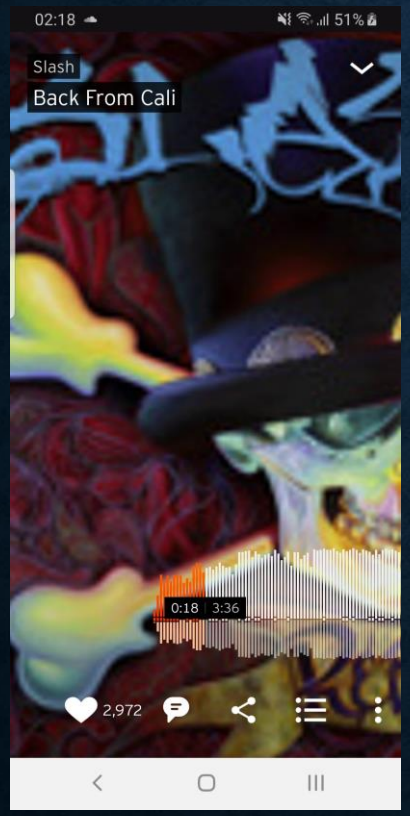
Introduction to Redux

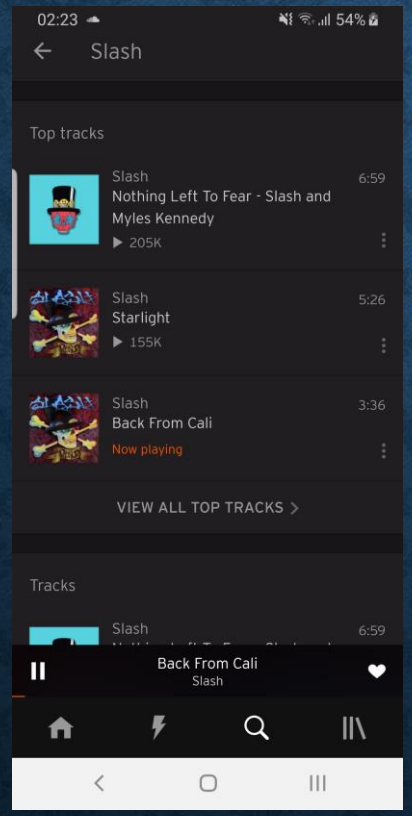
Tsapelas Christos
Java the Hutt
Software Technologies 2019

Summary

- What is Redux ?
- Why to use Redux ?
- Redux Architecture
- React example

First of all… The state!





1. Am I using the player or just scrolling?
2. The data of the app and the data of my account do they belong to the same container?
3. How all this information is synchronized?

Redux the box

- The box organizes the state in a single place
- You can ask the box what is the current state
- You can describe a change in the state to the box
- The box will notify when its state changes

Three principles of Redux

1. The state of whole application is stored in a single Javascript object, called the ***store*** (Single source of Truth)
2. The state is read-only (immutable). A state can change by describing a change with another Javascript object, called ***action***
3. Changes are performed by pure functions, called ***reducers***. A reducer accepts a state and an action and returns a new state.

Actions

- Actions are plain JavaScript objects
- Responsible for sending data from app to the store. One and only source of information of the store.
- Signature property is the *type property*
- It is a good practice to pass as little data as possible
- The functions responsible for creating actions are called, ***Action creators***

Action example

```
/*
 * action types
 */

export const ADD_TODO = 'ADD_TODO'
export const TOGGLE_TODO = 'TOGGLE_TODO'
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER'
```

Copy

```
/*
 * action creators
 */

export function addTodo(text) {
  return { type: ADD_TODO, text }
}

export function toggleTodo(index) {
  return { type: TOGGLE_TODO, index }
}

export function setVisibilityFilter(filter) {
  return { type: SET_VISIBILITY_FILTER, filter }
}
```

Reducers

- Given an action, reducers specify how the app's state changes
- Reducer is a pure function that takes the previous state and an action and returns the next state
- State is **NOT** mutated. Instead, a copy is created with the updated values
- A fundamental pattern in Redux apps is the ***reducer composition.*** We can write reducers that accept a whole state, and each reducer knows how to update just a slice of that state

Reducer example

Copy

```
import { combineReducers } from 'redux'
import {
  ADD_TODO,
  TOGGLE_TODO,
  SET_VISIBILITY_FILTER,
  VisibilityFilters
} from './actions'
const { SHOW_ALL } = VisibilityFilters

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}
```

