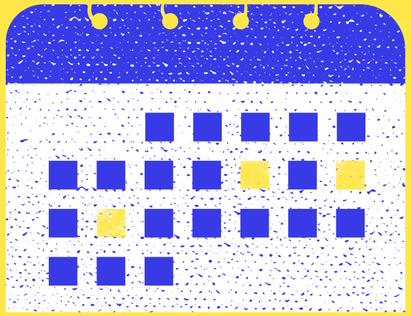
MobX

Simple, Scalable State Management



Agenda

- Introduction
- Building intuition for MobX
- MobX in action
- MobX with React
- React optimizations with MobX
- Peek into MobX
- Resources
- QnA

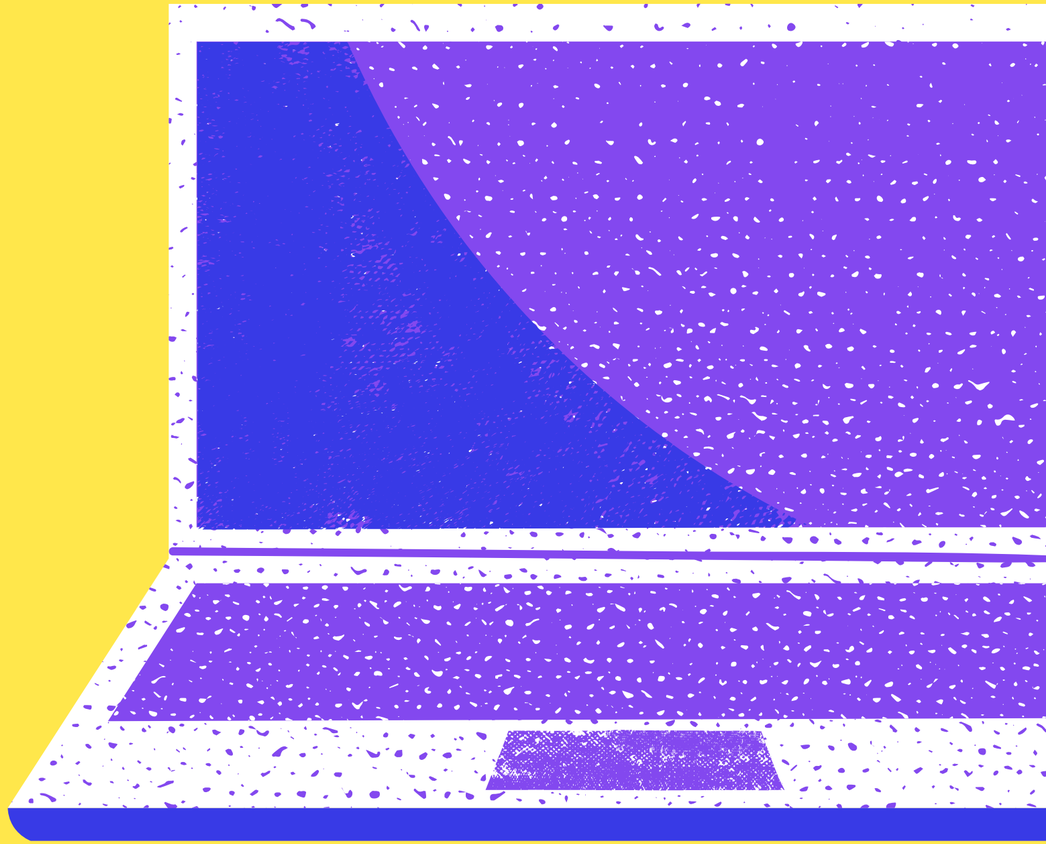
Introduction

- UI should be a pure function of the state of the application. The rendering should be based solely on the state, but without changing it.

- Pure user interfaces are easy to reason about, to mock and to test.

  UI = view (applicationState)

- It's all the fault of the assignment operator!



Problems

- Once the `applicationState` starts to change, the UI becomes stale unless the developer intervenes.
- Creating a complete fresh view from the application state is usually expensive for any decently sized web app.
- We take a best effort guess about which parts of the UI need an update in our event handlers. This clutters code and is error prone. We create events on top of our state that notify view components about state changes which introduces a lot of boilerplate subscription management.

Solutions

- Re-apply the `view` function at 60 frames per second and no user will ever see stale data. This is how games work. (Canvas)
- Using immutable data + ReactJS is a popular approach which renders fast and removes boilerplate from the UI, but introduces boilerplate code around state management. (DOM)
- Keep the UI in sync with state without trashing performance by fixing the assignment operator.

