Mobx - A Simple State Management Library

And also what we have learned during our experiment with it

Graham Goh
graham.goh@myob.com
Team Andromeda

Redux too many things to learn?

Reducers

Immutables

Denormalization

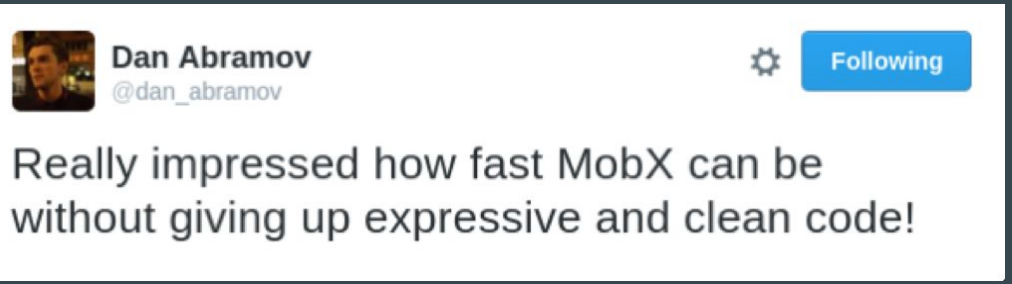
Connect

Selectors

Smart & Dumb components

Thunks & Sagas


Dan Abramov
@dan_abramov
Unhappy with Redux? Try MobX:


Dan Abramov
@dan_abramov
Following
Really impressed how fast MobX can be
without giving up expressive and clean code!

“MobX makes state management simple again”

The Concept

Events invoke actions. Actions are the only thing that modify state and may have other side effects.

```
@action onClick = () => {
  this.props.todo.done = true;
}
```

State is observable and minimally defined. Should not contain redundant or derivable data. Can be a graph, contain classes, arrays, refs, etc.

```
@observable todos = [{
  title: "learn MobX",
  done: false
}]
```

Computed values are values that can be derived from the state using a pure function. Will be updated automatically by MobX and optimized away if not in use.

```
@computed get completedTodos() {
  return this.todos.filter(
    todo => todo.done
  )
}
```

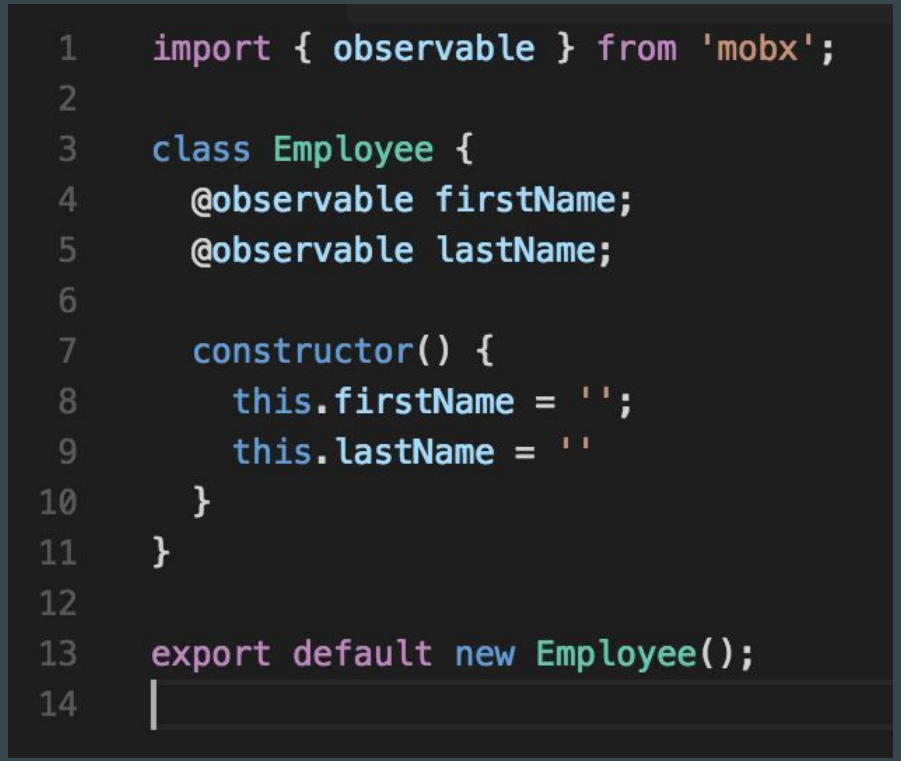
Reactions are like computed values and react to state changes. But they produce a side effect instead of a value, like updating the UI.

```
const Todos = observer({ todos } =>
  <ul>
    todos.map(todo => <TodoView ... />
  </ul>
)
```

Observable (Our State)

MobX adds observable capabilities to existing data structures.