Reducer example

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

Reducer example

```
const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Store

- The store is the core of the Redux library and it wires the three principles. Is the object that brings together the *actions* and the *reducers*
- Three main methods:
  1. getState
  2. dispatch
  3. subscribe
- Built in *createStore* function to create a store and import your *reducers*

Store

- getState – Allows access to a state
- Dispatch – Allows a state to be updated for a given action (dispatch(action))
- Subscribe – registers listeners and handles unregistering of listeners (subscribe(lstener))

All together

```
import {
  addTodo,
  toggleTodo,
  setVisibilityFilter,
  VisibilityFilters
} from './actions'

// Log the initial state
console.log(store.getState())

// Every time the state changes, log it
// Note that subscribe() returns a function for unregistering the listener
const unsubscribe = store.subscribe(() => console.log(store.getState()))

// Dispatch some actions
store.dispatch(addTodo('Learn about actions'))
store.dispatch(addTodo('Learn about reducers'))
store.dispatch(addTodo('Learn about store'))
store.dispatch(toggleTodo(0))
store.dispatch(toggleTodo(1))
store.dispatch(setVisibilityFilter(VisibilityFilters.SHOW_COMPLETED))

// Stop listening to state updates
unsubscribe()
```

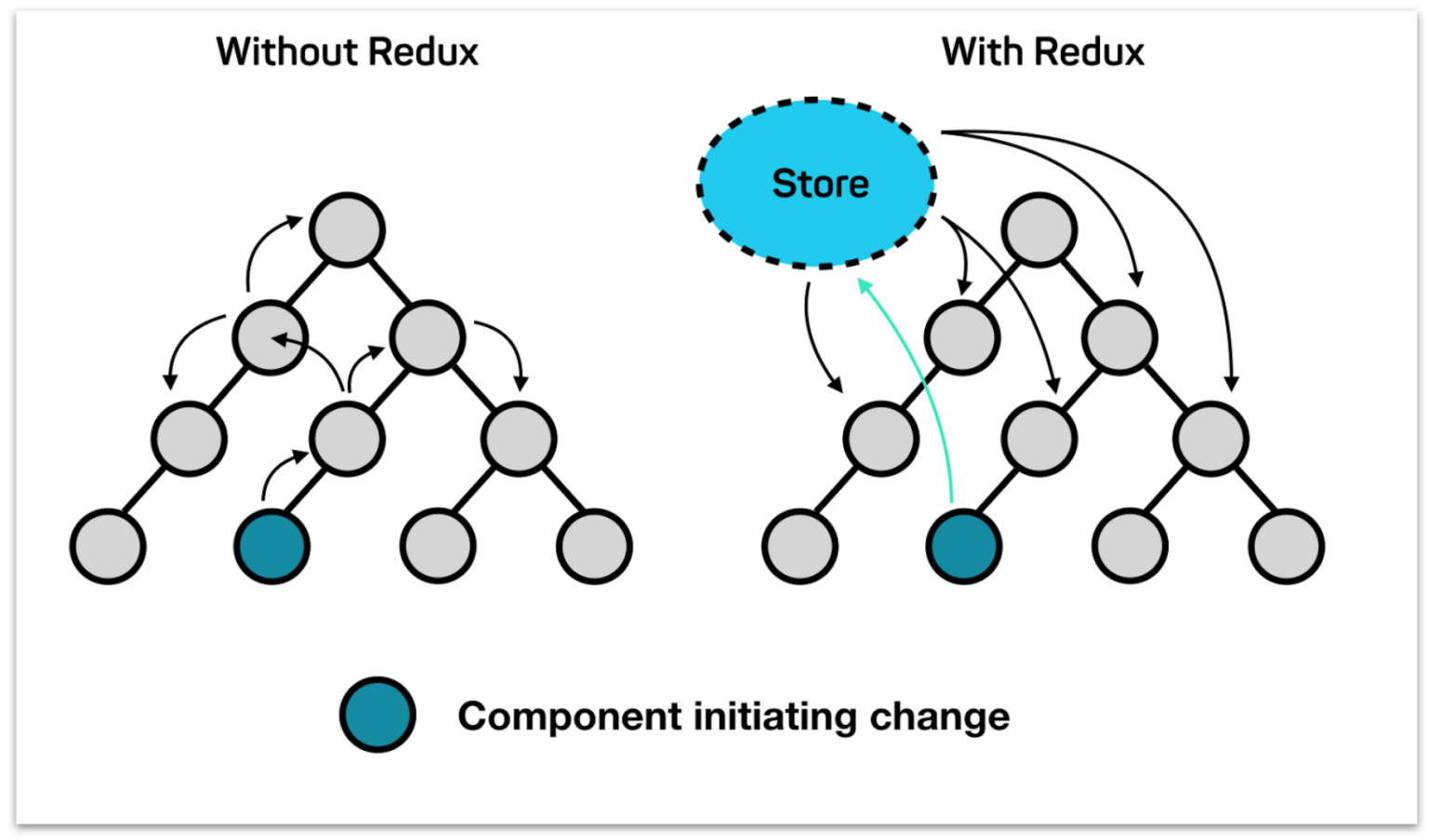
Why Redux?

- Is a standalone library that can be used with any UI layer or framework, such as React, Angular, Vue, Ember, etc.
- Same piece of application state needs to be mapped to multiple container components, For example, session state.
- Global components can be accessed from anywhere
- No need for too many props to be passed through multiple hierarchies of components

Why Redux?

- State management using setState is bloating the component. Reducers break up the code and make it more readable and maintainable
- Caching page state
- Very helpful, for maintaining very big and fast-growing apps

