤ by GitHub view raw

or else you might run into this error

```
Uncaught ReferenceError: regeneratorRuntime is not defined
    at eval (ImageStore.jsx:39)
    at eval (ImageStore.jsx:79)
    at eval (ImageStore.jsx:88)
    at Module../src/store/ImageStore.jsx (app.bundle.js:1809)
    at __webpack_require__ (app.bundle.js:727)
    at fn (app.bundle.js:101)
    at eval (Card.jsx:7)
    at Module../src/components/Card.jsx (app.bundle.js:1726)
    at __webpack_require__ (app.bundle.js:727)
    at fn (app.bundle.js:101)
```

Let's change the image in the `Card.jsx` component on button click. We are going to use this fun and free API to fetch the characters' images from the Rick and Morty show. Check out their documentation for more details.

You'll find from this section that we can get a single character by adding the `id` as a parameter: `/character/1`

```
https://rickandmortyapi.com/api/character/1
```

Let's create an image store with observable `imageUrl` containing default value. Then we create a `fetchImage` action that returns the JSON response of a single character.

After `await` a new asynchronous function is started, so after each `await`, state modifying code should be wrapped as action. There are multiple ways to do this. Read this section of Mobx documentation for more details.

One way is to use the `runInAction`, which is a simple utility that takes a code block and executes in an anonymous action. Here we are wrapping the state modifying part after await in `runInAction`.

```
import { action, runInAction, observable } from "mobx";

class ImageStore {

    id = 1

    @observable imageUrl = `https://rickandmortyapi.com/api/character/avatar/1.jpeg`

    @action async fetchImage() {
```

```
            const characterId = ++this.id
            const response = await fetch(`https://rickandmortyapi.com/api/character/${characterI
            const data = await response.json()
            runInAction(() => {
                this.imageUrl = data.image
            })
    }
}

const imageStore = new ImageStore()

export default imageStore;
```

ImageStore.jsx hosted with ❤ by GitHub view raw

You can also run only the state modifying part of the callback in an action. Here we've created an action to set the URL outside the `fetchImage` and then called it as required.

```
class ImageStore {

    ...

    @action async fetchImage() {
            ...
            this.setImageUrl(data.image)
    }


    @action setImageUrl(url) {
        this.imageUrl = url
    }
}
```

ImageStore.jsx hosted with ❤ by GitHub view raw

Then in `Card.jsx` component

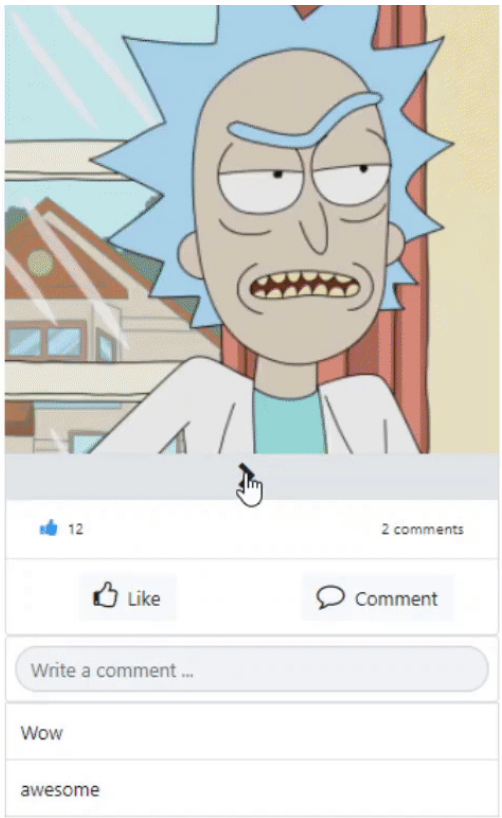
- Import the `imageStore` and set the source of the image to the observable `imageUrl` from the store.
- Implement `useObserver` to react to changes.
- Add a button with `onClick` handler that calls the `fetchImage` to get the image URL.

```
import React from "react";
import Count from "./Count";
import Buttons from "./Buttons";
import imageStore from '../store/ImageStore'
import { useObserver } from "mobx-react";

export default function Card() {
    return (
        useObserver(() => (
            <div className="card">
                <img src={imageStore.imageUrl} className="card-img-top" alt="..." />
                <button className="btn btn-light" onClick={() => { imageStore.fetchImage() }}>
                    <i className="fa fa-chevron-right" />
                </button>
                <Count />
                <div className="card-body" >
                    <Buttons />
                </div>
            </div>
        ))
    );
}
```

Card.jsx hosted with ❤ by GitHub view raw

Aaand we're done! Here's what your final output will look like:



. . .

#Note

The nice thing about bundling actions with stores is that we can use them in `onClick` handlers. Which means most of the components, like in this example, can be stateless functional components. To make a class component an observer we can use `@observer` decorator or wrap the component with `observer` function.

```
import React from "react";
import { observer } from "mobx-react";

//With decorator
@observer
export default class Form extends React.Component{
 ...
}

//Without decorator

class Form extends React.Component{
 ...
}
export default observer(Form )
```

Mobx docs are well-written and contain a lot of best practices.

You can find all the code of this example here —

And that's it. Thanks for reading! 😃

Mobx React Beginner Tutorial Mobx React

217 claps

WRITTEN BY
Rosy Shrestha

Follow

More From Medium

Functional Programming: Using the Functor Type Class in TypeScript
Cedric Dose in Better Programming

Why console.log()?
Patrick Flor in Tildeslash

The tradeoffs of CSS-in-JS
Oleg Isonen in freeCodeCamp.org

How to build a simple Task Manager w/ React & Laravel RESTful API
Magdalena Siljanoska in The Startup

What are Methods & Properties in JavaScript?
Reina Mitchell in JavaScript In Plain English

How to add React Router to your React App
Reinald Reynoso in JavaScript In Plain English

Creating a masonry layout with CSS Grid
Adam Collier

Image Processing in NodeJS with Jimp
Ross Bulat

Discover Medium
Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours
Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member
Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

Medium

About Help Legal