State Management beyond the libraries

@mweststrate - Mendix
HolyJS 2018

HOLY
JS/...

NOT SO

Rebranding of this conference

State management design

Introduction to MobX

Patterns are beautiful!

Conclusion

Rebranding of this conference

State management design

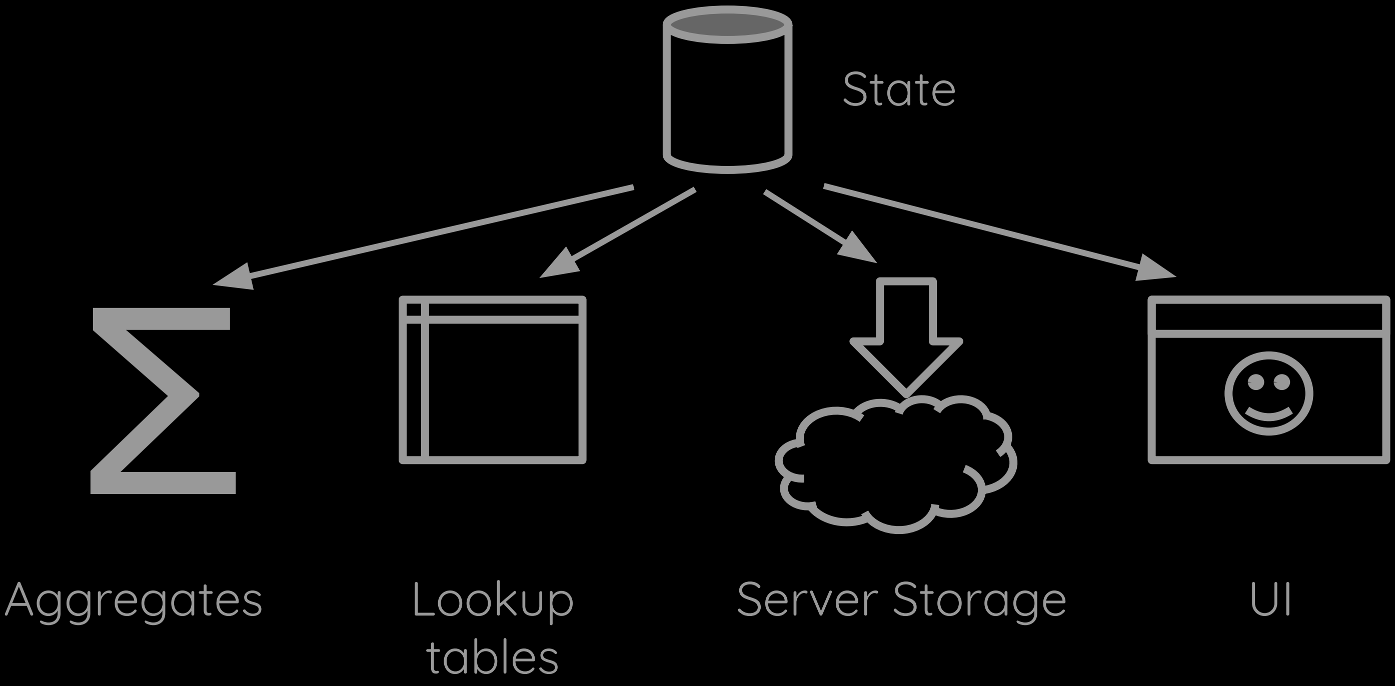
Introduction to MobX

Patterns are beautiful!

Conclusion

Getting information from A to B

And keeping all consumers up to date after mutation



State management essentials

What are the moving parts?

How are changes propagated?

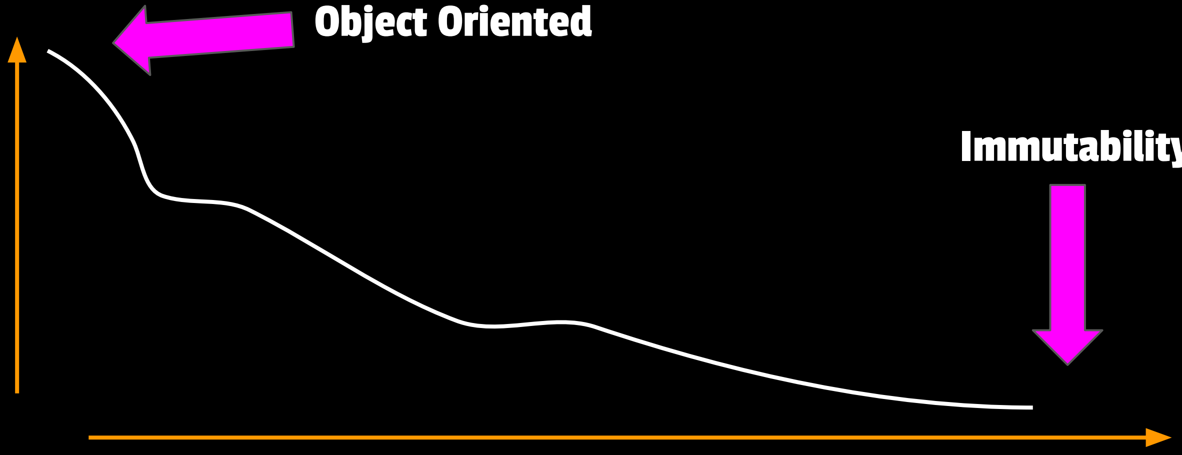
Where does state live?

How to deal with references?

The State Management Paradox

Ease of reading, writing and optimization

Object Oriented



Immutability

Amount of moving parts

“You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.”

Joe Armstrong

Let’s peel a banana

a.k.a. how hard is it to flip a boolean?

```
class Banana {
  peeled = false

  setPeeled(value) {
    this.peeled = value
  }
}

forest.trees[18].gorillas[“Joe”].banana.setPeeled(true)
```

```
function peelBanana(forest, treeIdx, gorillaName, peeled) {
  return {
    ...forest,
    trees: forest.trees.map((tree, idx) =>
      idx !== treeIdx ? tree : {
        ...tree,
        gorillas: {
          ...tree.gorillas,
          [gorillaName] : {
            ...tree.gorillas[ gorillaName],
            banana: {
              ..tree.gorillas[ gorillaName].banana,
              peeled: peeled
            }
          }
        }
      }
    )
  }
}

store.setState(peelBanana(store.getState(), 18, “Joe”, true))
```

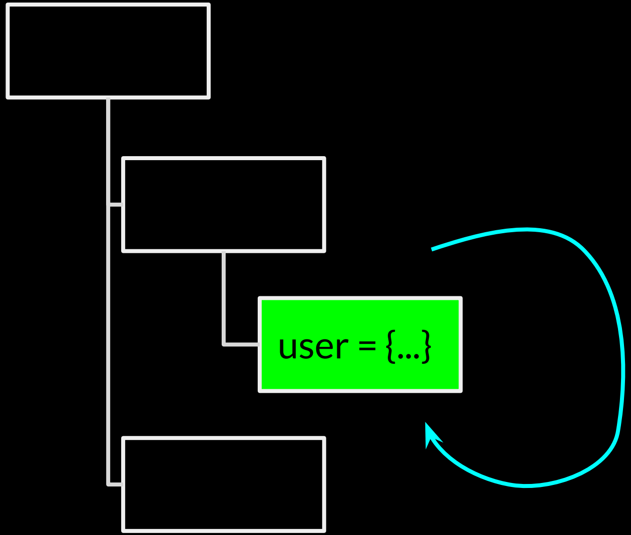
Let’s not dismiss ideas with clever one liners

In JavaScript, nothing should be considered holy

NOT SO HOLY JS /...