State hierarchy



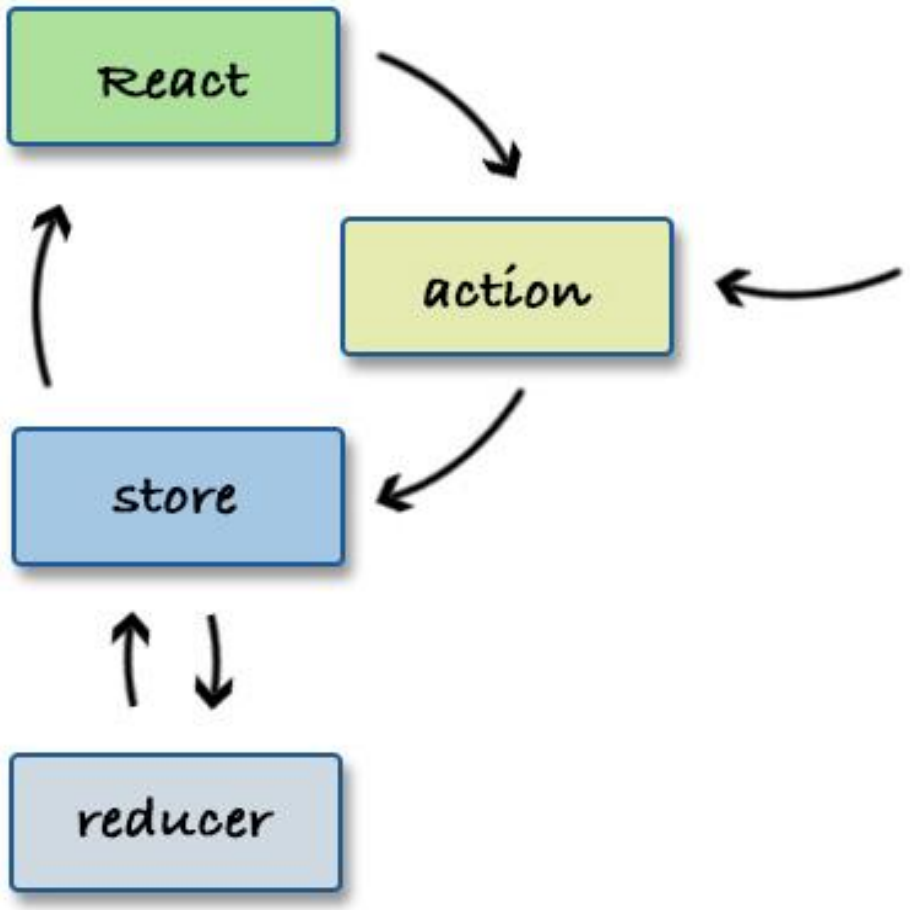
But…

- High learning curve
- Increased layering complexity in writing state manipulation logic, such as actions and reducers
- Writing and managing state containers/wrappers for each component to avoid o mass of nested state containers

Redux Dataflow

- Redux architecture is about unidirectional dataflow
- The data lifecycle follows four steps:
  1. An action is triggered, through dispatch(action)
  2. The Redux store calls the reducer we have provided for handling the specific action
  3. The root reducer may combine the output of multiple reducers into a single state tree
  4. The Redux store saves the complete state tree returned by the root reducer

Redux Dataflow



React and Redux example

index.js > ...

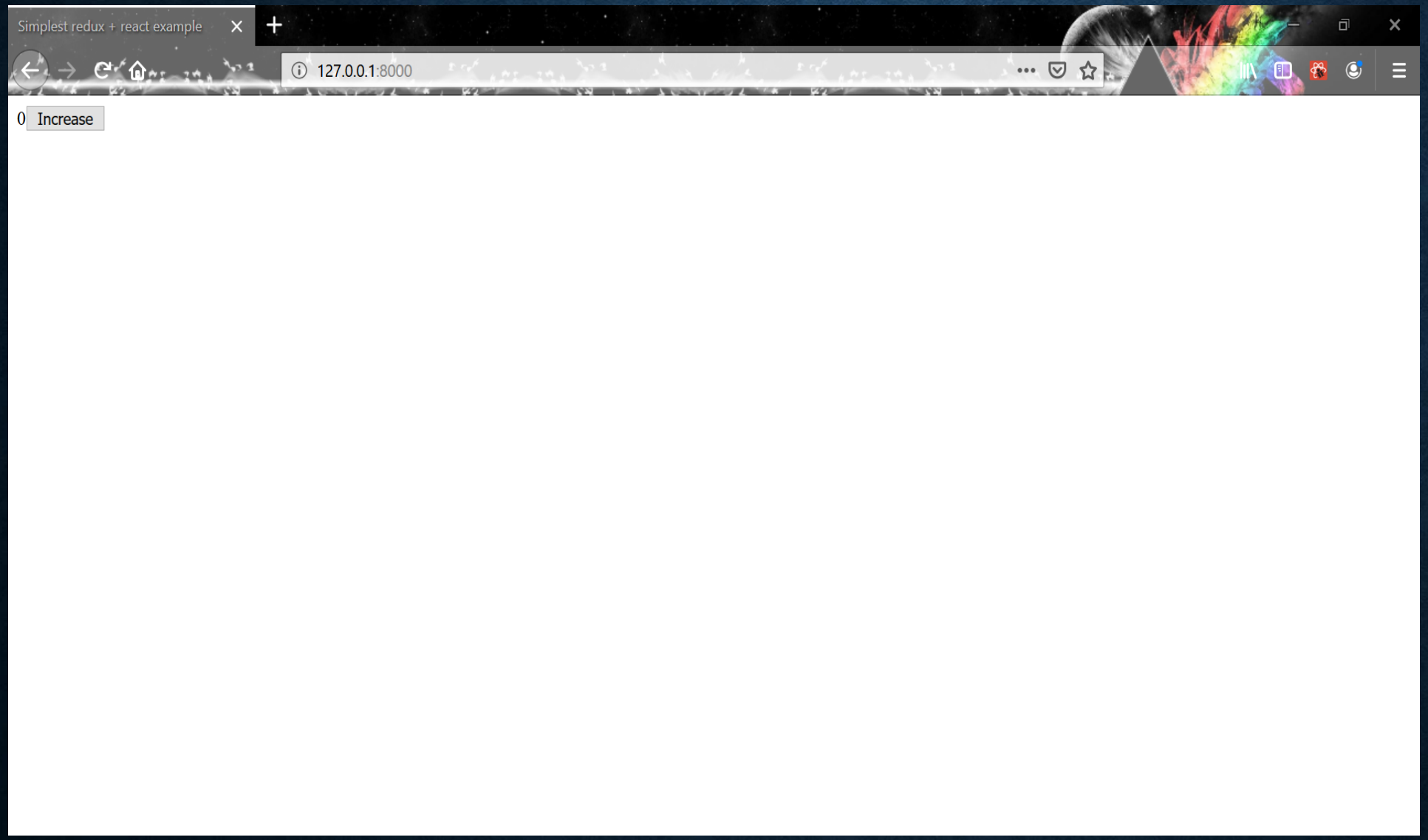
```js
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import ReactDOM from 'react-dom'
import { createStore } from 'redux'
import { Provider, connect } from 'react-redux'

