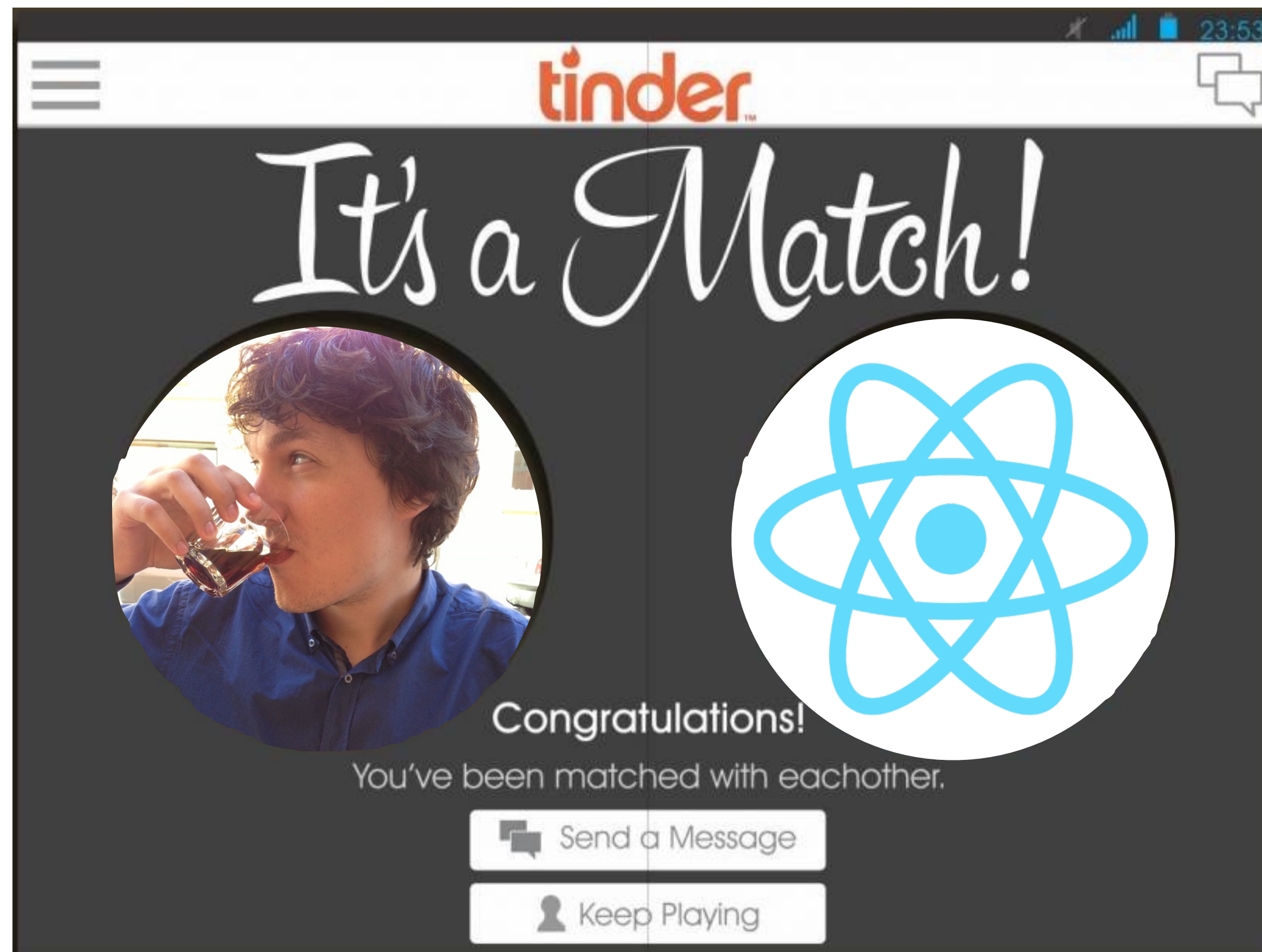


Purifying React



@robinpokorny

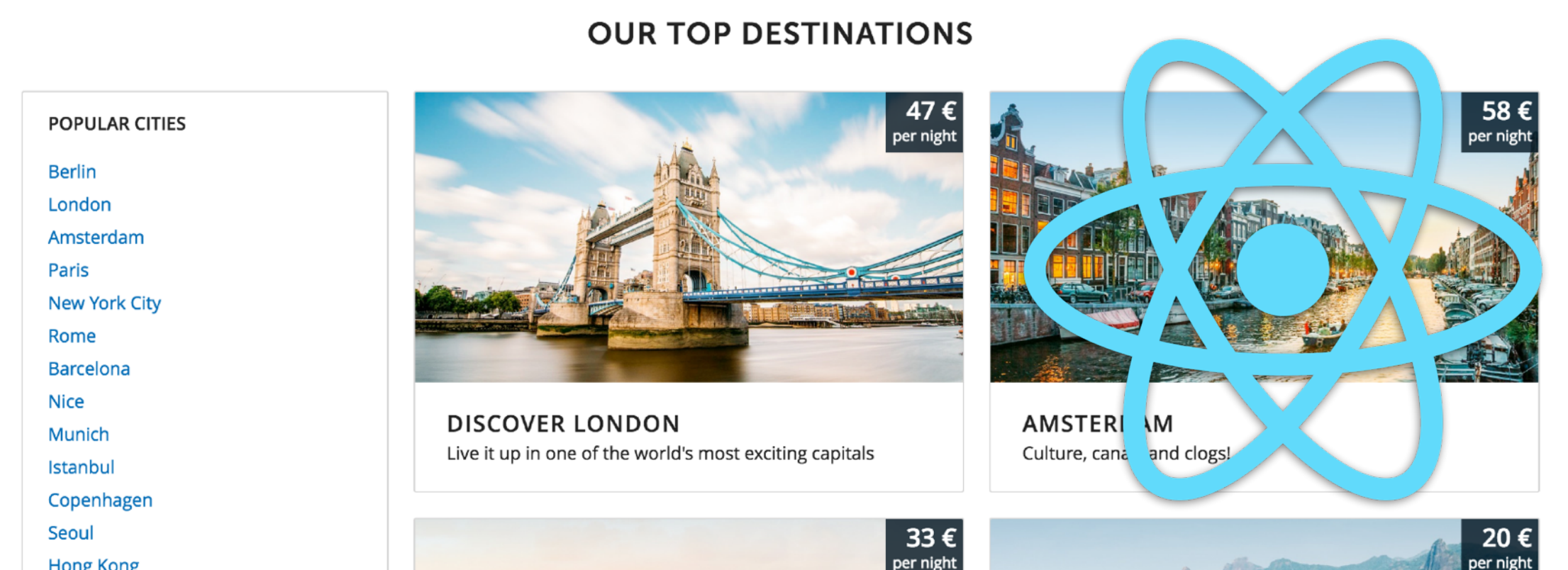
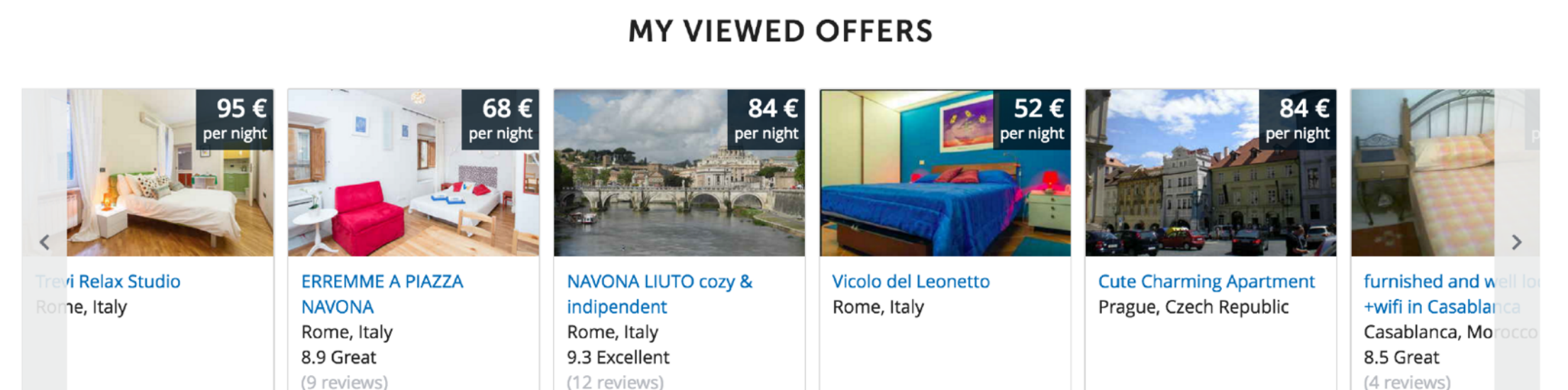
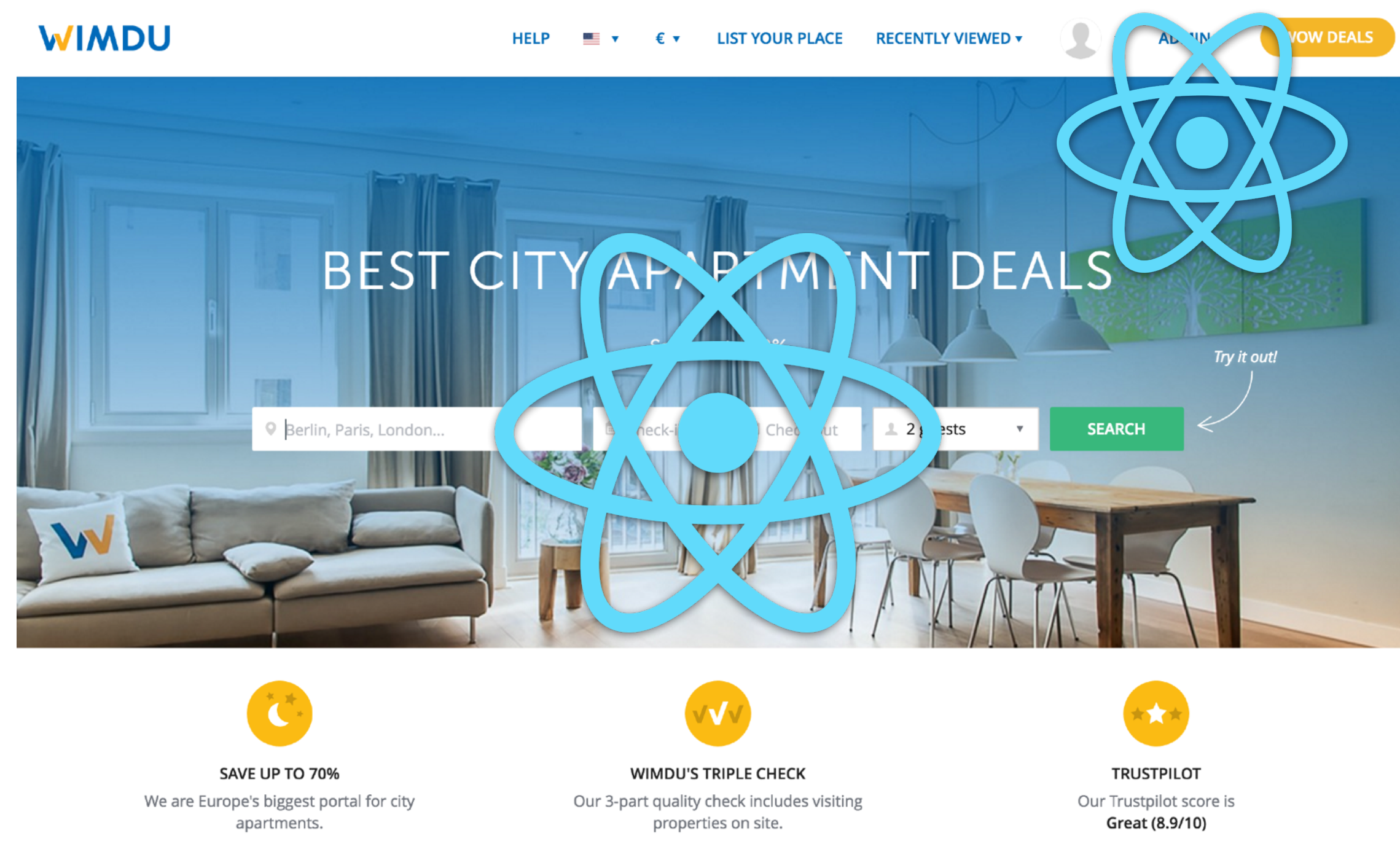
Hi, I will share how we made our front end **pure**, by incrementally introducing **Redux**, **ImmutableJS**, and **higher-order components**, all under constant requests for new features.



