# Summer of Innovation Fridays

Masha Thomas Follow
Sep 21, 2018 · 6 min read

This past summer, our team at The Orchard decided to turn our Fridays into Innovation Fridays giving engineers an opportunity to try out new things and work on projects that aren't necessarily on a roadmap. One of our Engineering teams has been working on a large project to modernize our asset ingestion and transcoding pipeline. This has primarily been a backend project, but on Innovation Fridays, they decided to try out what a new UI workflow might look like with real time status updates and an integration with third parties like Google Drive. In the process, they decided to experiment with something new in the React code. Traditionally, The Orchard's frontend applications are standardized to use React with Redux. However for this Summer of Innovation project, the team chose to check out MobX. Here is a comparison summary of our findings written by Kevin Li, one of our engineers.

## Redux and MobX

These are two state management alternatives for managing application data and state on the frontend. For the most part they both support the same use cases. The main differences are in how data is organized and interacted with.

To lay out our findings we've included screenshots are taken from MobX documentation and Redux documentation.

## State Structure

First, we analyzed State Structure which is a means to understand how data is represented and organized.

## Redux

Redux revolves around functions. These functions, called reducers, are used to define the initial state. The original state is considered the input and the next state is computed. The state is (expected to be) immutable and the next state is always (expected to be) a new object. Reducers are often grouped together to form a single state store.

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([{ text: action.text, completed: false }])
    case 'TOGGLE_TODO':
      return state.map(
        (todo, index) =>
          action.index === index
            ? { text: todo.text, completed: !todo.completed }
            : todo
      )
    default:
      return state
  }
}
```

## MobX

If you are familiar with object-oriented programming you will find MobX to be similar. Data is organized into classes and the state is comprised of instances of these classes. These classes are no different from regular javascript classes. You can mark certain attributes as observable and only when those change will it propagate as a change to observers. There is no strict requirement to keep all of these class instances in one place but a common pattern is to group them together into a single state store.

## Actions

We then moved forward with Actions, making updates to the state.

## Redux

Create an action type and in your reducers define how to handle the action type. To trigger the change, your store can dispatch an action. Actions are objects and the only requirement is they must have a type key. You can include additional information under different key names. When an action is picked up by a reducer, it will check the action type. If the reducer recognizes the type, it will compute a new state and return it. The new state then replaces the current state. Note that multiple reducers can process the same action.

## MobX

You can change the attributes directly on a class instance. You can also have functions that live outside of class definitions. Any function is free to change the state if they have access to the class instances. If you enable strict mode, only functions decorated as an action can change an observable attribute.

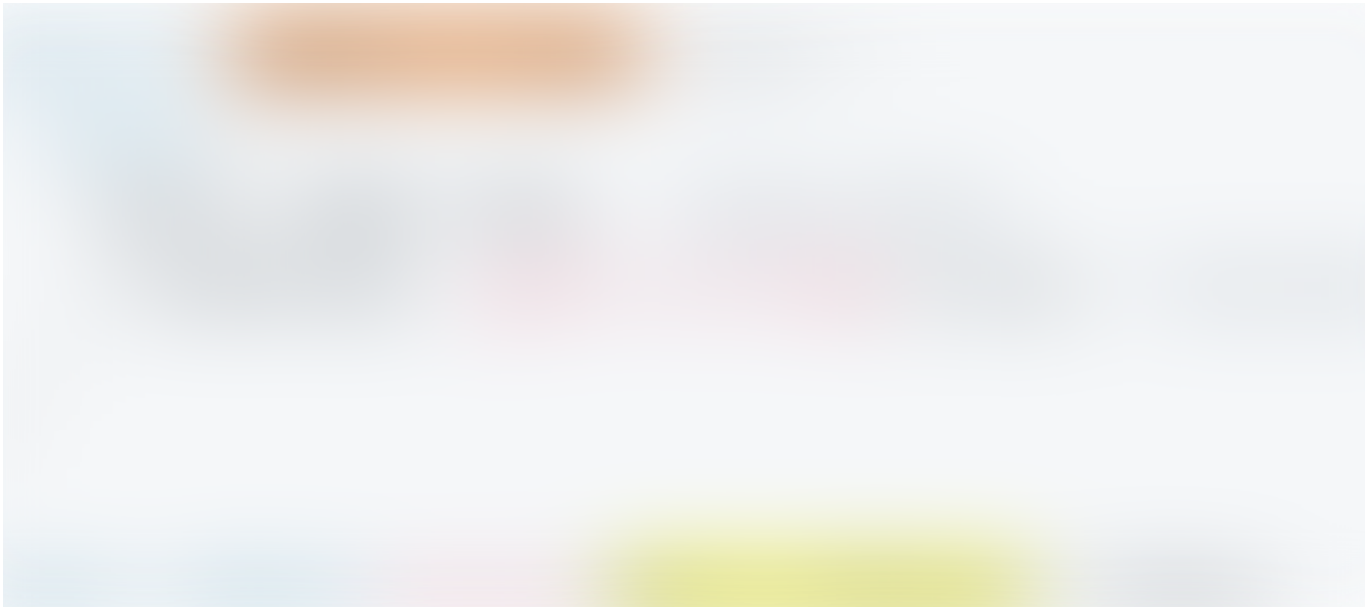


## Listening for Changes

Reacting to state changes in React applications.

## Redux

State data is passed to React components as props. When the state changes, the props will change and prop changes are what triggers a rerender in React. State data that isn't mapped to the prop of a component will not trigger a rerender when it changes.



## MobX

MobX also relies on React's behavior to update when props change. Decorate components that need to respond to state changes as an observer. Then you can either access the state data passed in as props, directly or from a global state (if you have one).

## Reactions

Data that depends on the state (aka derivative states).

## Redux

You have to store derivative information as part of the state. If you have several parts of your state that needs to be updated for a certain action, you need to update each of the corresponding reducers to handle that action.

## MobX

You can decorate a function as computed if the value only changes when the state changes. Whenever any observable value used in your function changes, the computed function will be rerun and the value available the next time the function is accessed will reflect the change. The function will NOT be recomputed if none of the observable values within the function have changed.

## Support

In regard to Support, Redux is more popular (3 times as much going by GitHub stars) so there are more resources to work with. It has a great Chrome extension that gives insight into the state during development.

## Findings

- Redux requires more boilerplate code. When you want to add a new key you have to create the corresponding reducer and actions for interacting with it. In MobX you have to define the class (similar to a reducer) and mark which keys are observable.
- When working with React components you use a mapStateToProps function to link a state to a component. In MobX you decorate a component as an observer and whenever any references to an observable value changes it will trigger a rerender. It

does not matter if it was passed in via props or accessed directly through a global store.

- MobX offers very granular control. Observable values can be a subset of class attributes. Since every class can have instance functions that control its own state, you can just call the function to update the state. For example, if you have a list of to-dos and you wanted to update the third to-do's status you can do todos[2].setStatus('done'). In Redux you would need to replace the entire todos array or pass the index as part of the action payload.
- It is easy to track the change history in Redux because the state is immutable and changes must go through the action / reducer flow. You can clearly see the before and after of each dispatched action. Since there are so many places that can change the state in MobX it can be hard to track state changes.
- It seems that MobX's greatest strength is how easily you can define things that "react" to state changes. These reactions can then be treated as observable values by React components.

## Resources

https://redux.js.org/

https://github.com/mobxjs/mobx

React Innovation Mobx