Example: React setState

Example: React setState

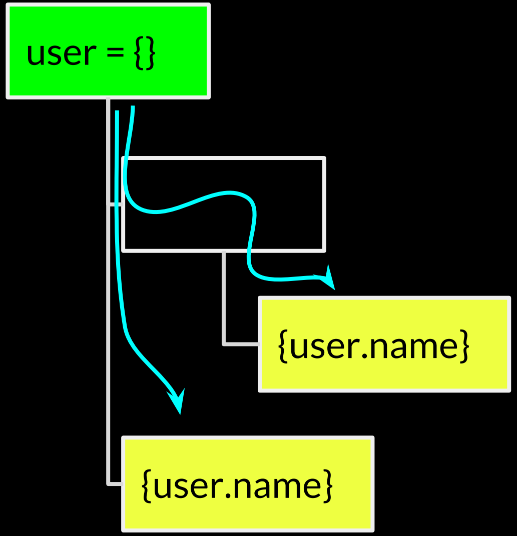


setState

imperative render()

no subscribers

Example: React setState



setState

update entire subtree

requires optimization

touches unrelated components

State management essentials

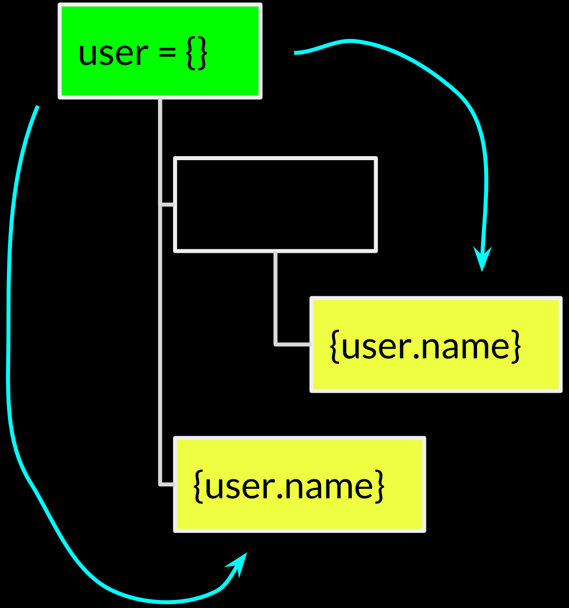
What are the moving parts?

How are changes propagated?

Where does state live?

How to deal with references?

Example: React context



subscribe to data higher in tree

re-render on change

What should we subscribe to?

“Something changed”

“Some user changed”

“User 5423 changed ”

“User 5423.profile.address changed”

Context

Single event per context

Redux

Global event, but select relevant parts

MobX

Event per property, selection is automatic

MobX Demo

State management essentials

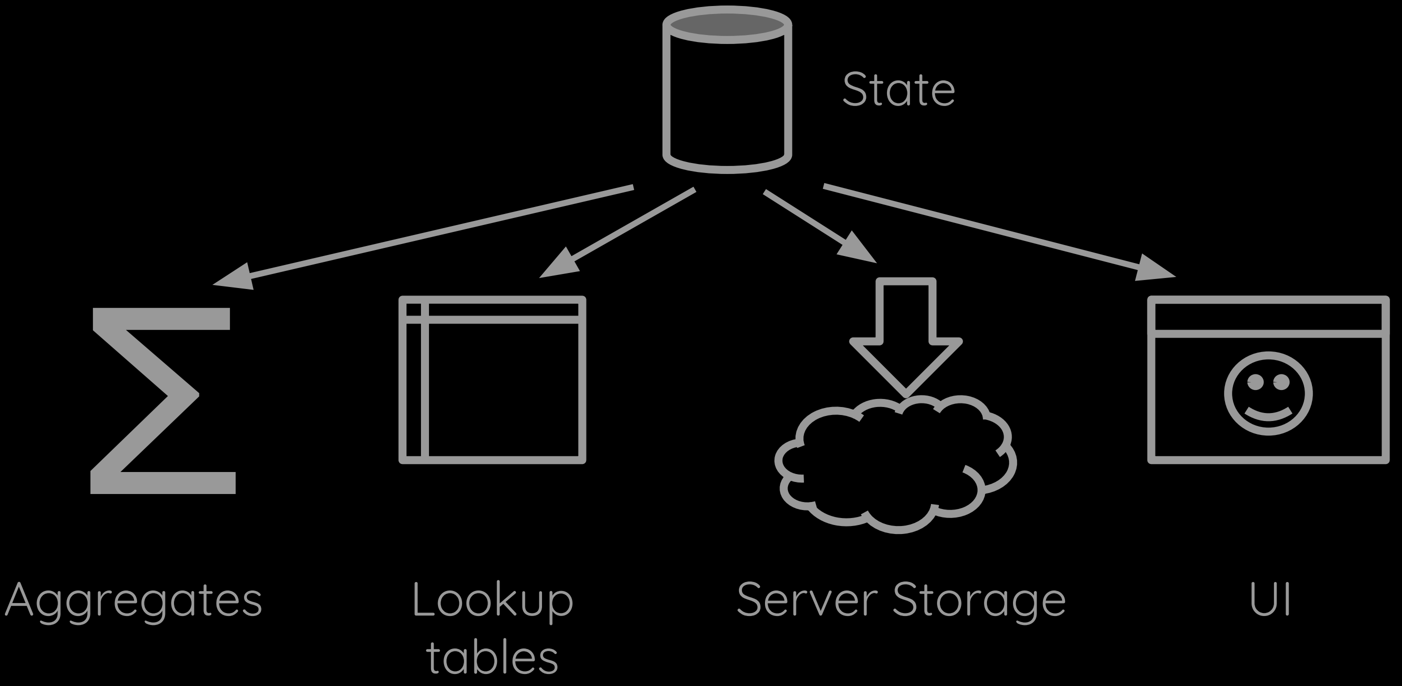
What are the moving parts?

How are changes propagated?

Where does state live?

How to deal with references?

Side effects can only live outside components, if state can



State

Aggregates

Lookup tables

Server Storage

UI

React state

Component

Redux

Single store

MobX

User defined objects / classes

Decouple state
from UI layer

State management essentials

What are the moving parts?

How are changes propagated?

Where does state live?

How to deal with references?

```
class Person {

  constructor(father, mother, firstName) {
    this.father = father
    this.mother = mother
    this.firstName = firstName
    this.lastName = parent.lastName
    this.address = parent.address
  }

}
```

Should change with parent?

Should change with parent!
(Unless..)

```
class Person {

  constructor(father, mother, firstName) {
    this.father = father
    this.mother = mother
    this.firstName = firstName
    this.lastName = parent.lastName
  }

  get address() {
    return this.stillLivingWithParents ?
      this.mother.address : this.ownAddress
  }

}
```

```
class Person {

  constructor(father, mother, firstName) {
    this.fatherID = father.id
    this.motherID = mother.id
    this.firstName = firstName
    this.lastName = parent.lastName
  }

  get address() {
    return this.stillLivingWithParents ?
      this.mother.address : this.ownAddress
  }

  get mother() {
    return citizenStore.get(this.motherID)
  }
}
```

Reference: Identity or Value?

```
function printPrice(book: Book) {
    setTimeout(
        () => console.log(book.price),
        1000
    )
}
```

Price might have changed

VS

Price might be stale

State management essentials

What are the moving parts?

How are changes propagated?

Where does state live?