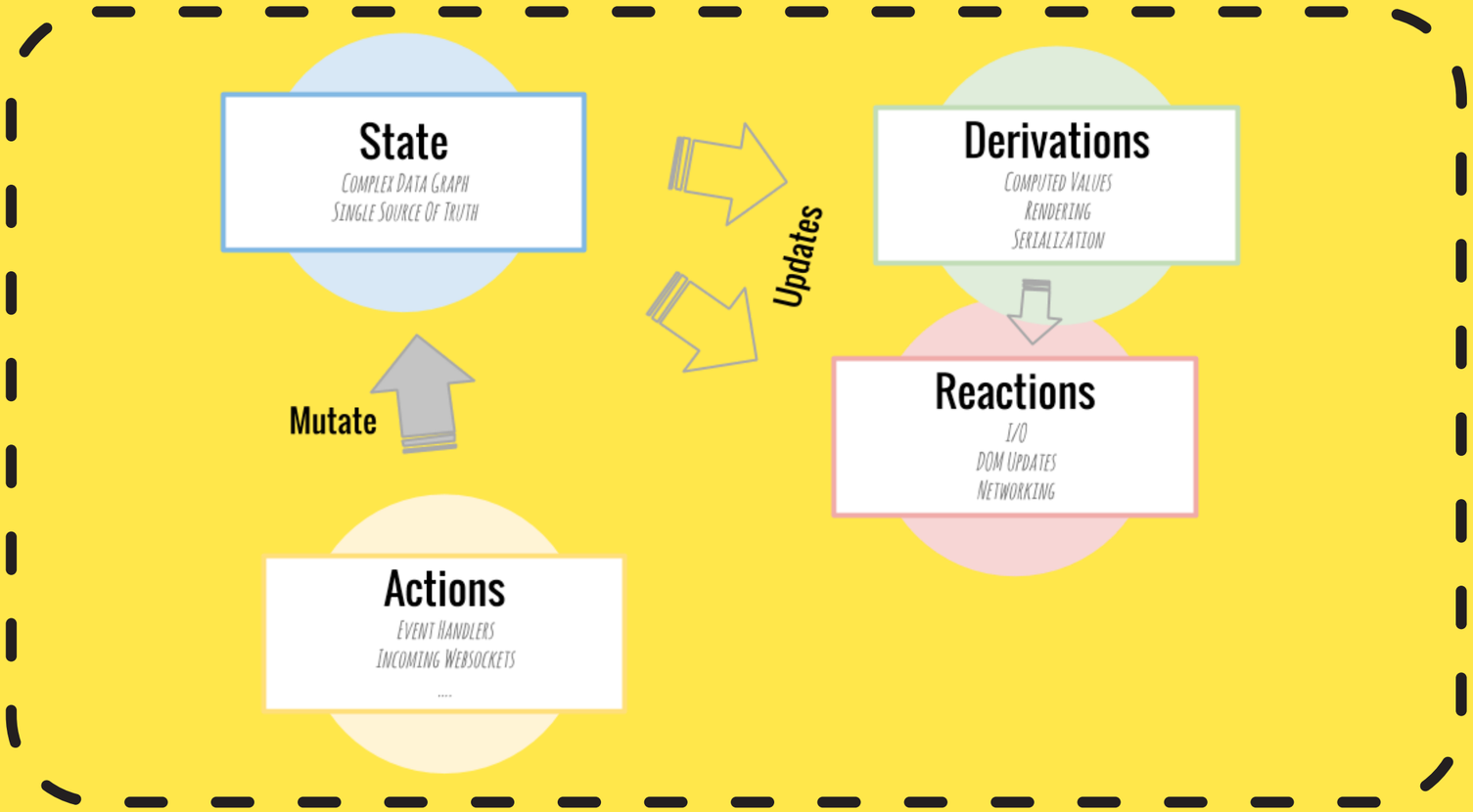
You guessed it right!

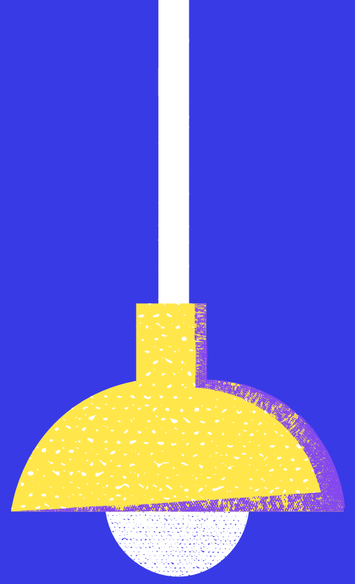
C100 = SUM(B2:B99) / A1

In Excel, the equal sign establishes a proper relationship.
`C100` will follow any changes that happen in the `B` column or `A1`.