I'm Robin and I met React on the internet.

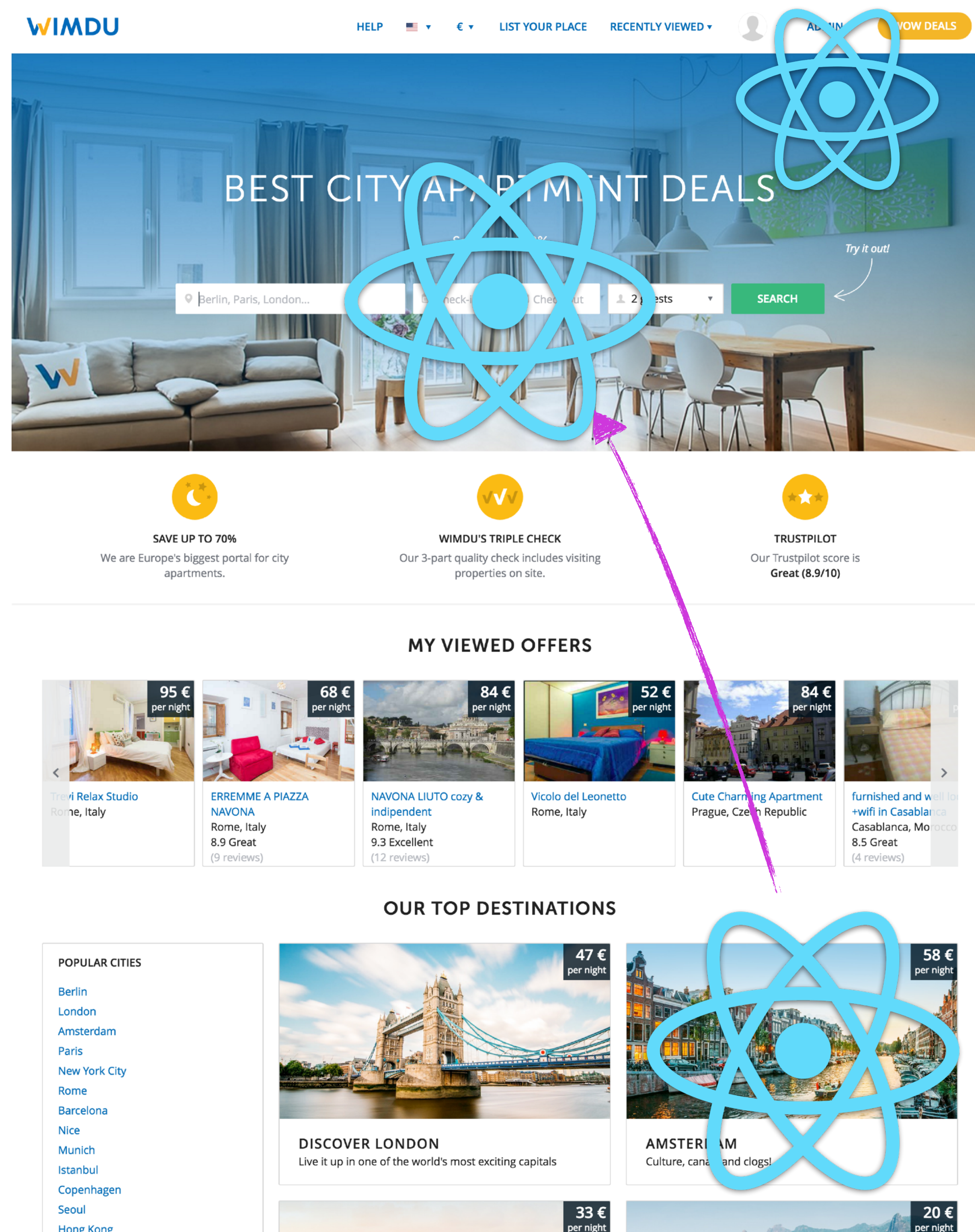
We've been together since.