How to deal with references?

Rebranding of this conference

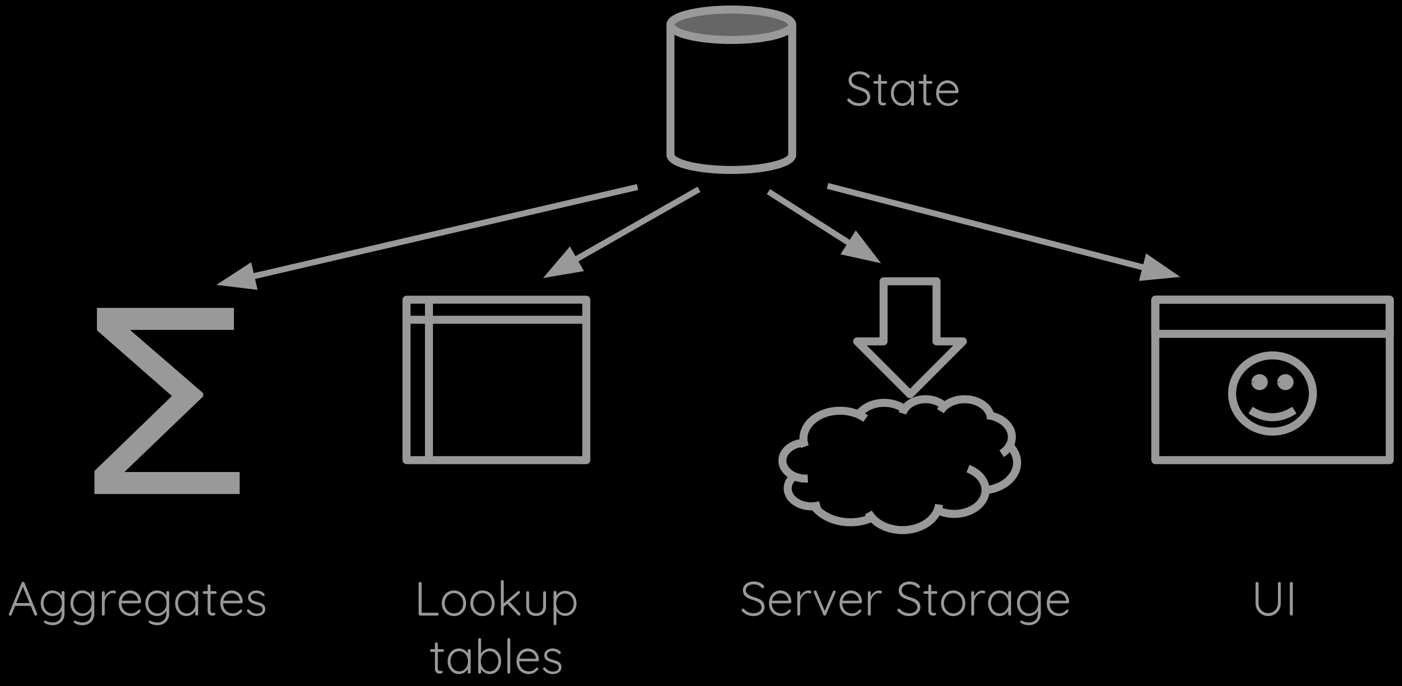
State management design

Introduction to MobX

Patterns are beautiful!

Conclusion

Everything that can be derived from state should be derived. Automatically



Observable values

state that can be change over time

Actions

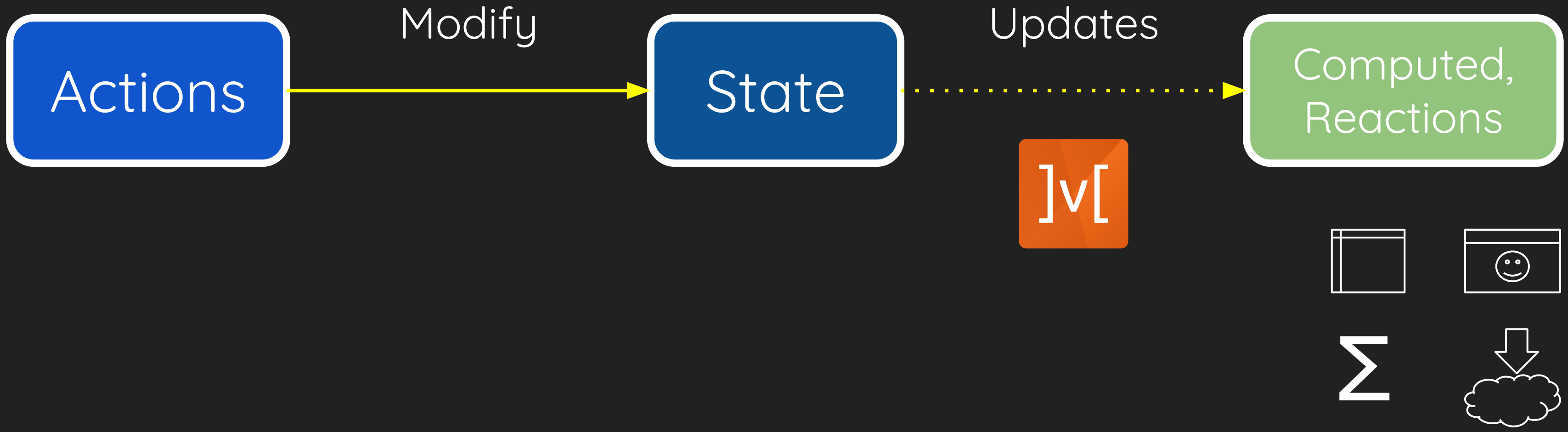
interactions that change state

Computed values

values that can be derived

Reactions

side effects that should respond to state changes



Actions

Modify

State

Updates

Computed, Reactions

Demo

Store design

```
const store = observable({
    cities: {
       MSC: new City({ name: "Moscow", x: 17, y: 12 })
       AMS: new City({ name: "Amsterdam", x: 25, y: 7 })
    },
    arrows: [],
    selection: "MSC"
})

store.arrows.push(
   new Arrow({ from: store.cities.AMS, to: store.cities.MSC })
)
```

The Arrow Component

```
const ArrowView = ({ arrow }) => {
    const {from, to} = arrow;
    const [x1, y1, x2, y2] = [
        from.x + from.width/ 2,
        from.y + 30,
        to.x + to.width / 2,
        to.y + 30
    ]
    return <path className="arrow"
        d={`M${x1} ${y1} L${x2} ${y2}`}
    />
}
```