// React component
class Counter extends Component {
  render() {
    const { value, onIncreaseClick } = this.props
    return (
      <div>
        <span>{value}</span>
        <button onClick={onIncreaseClick}>Increase</button>
      </div>
    )
  }
}

Counter.propTypes = {
  value: PropTypes.number.isRequired,
  onIncreaseClick: PropTypes.func.isRequired
}

// Action
const increaseAction = { type: 'increase' }

// Reducer
function counter(state = { count: 0 }, action) {
  const count = state.count
  switch (action.type) {
    case 'increase':
      return { count: count + 1 }
    default:
      return state
  }
}
```

React and Redux example

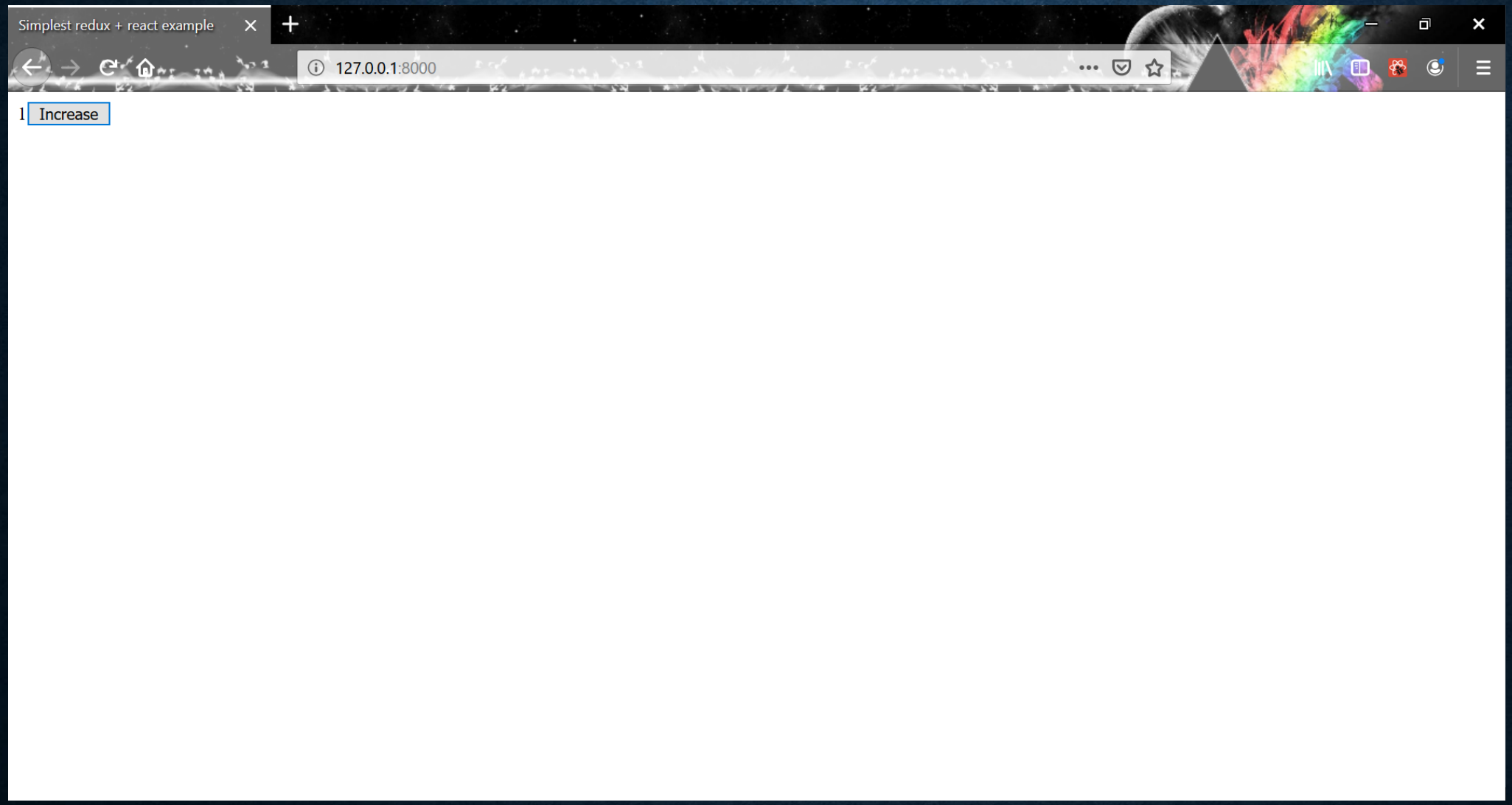
index.js > ...