Nine months ago I joined Wimdu to help it with its front end.



Our site is server-rendered by Rails.

We have a growing number of async-loaded independent **React** components to enhance the page.



The problem occurred when we wanted them to communicate amongst themselves. We decided to implement a state container— Redux.

We only introduced Redux when we felt we needed it.
We try to avoid premature optimisation and over-engineering.

As we were aware that a rewrite would be too big, paralysing us for weeks we came up with an incremental process.

Now, we need to purify our code base...

'Pure' is a concept in functional programming ([learn more](#)).

this to params

Function




We start purifying at the bottom—individual functions.
This was not a project or task. We only refactored code we
were touching during our regular work.

Old

```
renderGroups() {  
  const { groups } = this.props  
  
  ...  
}
```

New

```
renderGroups(groups) {  
  ...  
}
```

instacod.es/107138 

First step was easy.
Get rid of **this** and pass data in parameters.
Only lifecycle function could still access **this**.


```
const addParam = (options, name, param) => {  
  options[name] = param  
}
```

```
const addParam = (options, name, param) => {  
  return Object.assign(  
    {},  
    options,  
    { [name]: param }  
  )  
}
```

Second step proved to be more challenging.
Instead of changing the object, function should return
changed object without **modifying** the original.

state to props

this to params

Component

Function




When all functions in a component are pure,
we make the component pure, too

propstate

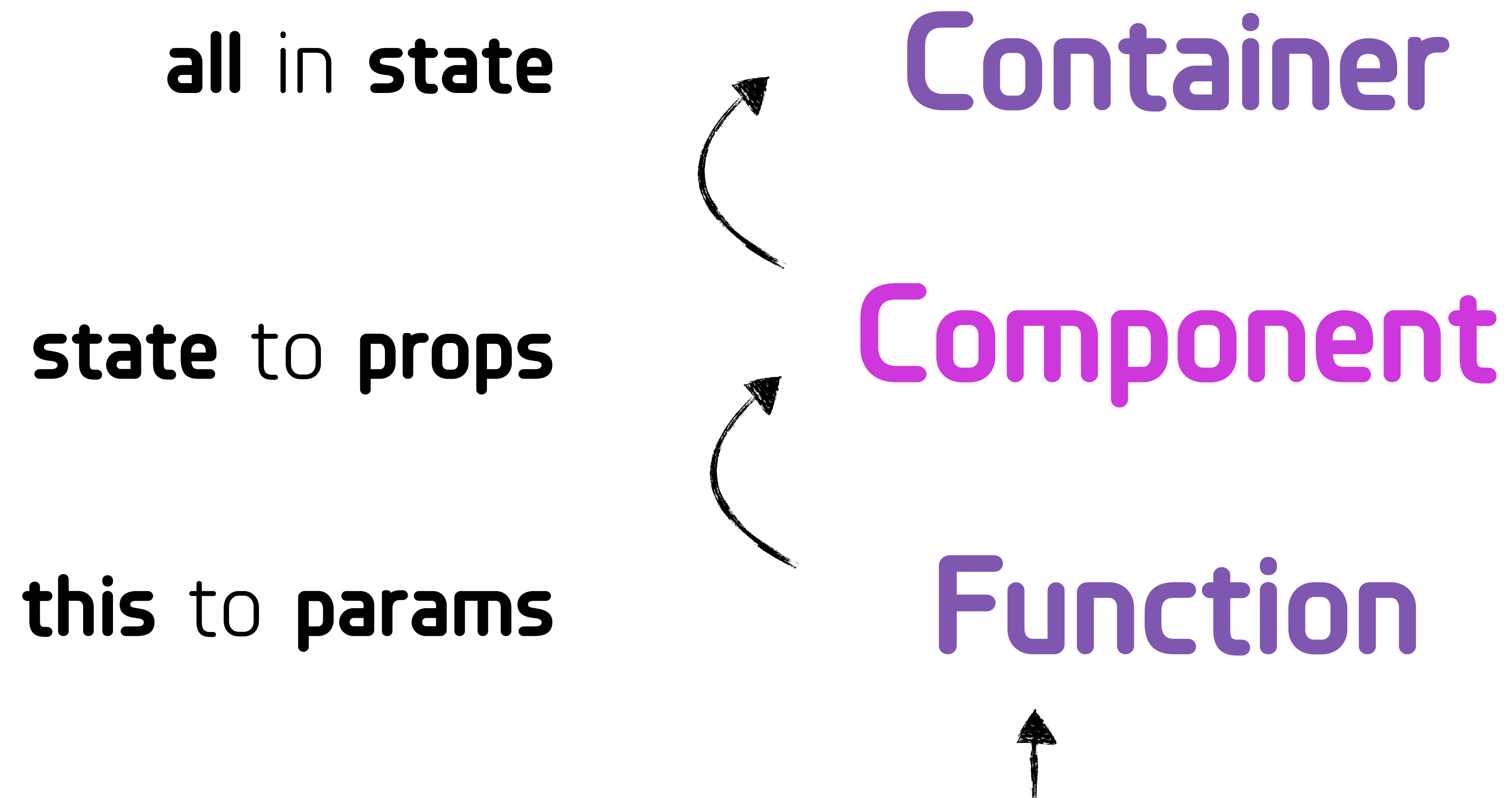
169	124		if (this.props.isHidden) return;
170	125		
171		-	const { filtersHidden } = this.state;
	126	+	const { filtersHidden } = this.props;
172	127		