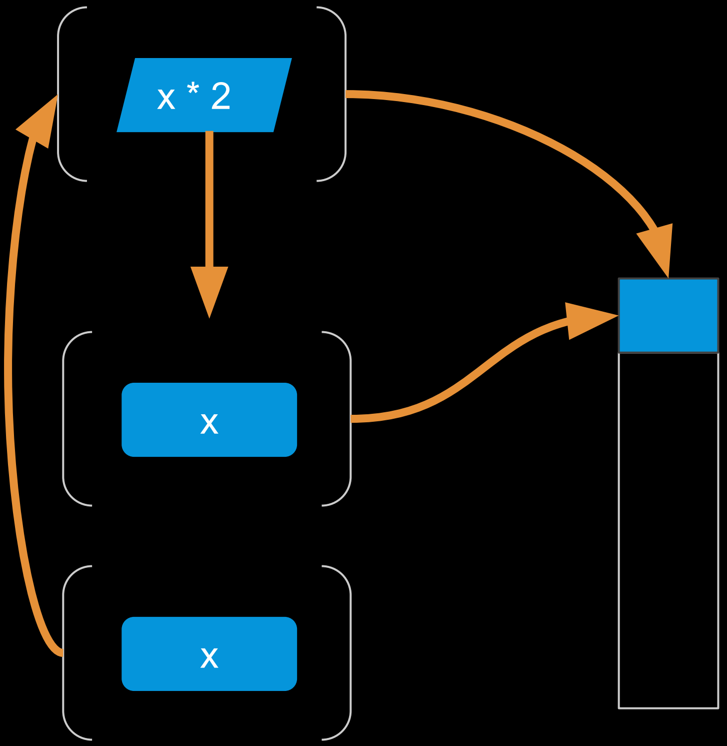
arrow.to
arrow.from

from.x
from.y
from.name

to.x
to.y
to.name

How MobX works

1. Wrap properties with getter / setter
2. Store running function in a stack
3. Getters register observers
4. Setters notify observers
5. MobX optimizes dependency graph



Transparant Reactive Programming

Decoupling of producers & consumers of information

Straightforward to write

Optimized, minimal dependency tree

Rebranding of this conference

State management design

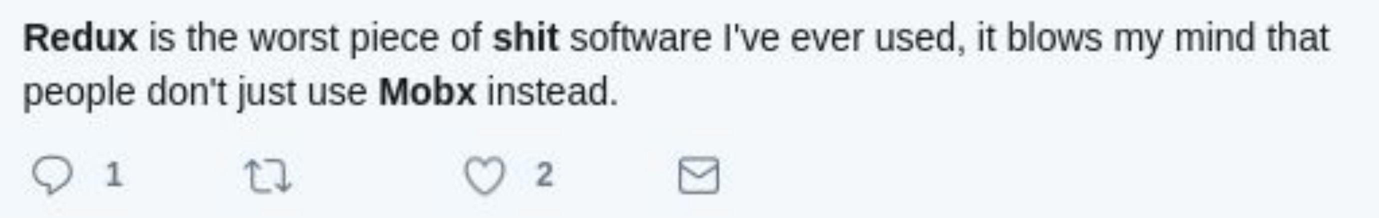
Introduction to MobX

Patterns are beautiful!

Conclusion

MobX or Redux?

Immutable or Mutable



Redux is the worst piece of **shit** software I've ever used, it blows my mind that people don't just use **Mobx** instead.

Don’t 💩 on other ideas

New project…

- React
- Vue
- Angular
- Svelte
- Ember

x

- Webpack
- Parcel
- Browserify

x

- ES5
- ES6
- TypeScript
- Flow
- Reason

x

- Vanilla
- Redux
- RxJS
- Apollo
- MobX

x

- Freestyler
- Emotion
- Fela
- Styled JSS
- React jSS
- Rocky
- Styled Components
- Aphrodite
- Glamour
- Glamourus

Gazillion of options

We can’t justify them all

We don’t want to be seen as ignorant either

Things I never used for real

Angular / Ember / Vue

Redux

RxJS

Immer

MobX-state-tree

Stop defending all the choices you *don’t* make

“Didn’t try” can be fine

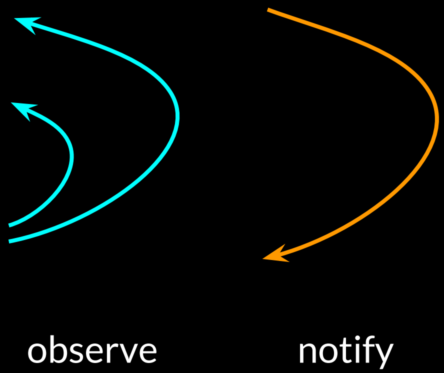
You have to learn to be able to use

but, you don’t have to use to be able learn!

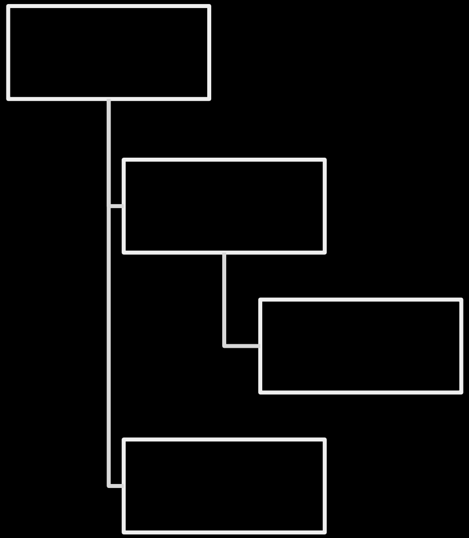
Software Engineering is about Patterns

MobX

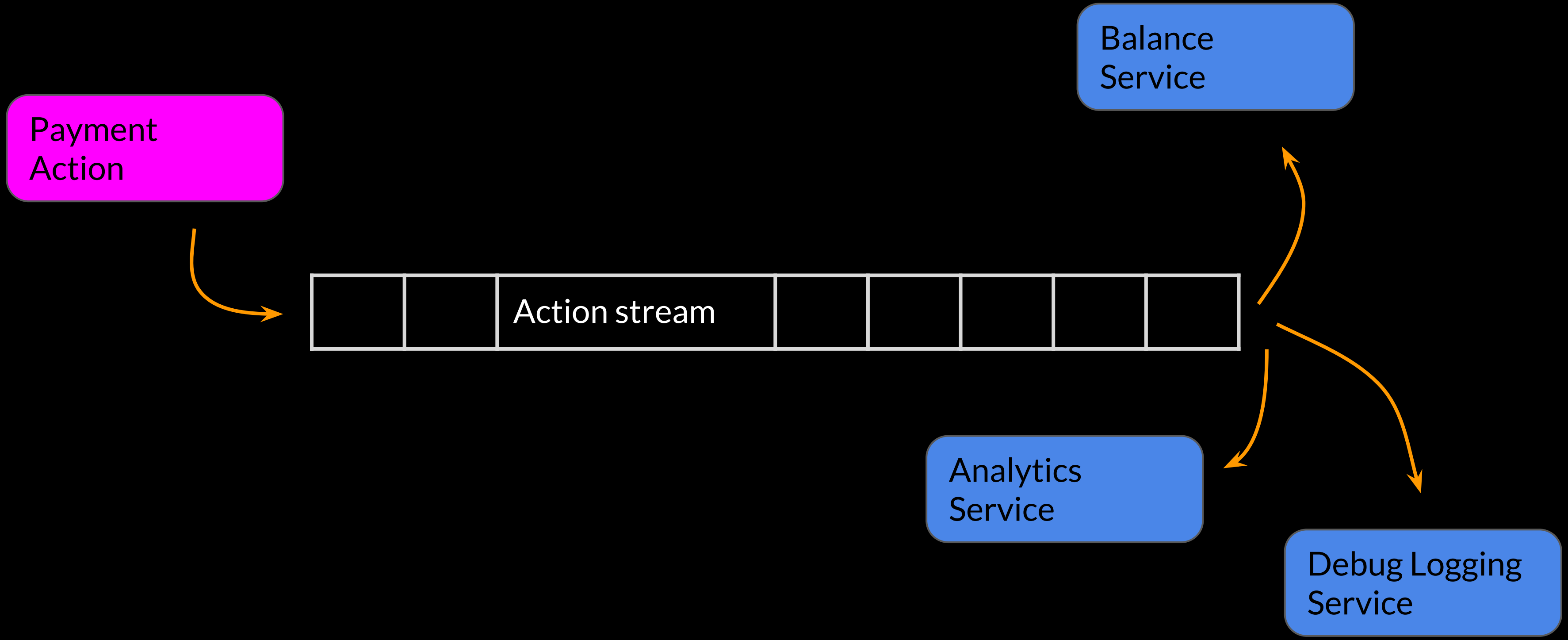
Price	17
Amount	3
Total	= Price * Amount 51



