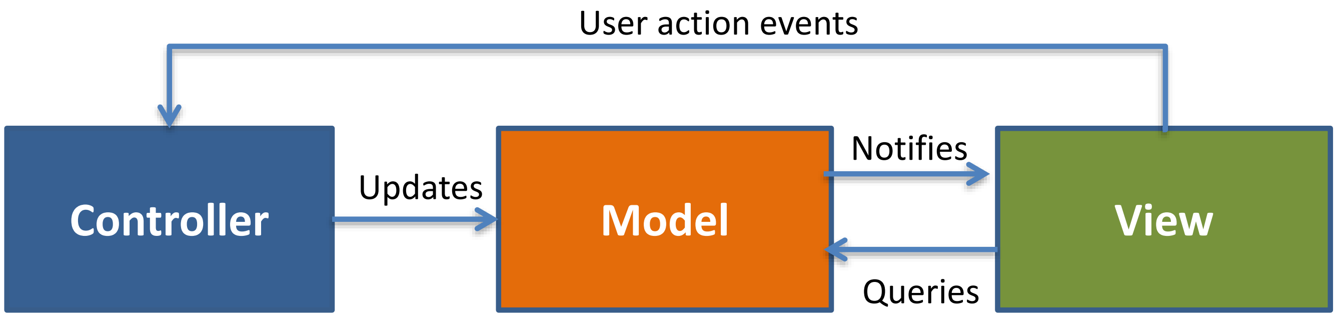
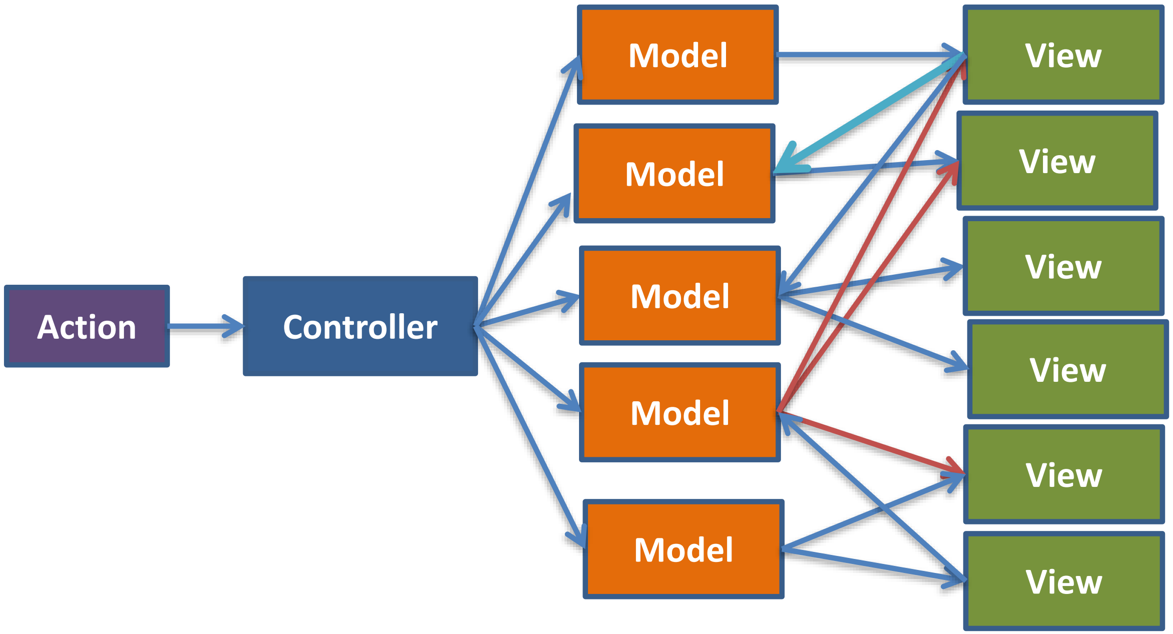
Flux

Enrique Barra

Two way data binding

- These are frameworks such as Angular, Backbone, Ember, …
- Also called *mutation*





Facebook had a recurring bug in the notification and chat panels.

Flux

- Flux is an architecture that they use at Facebook for their client apps

- It perfectly fix with React components

- The most important concept is that data flows in one direction

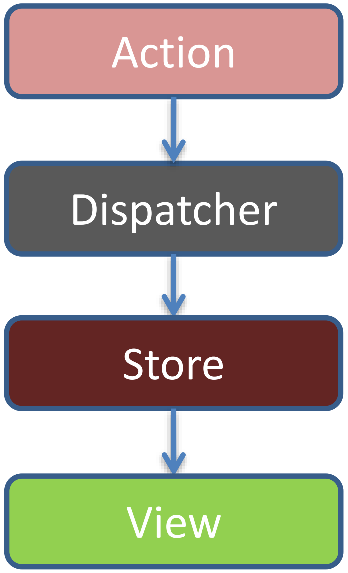
- It isn't a framework, it is a pattern, an architecture