This means a component should only depend on its **props**.
Everything in **state** was moved to the parent component's
state and passes down via **props**.


```
const MyComponent = ({ steps, modifier = '' }) => (  
  <div class={ modifier }>  
    ...  
  )  
  
MyComponent.propTypes = {  
  steps: PropTypes.arrayOf(  
    PropTypes.shape({  
      completed: PropTypes.bool.isRequired,  
      title: PropTypes.string.isRequired  
    })  
  ).isRequired,  
  
  modifier: PropTypes.string  
}
```

instacod.es/107140 

This is how an ideal pure component looks like.
Note that we are thoroughly describing **propTypes**.
They serve also as a documentation.



Now when all children components are pure
we can make the top-level container pure too.
Only this container is aware of the data flow.



```
import MyComponent from './my-component'

class Wrapper extends React.Component {
  constructor() {
    this.state = {
      active: false,
      list: [],
    };
  }

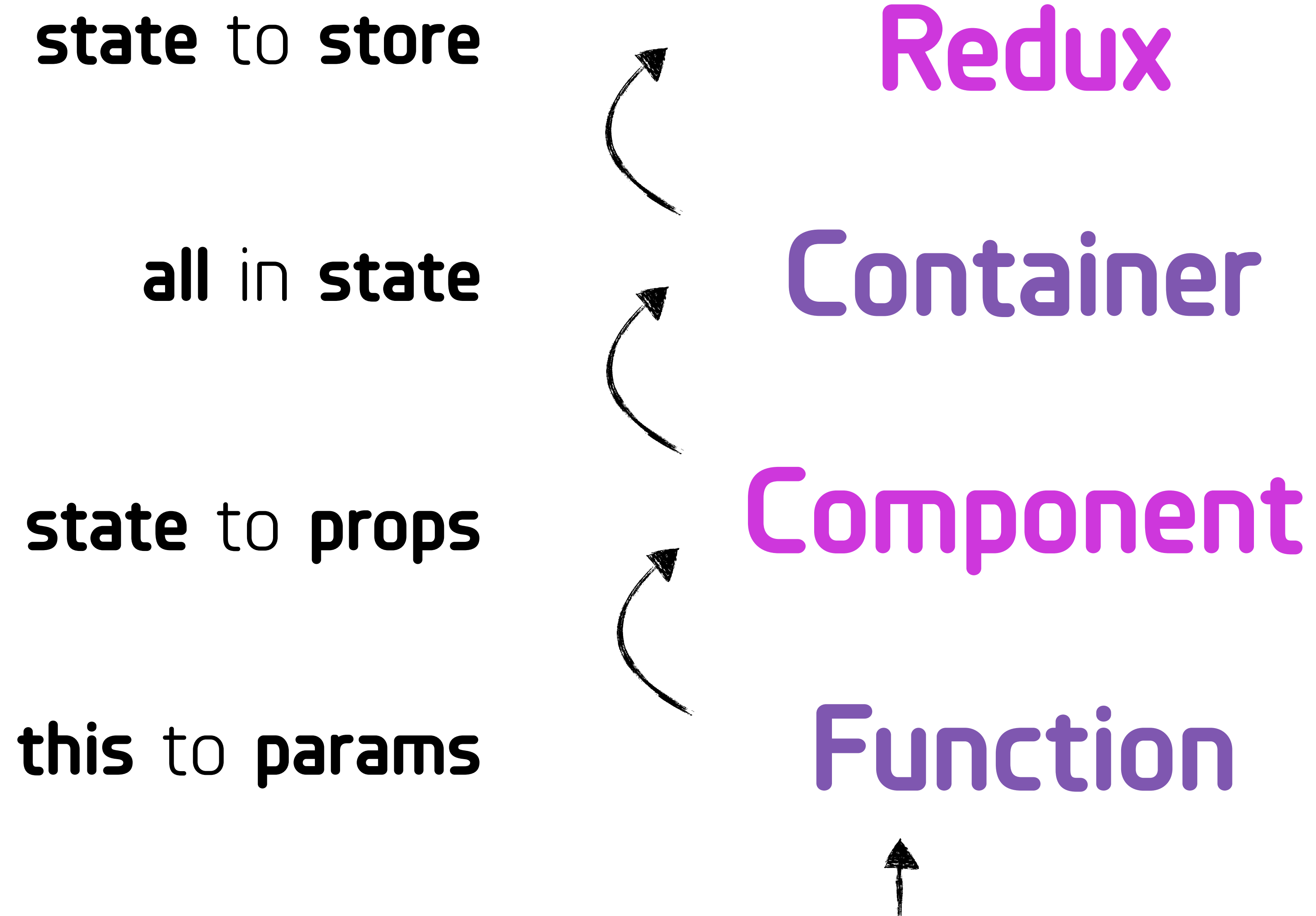
  open() { ... }
  close() { ... }

  render() {
    return (<MyComponent
      {...this.state}
      onOpen={this.open.bind(this)}
      onClose={this.close.bind(this)}
      translations={this.props.translations}
    />)
  }
}

export default Wrapper
```