Redux - Immutable Tree



Redux - Action



Payment Action

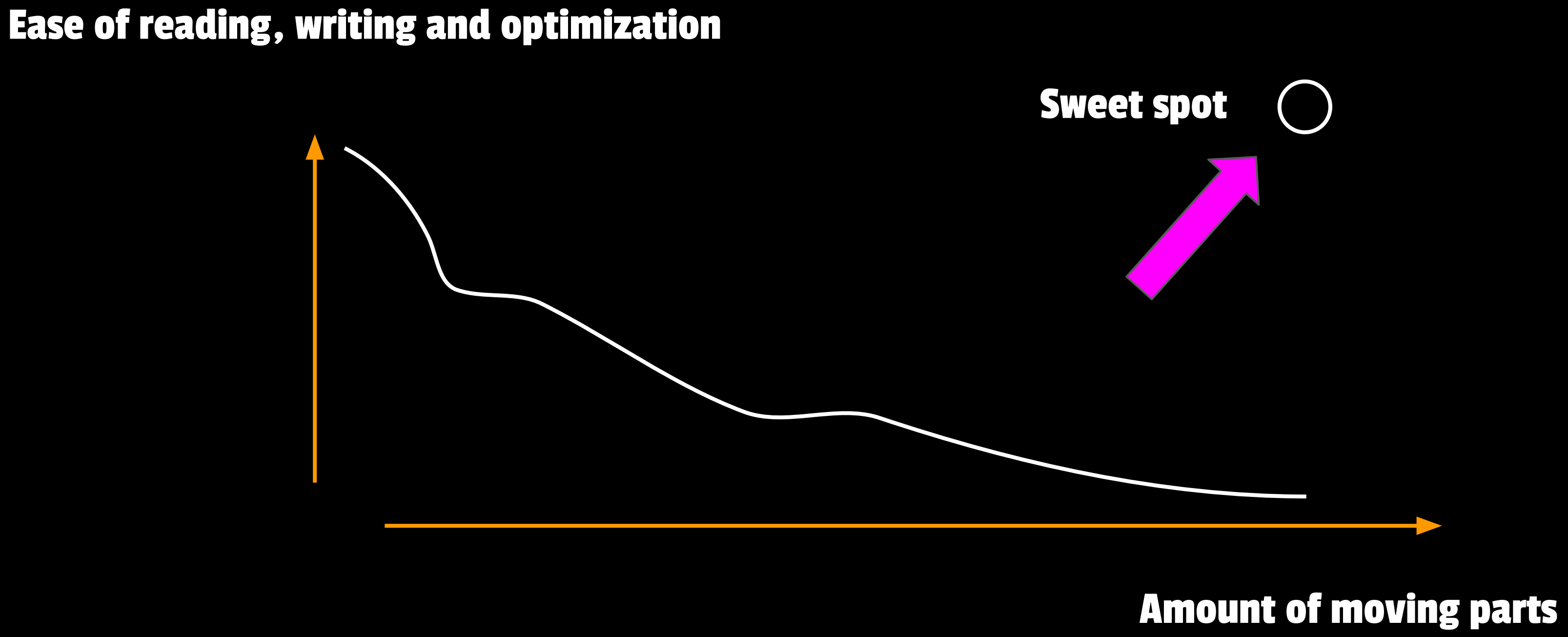
Action stream

Balance Service

Analytics Service

Debug Logging Service

The State Management Paradox



Ease of reading, writing and optimization

Sweet spot

Amount of moving parts

Can we apply Redux patterns to MobX ?

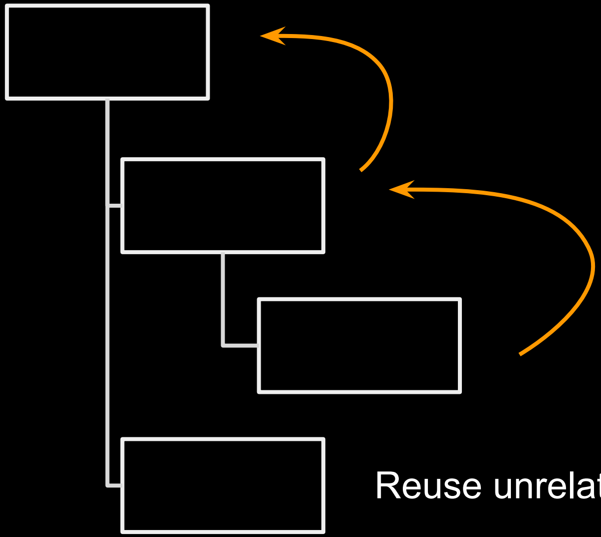
Snapshots

```
class Todo {
   @observable title
   @observable done

   @computed get snapshot() {
       return {
           title: this.title,
           done: this.done
       }
   }
}
```

Snapshots

```
class Todos {
    @observable todos = []

    @computed get snapshot() {
        return {
            todos: this.todos.map(todo =>  todo.snapshot)
        }
    }
}
```