Changes will trigger observers.



Derivations (Computed Values)

With MobX you can define values

that will be derived automatically

when relevant data is modified.

```
@computed get fullName(){
  return `${this.firstName} ${this.lastName}`
}
```

MobX will ensure that *fullName*

is updated automatically when firstName

or lastName is modified.

Observer (Our View)

Observer decorator can be used to turn ReactJS components into reactive components.

Changes to employee will trigger a re-render of EmployeeInfo.

```
import React, { Component } from 'react';
import { observer } from 'mobx-react';

@observer
export default class EmployeeInfo extends Component {

  render() {
    const { firstName, lastName, fullName } = this.props.employee;
    return (
      <div>
        Firstname:
          <input type="text" value={firstName} />
        Lastname:
          <input type="text" value={lastName} />
        <br />
        <div>Your name is {fullName}</div>
      </div>
    );
  }
}
```

Provider (Injection)

In order to bind a component and store(s),

we can use Provider component.

```
const employee = new Employee();

render(
  <Provider employee={employee}>
    <EmployeeInfo />
  </Provider>,
  document.getElementById('root')
);
```

Inject allow a component to have store

available as props.

```
@inject('employee') @observer
export default class EmployeeInfo extends Component
```

In Redux, it is similar to **mapstatetoprops.**

```
Firstname:
  <input type="text" value={this.props.employee.firstName}
```

Action (State Change)

```
Firstname:
  <input type="text" value={firstName} onChange={this.onChangeFirstName} />
```

```
onChangeFirstName = (e) => {
  this.props.employee.changeFirstName(e.target.value);
}
```

With store injected, just a simple function call

on the store object.

```
@action
changeFirstName = (firstName) => {
  this.firstName = firstName;
}
```

Custom Reactions (Side Effects)

Add custom function to run

when state is updated.

Eg. logging, validations...

Optimized to only run when

firstName is changed.

```
1   import { observable, action, autorun } from 'mobx';
2
3   export default class Employee {
4     @observable firstName;
5     @observable lastName;
6
7     constructor() {
8       autorun(() => {
9         console.log('firstName updated to ' + this.firstName)
10      })
```

Complete Code (View)

```
import React, { Component } from 'react';
import { observer, inject } from 'mobx-react';

@inject('employee') @observer
export default class EmployeeInfo extends Component {

  onChangeFirstName = (e) => {
    this.props.employee.changeFirstName(e.target.value);
  }

  onChangeLastName = (e) => {
    this.props.employee.changeLastName(e.target.value);
  }

  render() {
    const { firstName, lastName, fullName } = this.props.employee;
    return (
      <div>
        Firstname:
          <input type="text" value={firstName} onChange={this.onChangeFirstName} />
        Lastname:
          <input type="text" value={lastName} onChange={this.onChangeLastName} />
        <br />
        <div>Your name is {fullName}</div>
      </div>
    );
  }
}
```

Complete Code (Store)

```
1   import { observable, action, autorun, computed } from 'mobx';
2
3   export default class Employee {
4     @observable firstName;
5     @observable lastName;
6
7     constructor() {
8       autorun(() => {
9         console.log('firstName updated to ' + this.firstName)
10      })
11    }
12
13    @computed get fullName() {
14      return `${this.firstName} ${this.lastName}`
15    }
16
17    @action
18    changeFirstName = (firstName) => {
19      this.firstName = firstName;
20    }
21
22    @action
23    changeLastName = (lastName) => {
24      this.lastName = lastName;
25    }
26  }
27
```

More advance stuff

Control equality check

- asReference
- asFlat
- asStructure
- asMap

Side effects

- Reaction
- When ....

What We Have Learned

...

Team Andromeda

What we like?

- Simple to learn and start using
- Less boilerplates
- Can control where state is updated (StrictMode)
- Unopinionated (not one way of doing things)

So what is the catch?

Enabling Decorators

Decorator is a Stage-2 ES.next feature.