instacod.es/107146 

All data is inside this container's **state**.
Modifications are possible only with provided methods.
MyComponent passes these 'actions' further.




Introducing **Redux** is now easy.
We have the data structure described.
All components keep their **APIs** (= propTypes).


```
import { connect } from 'react-redux'

import * as actions from '../actions'

export default connect(
  (state) => ({
    active: state.users.active,
    list: state.users.list,
  }), actions)(MyComponent)
```

instacod.es/107141 

We can remove the **Wrapper** and connect to Redux.
Data is now in store, actions correspond to methods.
MyComponent has not changed.

react-rails

As mentioned earlier, our app is in fact Rails app.
The **react-rails** gem enables mounting React in templates.
It also passes data from Rails to the component.


```
- state = {users: {  
    active: false,  
    list: new_users_list  
}}  
  
- :javascript  
window.__INITIAL_STATE__.push(#{state.to_json})  
  
= react_component('Wimdu.MyComponent',  
    { translations: translations },  
    { prerender: false })
```


To pass the initial state (from multiple templates) we
serialise it to JSON and append it to the array.
Component is referenced by (global) variable name.

```
import tx from 'transform-props-with'

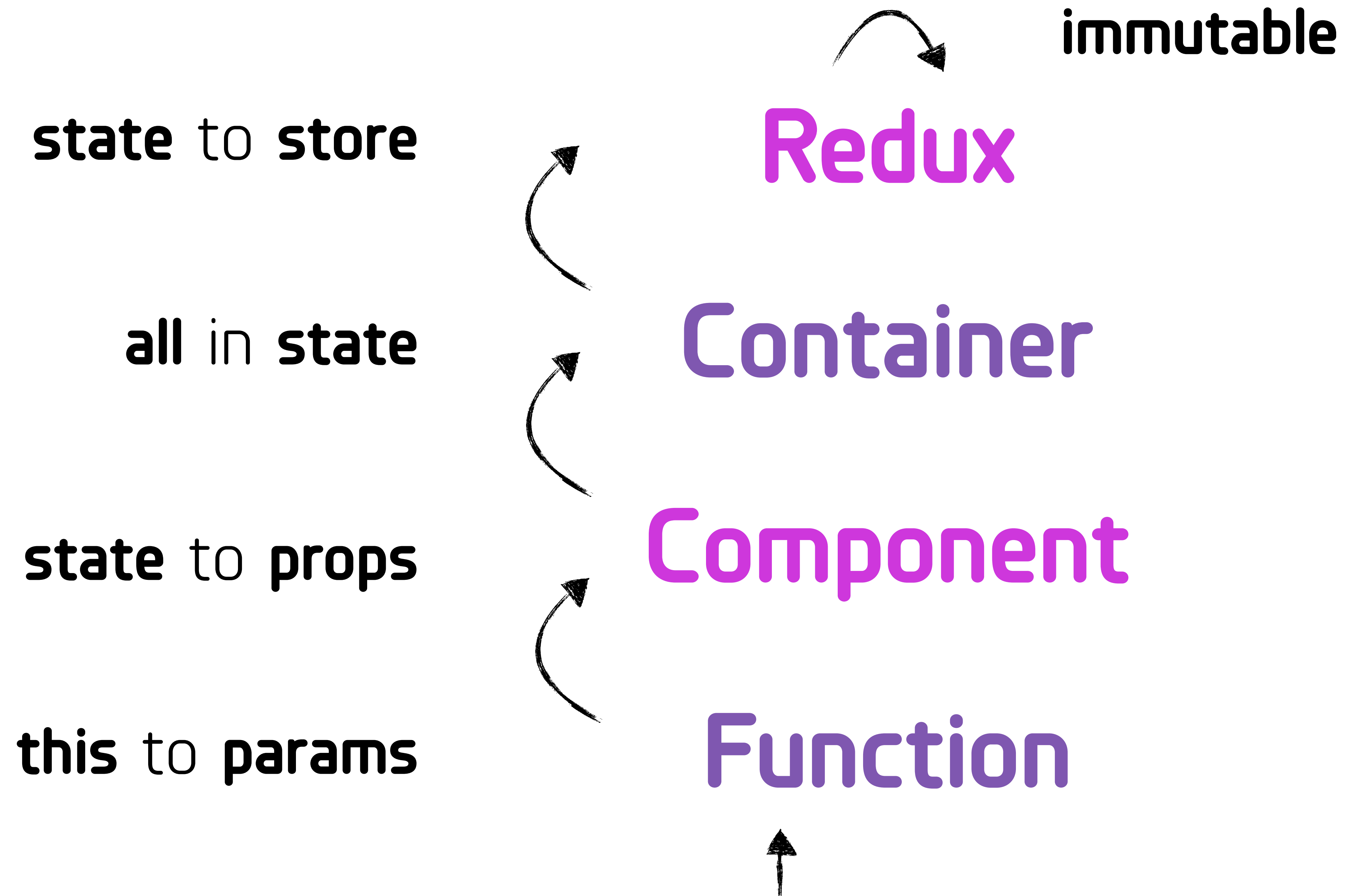
if (!window.Wimdu.store) {
  const initialState =
    Map().mergeDeep(...window.__INITIAL_STATE__)

  window.Wimdu.store = configureStore({ initialState })
}

window.Wimdu.MyComponent =
  tx({ store: window.Wimdu.store })(MyComponent)
```