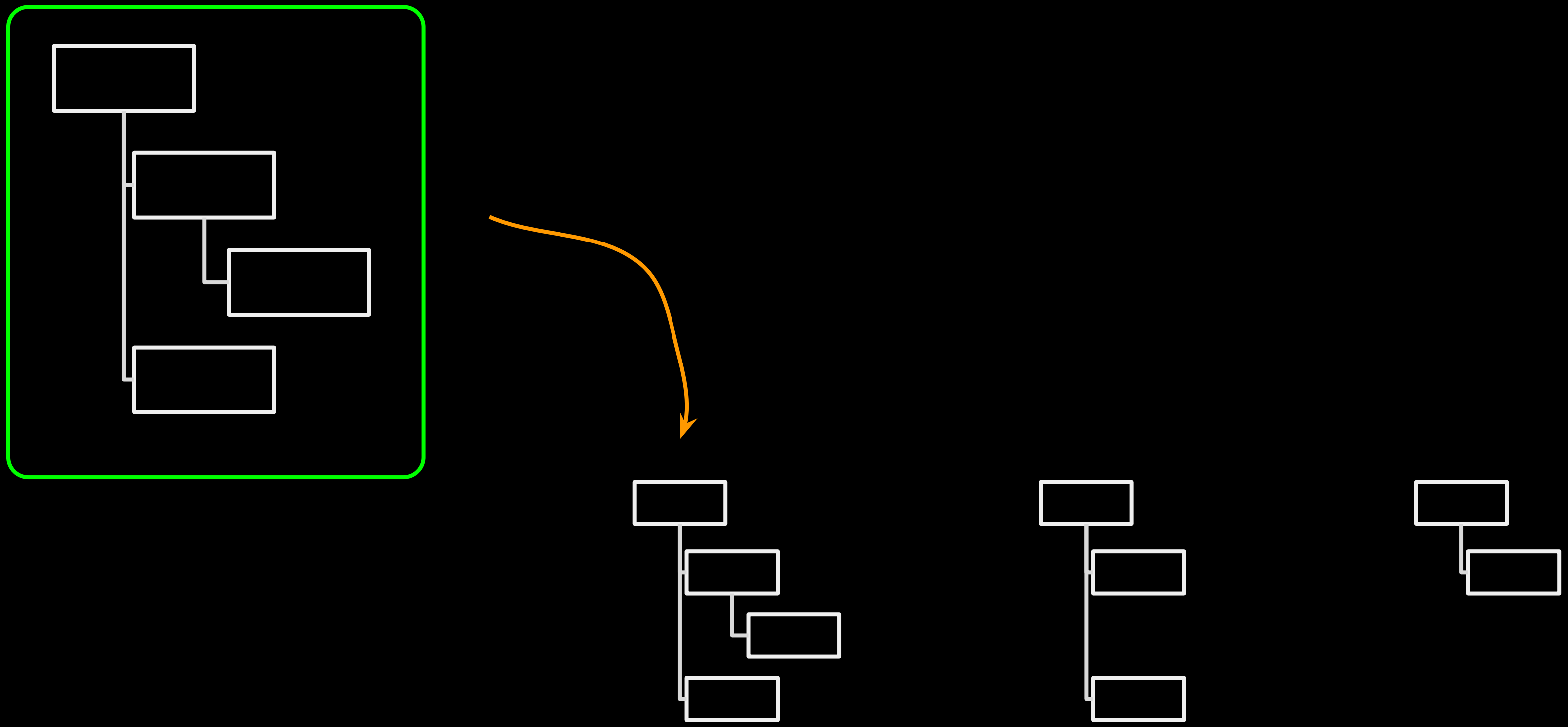
Structural sharing with snapshots

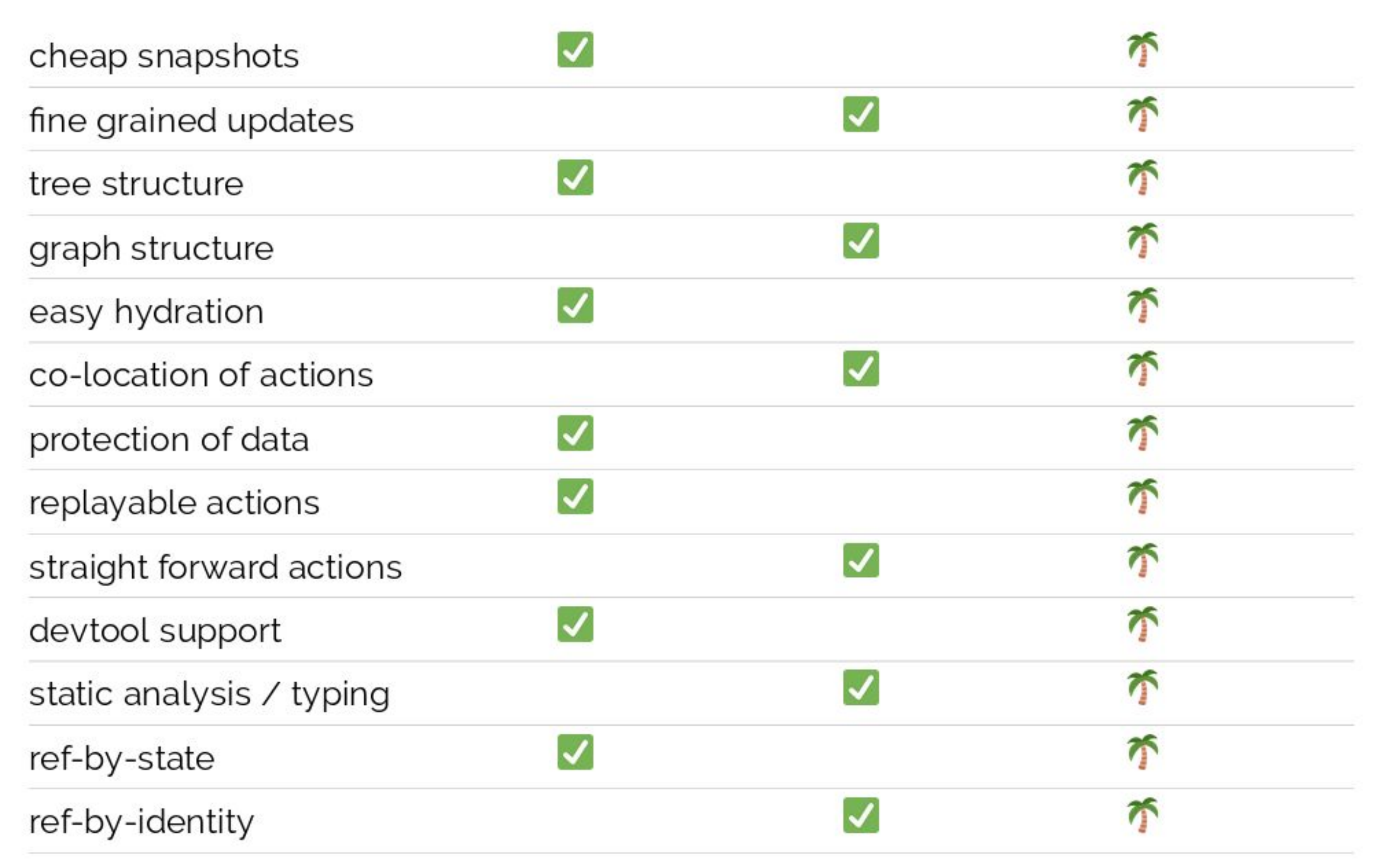


Snapshot changes bubble up

Reuse unrelated snapshots

MobX-state-tree





	Redux	MobX	MST
cheap snapshots	✅		🌴
fine grained updates		✅	🌴
tree structure	✅		🌴
graph structure		✅	🌴
easy hydration	✅		🌴
co-location of actions		✅	🌴
protection of data	✅		🌴
replayable actions	✅		🌴
straight forward actions		✅	🌴
devtool support	✅		🌴
static analysis / typing		✅	🌴
ref-by-state	✅		🌴
ref-by-identity		✅	🌴

Can we apply MobX patterns to Redux?