Need to enable it in babel.

```
{
  "presets": [
    "react",
    "es2015",
    "stage-1"
  ],
  "plugins": ["transform-decorators-legacy"]
}
```

Or alternative without decorators:

```
class Timer {
    constructor() {
        extendObservable(this, {
            /* See previous listing */
        })
    }
}
```

React component class, ES6:

```
const Timer = observer(class Timer extends React.Component {
    /* ... */
})
```

Defining Proptypes

In components where @Inject was used, to define proptypes, we need to use wrappedComponent.

```
EmployeeInfo.wrappedComponent.propTypes = {
  employee: PropTypes.object.isRequired,
}
```

@Inject wraps original component with another component. To access it, we need to access the inner component.

Same with testing

To shallow render with Enzyme, we need to render the original component, not the one introduced by Mobx.

```
shallow(<EmployeeInfo.wrappedComponent employee={fakeEmployee} />);
```

Extra proptypes from mobx

`PropTypes`

MobX-react provides the following additional `PropTypes` which can be used to validate against MobX structures:

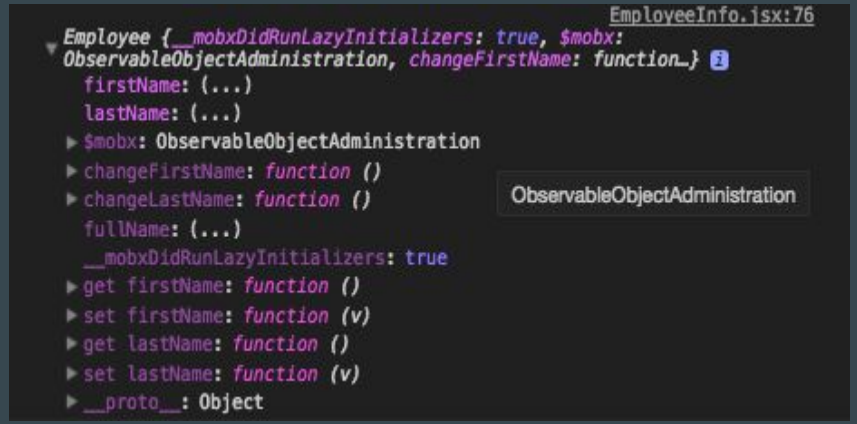
- `observableArray`
- `observableArrayOf(React.PropTypes.number)`
- `observableMap`
- `observableObject`
- `arrayOrObservableArray`
- `arrayOrObservableArrayOf(React.PropTypes.number)`
- `objectOrObservableObject`

Use `import { PropTypes } from "mobx-react"` to import them, then use for example `PropTypes.observableArray`

Serializing Observable Objects

When we console out an @observable object.

```
render() {
  const { firstName, lastName, fullName } = this.props.employee;
  console.log(this.props.employee);
```

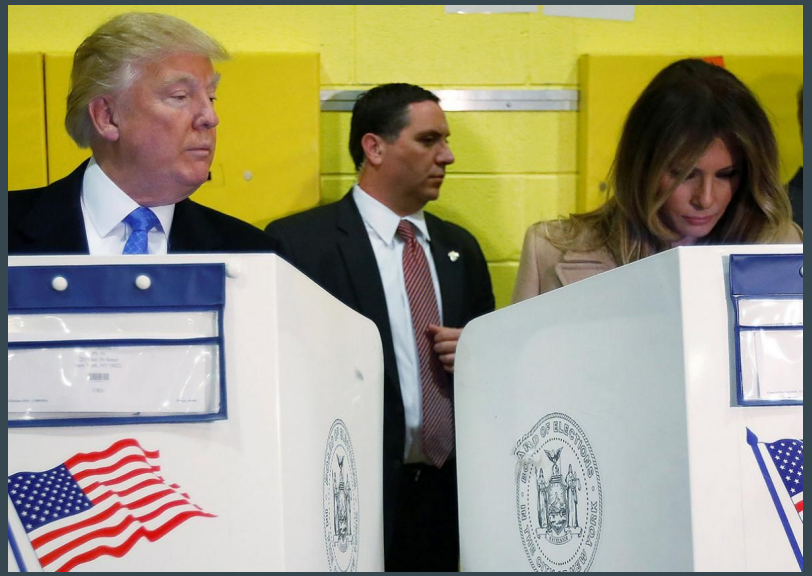


Solution:

```
render() {
  const { firstName, lastName, fullName } = this.props.employee;
  console.log(mobx.toJS(this.props.employee));
```

Would we use it again?

Team Andromeda
voted and ....



Yes

Thank You. Questions?