instacod.es/107137 

Thanks to ImmutableJS deep merging method we
combine all partial states into one store.
We use tx to pass this store to the component.



Changing data handling in **Redux** we introduce immutable structures (e.g. ImmutableJS).
No need to touch anything else.


```
import * as actions from './actions'

const State = Record({
  active: false, list: List()
})
const initialState = new State()

export default (state = initialState, action) => {
  if (!(state instanceof State)) return new State(state)

  switch (action.type) {
    case actions.OPEN: {
      return state.set('active', true)
    }
    ...
  }

  return state
}
```


instacod.es/107142 

This is an example Redux reducer.
Thanks to **Records** we have structure consistency,
documentation, and dot access notation.


```
import { combineReducers } from 'redux'


import users from './users/reducer'
import me from './me/reducer'
...

export default combineReducers({
  users,
  me,
  ...
})
```

instacod.es/107143 

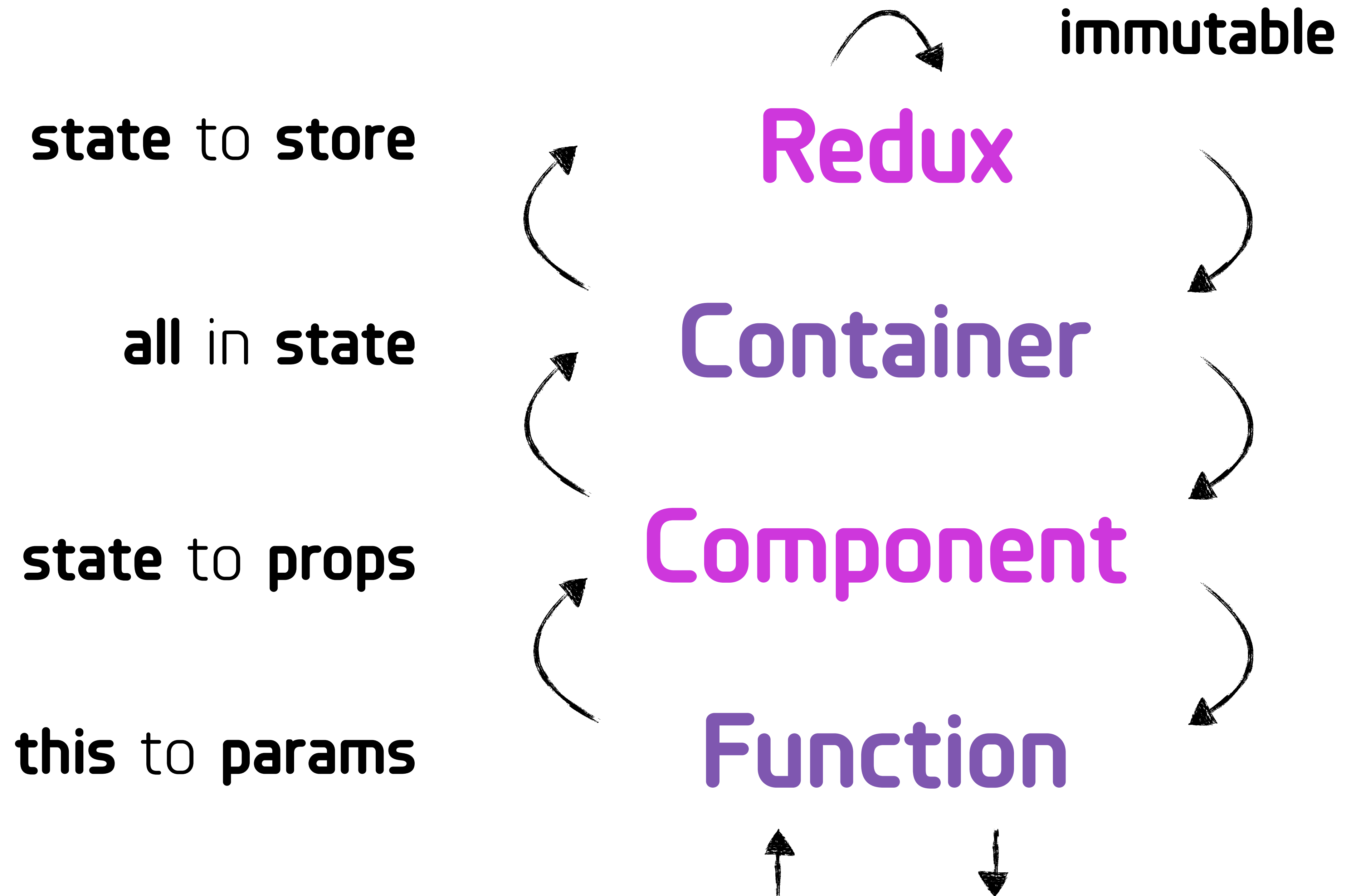
Unfortunately it is difficult to have **Record** of **Records**.
For namespacing we combine reducers the usual way.
Leafs (and only leafs) of reducer tree are **Records**.

```
export default connect(  
  (state) => ({  
    active: state.users.get('active'),  
    list: state.users.get('list').toJS(),  
  }), actions)(MyComponent);
```

instacod.es/107144 

To ensure backwards compatibility we convert immutable structures to simple JS objects at first.