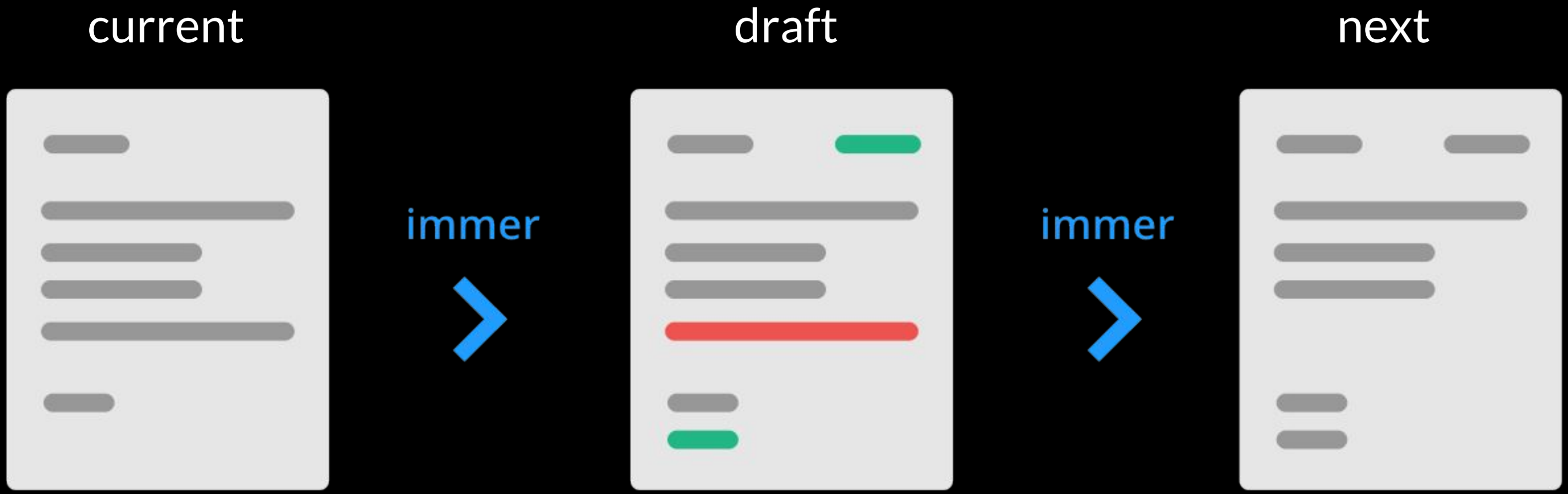
Demo

A reducer…

```
const byId = (state, action) => {
  switch (action.type) {
    case RECEIVE_PRODUCTS:
      return {
        ...state,
        ...action.products.reduce((obj, product) => {
          obj[product.id] = product
          return obj
        }, {})
      }
    default:
      return state
  }
}
```

Immer

```
const byId = produce((draft, action) => {
    switch (action.type) {
      case RECEIVE_PRODUCTS:
        action.products.forEach(product => {
          draft[product.id] = product
        })
        break
    }
})
```



current

immer

draft

immer

next

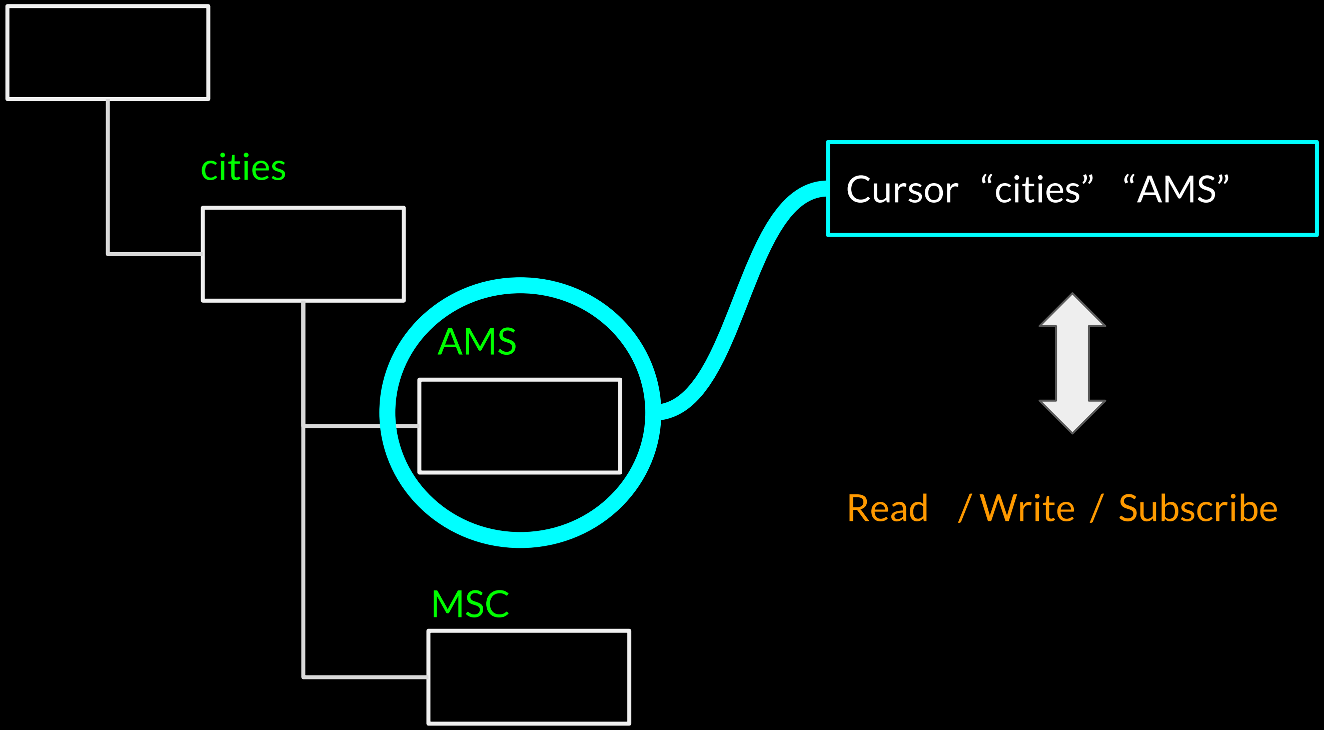
Your edits here.

```
import produce from “immer”

const peelBanana = produce((forest, treeIdx, gorillaIdx, peeled) => {
  forest.trees[treeIdx].gorillas[gorillaName].banana.peeled = peeled
}

store.setState(peelBanana(store.getState(), 18, “Joe”, true))
```

The Remmi experiment

Combining cursors, streams and value transformations

Cursors



Cursor “cities” “AMS”

Read / Write / Subscribe

Store design

```
const store = createStore({
    cities: {
       MSC: { name: "Moscow", x: 17, y: 12 }
       AMS: { name: "Amsterdam", x: 25, y: 7 }
    },
    arrows: {
       a1: { from: "AMS", to: "MSC" }
    },
    selection: "MSC"
})
```

Store

```
store.value()

store.update(draft => { })

store.subscribe(value => { })

store.do(transformations)
```