```js

// Store
const store = createStore(counter)

// Map Redux state to component props
function mapStateToProps(state) {
  return {
    value: state.count
  }
}

// Map Redux actions to component props
function mapDispatchToProps(dispatch) {
  return {
    onIncreaseClick: () => dispatch(increaseAction)
  }
}

// Connected Component
const App = connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter)

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)

```

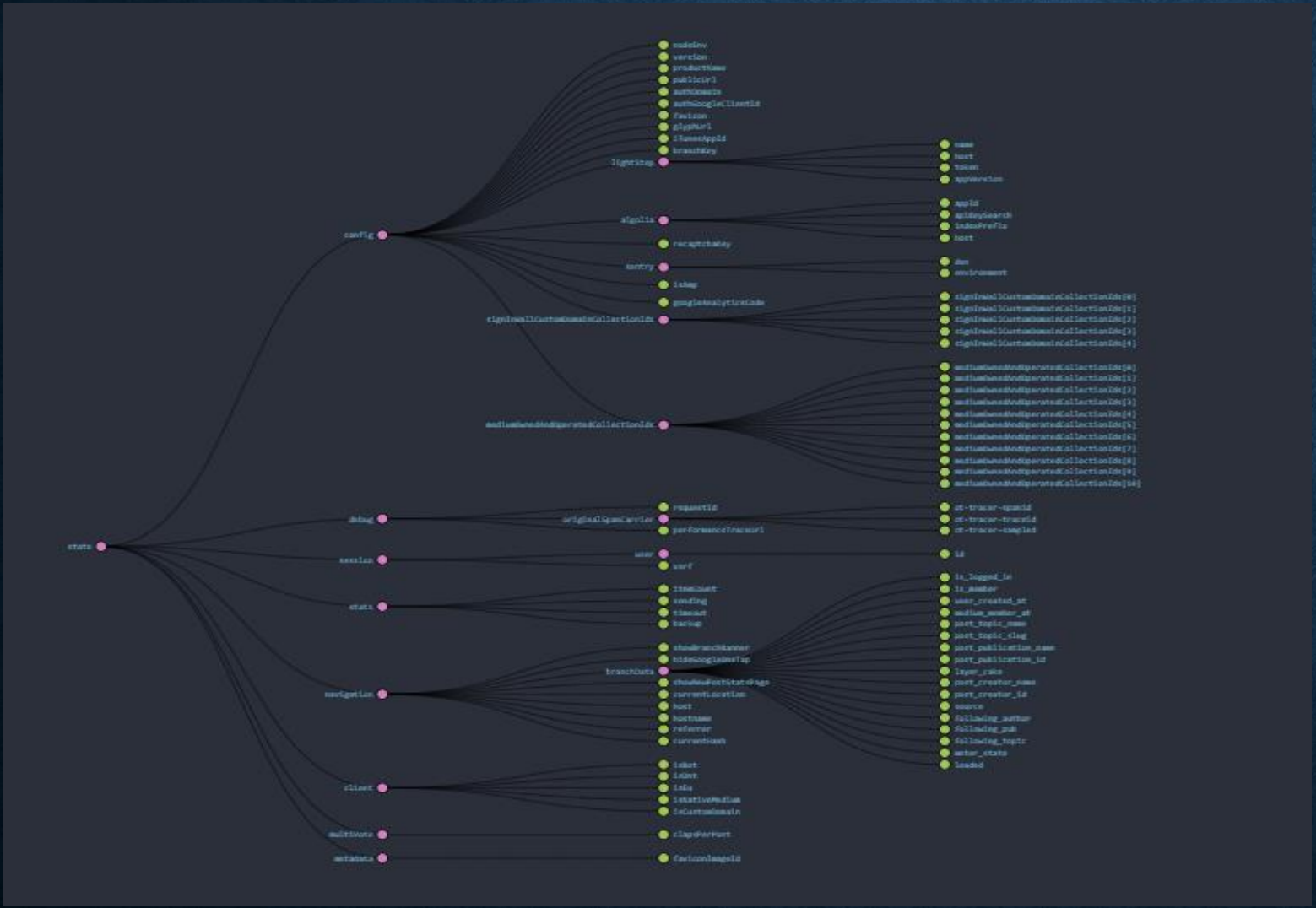
React and Redux example



React and Redux example



React and Redux example



THANK YOU