We 'immutablyfy' a component passing JS to its children.



First we went UP—purifying from smallest parts.
We introduced immutability at the top and went back DOWN.

Download this slide as a one-page summary: <http://buff.ly/1XbtFpH>

A secret tip for better React and Redux apps:

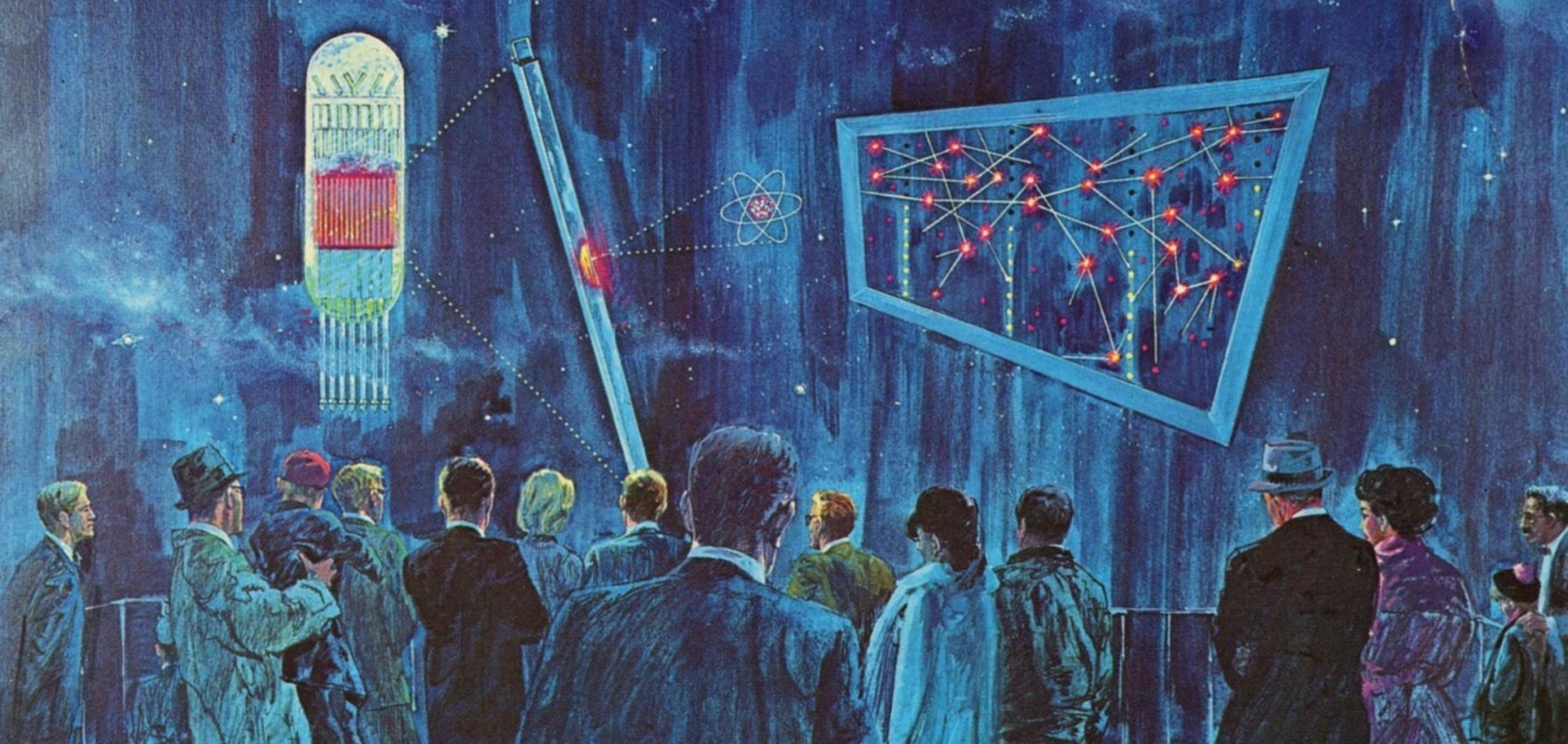
'How would I do it in Elm?'



I am fond of Elm (although not on the production now).

It helps me to decide how to structure the app.

Both React and Redux are inspired by Elm.



[@robinpokorny](#)

me@robinpokorny.com

I want to thank my team for their hard work. You made this happen!

Any question or feedback is welcomed.



This work by Robin Pokorny is licensed under a Creative Commons Attribution 4.0 International License.

Cover image was taken from 1952 Kaiser Aluminum ad:
<http://www.fulltable.com/vts/f/fut/f/world/SH536.jpg>

Image on the last slide is taken from a postcard:
Fission Room, Niagara Mohawk Progress Center, Nine Mile Point, NY