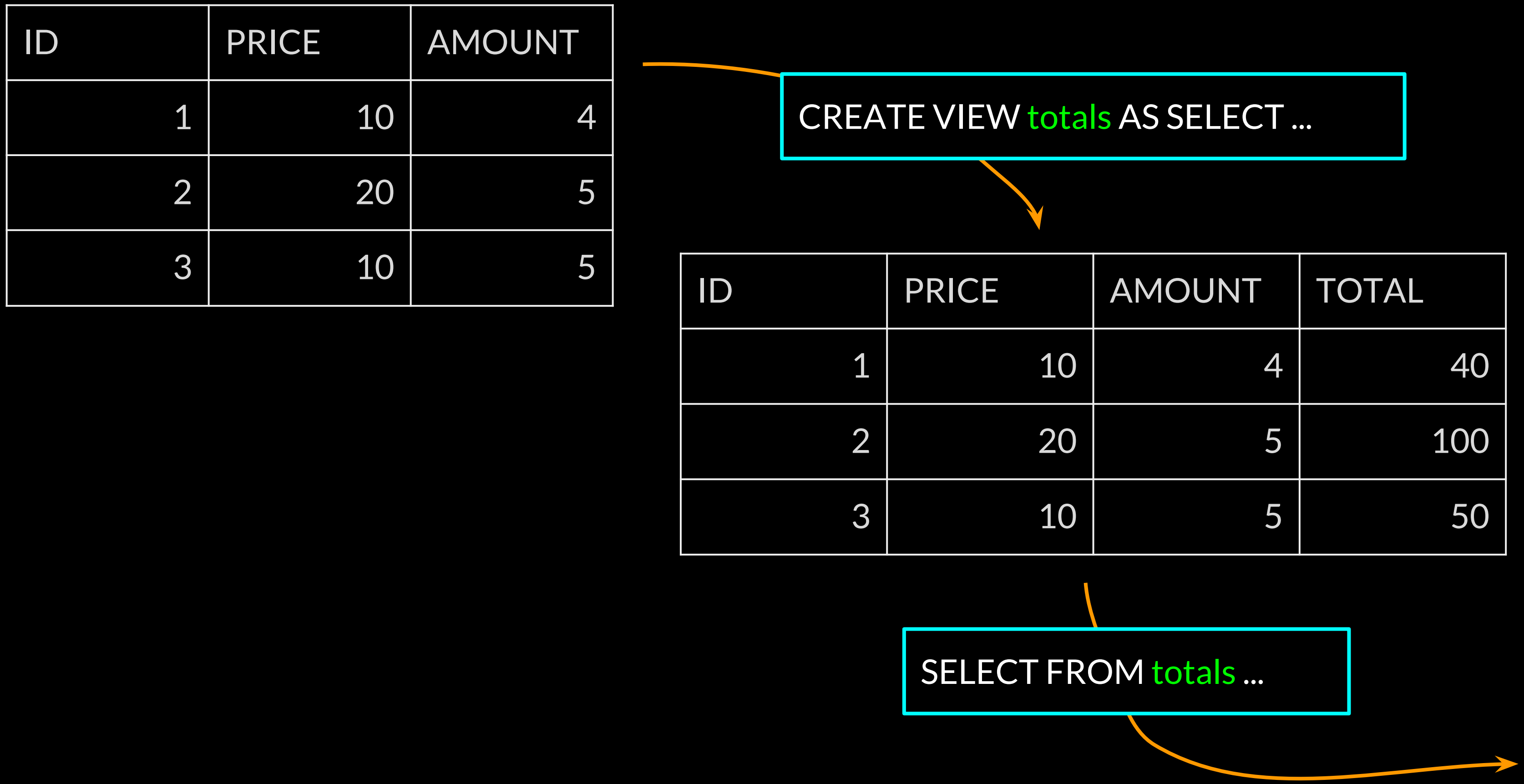
Cursors

```
const amsterdamCursor = store.do(
    select("cities"),
    select("AMS")
)

amsterdamCursor.value()
> { name: "Amsterdam", x: 25, y: 7 }

amsterdamCursor.subscribe(value => {
    console.log(value.name)
})

amsterdamCursor.update(draft => {
    draft.name = "A’Dam"
})

amsterdamCursor.do(select("name"))
```

Cursors

```
const amsterdamCursor = store.do(select(s => s.cities.AMS))
```

Materialized Views



ID	PRICE	AMOUNT
1	10	4
2	20	5
3	10	5

CREATE VIEW totals AS SELECT ...

ID	PRICE	AMOUNT	TOTAL
1	10	4	40
2	20	5	100
3	10	5	50

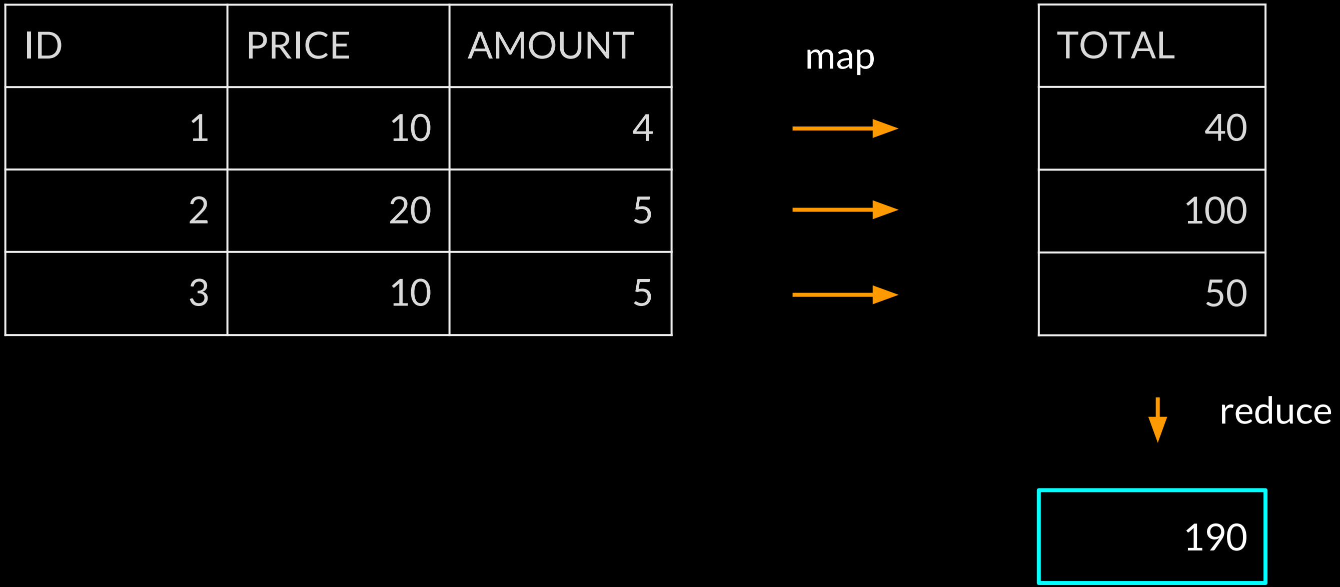
SELECT FROM totals ...

“Materialized” Views

```
const cityNamesCursor = store.do(select(
    s => Object
         .values(s.cities)
         .map(city => city.name)
))
```

Change one city, and all of them need to be mapped!

Map Reduce



ID	PRICE	AMOUNT
1	10	4
2	20	5
3	10	5

TOTAL
40
100
50

190

Map Reduce

```
const cityNamesCursor = store.do(
    select(s => s.cities),
    map(city => city.name)
))
```

Only re-evalutes changed cities

Transforming to React

```
function Sidebar({ selectionCursor }) {
    return selectionCursor.do(
        render(selection => <div>{selection.name}</div>)
    )
}
```

…using hooks!

```
function Sidebar({ selectionCursor }) {
    const selection = useCursor(selectionCursor)
    return <div>{selection.name}</div>
}
```

...using hooks!

```
function useCursor(cursor) {
    const [value, setValue] = useState(() => cursor.value())
    useEffect(() => cursor.subscribe(setValue), [cursor])
    return value
}
```

MobX – Transparent Reactivity

```
const ArrowView = observer(({ arrow }) => {
    const {from, to} = arrow;
    const [x1, y1, x2, y2] = [
        from.x + from.width/ 2,
        from.y + 30,
        to.x + to.width / 2,
        to.y + 30
    ]
    return <path className="arrow"
        d={`M${x1} ${y1} L${x2} ${y2}`}
    />
})
```

Remmi - Cursors

```
const ArrowView = memo(({arrowCursor, citiesCursor}) => {
    const arrow = useCursor(arrowCursor)
    const from = useCursor(citiesCursor.select(arrow.from))
    const to = useCursor(citiesCursor.select(arrow.to))
    const [x1, y1, x2, y2] = [
        from.x + boxWidth(from) / 2,
        from.y + 30,
        to.x + boxWidth(to) / 2,
        to.y + 30
    ]
    return <path className="arrow"
        d={`M${x1} ${y1} L${x2} ${y2}`}
    />
})
```

Demo

https://github.com/mweststrate/remmi

Rebranding of this conference

State management design

Introduction to MobX

Patterns are beautiful!

Conclusion

State management

Identify the moving parts?

How are changes propagated?

Where is state owned?

What is the meaning of a reference?

MobX

Everything that can be derived, should be derived, automatically

NOT SO

HOLY JS /...

There is nothing holy about JS

Don’t swear by anything

Learn from everything