TFRP (Transparent Functional Reactive Programming)

Conceptually MobX treats your application like a spreadsheet.





MobX in action

- Making a reactive Todo Store
- Debugging it with breakpoints

MobX with React

- Time to build a reactive user interface around this very same store.
- No useState setter calls anymore, nor do we have to figure out how to subscribe to the proper parts of the application state using selectors or higher order components that need configuration.
- Now all components will become smart. Yet they are defined in a dumb, declarative manner.

React Optimizations

Use many small components

Render lists in dedicated components

Dereference values late

Function props

Render lists in dedicated components

Bad

```
const MyComponent = observer(({ todos, user }) => (
    <div>
        {user.name}
        <ul>
            {todos.map(todo => (
                <TodoView todo={todo} key={todo.id} />
            ))}
        </ul>
    </div>
))
```

Good

```
const MyComponent = observer(({ todos, user }) => (
    <div>
        {user.name}
        <TodosView todos={todos} />
    </div>
))

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

Dereference values late

Slower

```
<DisplayName name={person.name} />
```

Faster

```
<DisplayName person={person} />
```

Function props

Tedious

```
const PersonNameDisplayer = observer(({ person }) => <DisplayName name={person.name} />)

const CarNameDisplayer = observer(({ car }) => <DisplayName name={car.model} />)

const ManufacturerNameDisplayer = observer({ car}) => (
    <DisplayName name={car.manufacturer.name} />
)
```

Better

```
const GenericNameDisplayer = observer(({ getName }) => <DisplayName name={getName()} />)

const MyComponent = ({ person, car }) => (
    <>
        <GenericNameDisplayer getName={() => person.name} />
        <GenericNameDisplayer getName={() => car.model} />
        <GenericNameDisplayer getName={() => car.manufacturer.name} />
    </>
)
```

Lets peek into MobX

Aim?
Share my learning from the MobX source code.

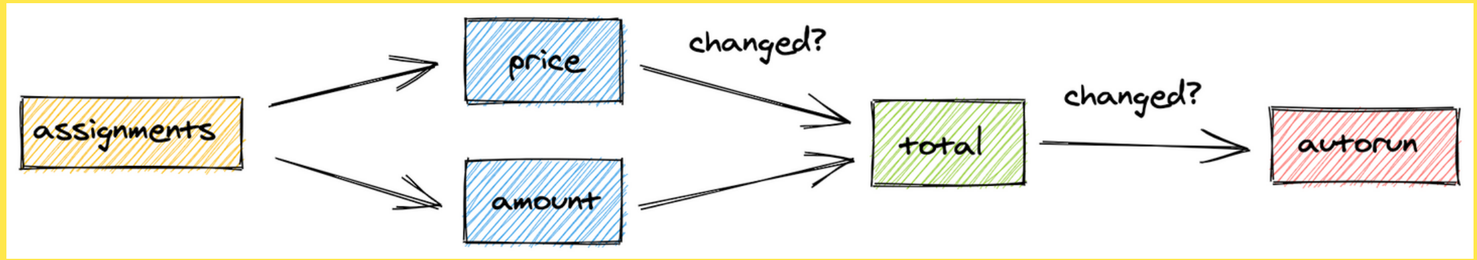
Why?
Better understanding while working with MobX.

Make MobX better. It's open source.

Computeds

```
import { makeObservable, observable, computed, autorun } from "mobx"

class OrderLine {
    price = 0
    amount = 1

    constructor(price) {
        makeObservable(this, {
            price: observable,
            amount: observable,
            total: computed
        })
        this.price = price
    }

    get total() {
        console.log("Computing...")
        return this.price * this.amount
    }
}

const order = new OrderLine(0)

const stop = autorun(() => {
    console.log("Total: " + order.total)
})
// Computing...
// Total: 0

console.log(order.total)
// (No recomputing!)
// 0

order.amount = 5
// Computing...
// (No autorun)

order.price = 2
// Computing...
// Total: 10

stop()

order.price = 3
// Neither the computation nor autorun will be recomputed.
```

- Computed values can be created by annotating JavaScript getters with computed.
- Computed values can be used to derive information from other observables. They evaluate lazily, caching their output and only recomputing if one of the underlying observables has changed.
- Conceptually they are very similar to formulas in spreadsheets.
- They help in reducing the amount of state you have to store and are highly optimized.



Dependency graph

Resources

- Michel Weststrate (Author of MobX)
- JSNation Conference (June 9-11, 2021)
- Blog on Pure rendering
- Manage Complex State in React Apps with MobX

Got any questions?

You can always reach out to me.

CRED

Aditya Sharma

Twitter

LinkedIn

Thank You

See you at CRED.
We are hiring.