- More info at: http://facebook.github.io/flux/

Architecture

- It consists of DISPATCHER/S, STORES and VIEWS (React Components)
- It is not the same as MVC (Model, View, Controller)
- This architecture also has controllers, because the views are sometimes called controller-views

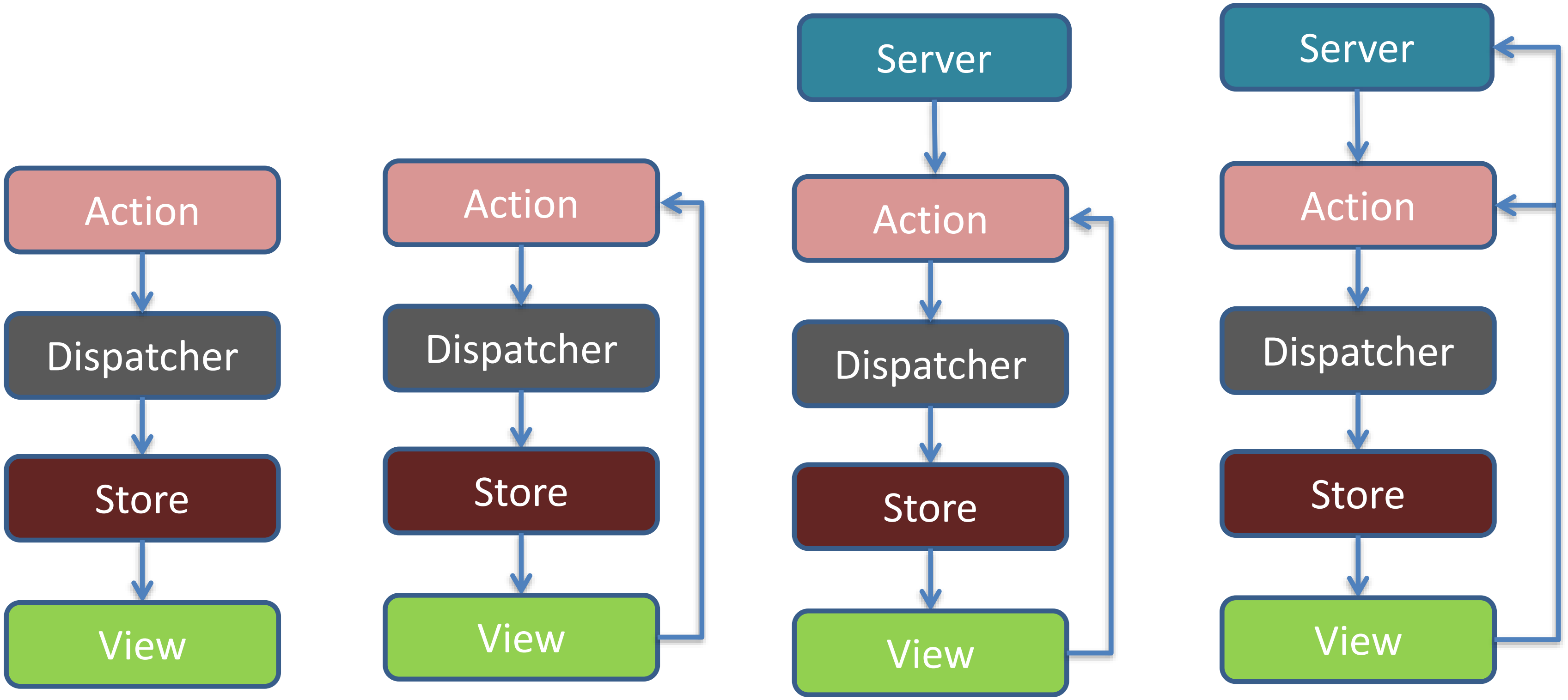
- Advantages:
- Unidirectional data flow and separation of concerns make easy to think about the app, trace bugs
- Different parts of the app will be decoupled



Flux

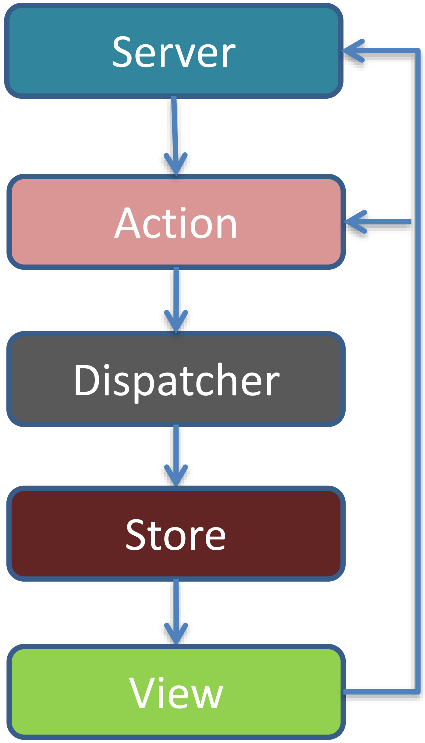
Flux

- Architecture **of the whole app**, that enforces that the data goes only in one direction

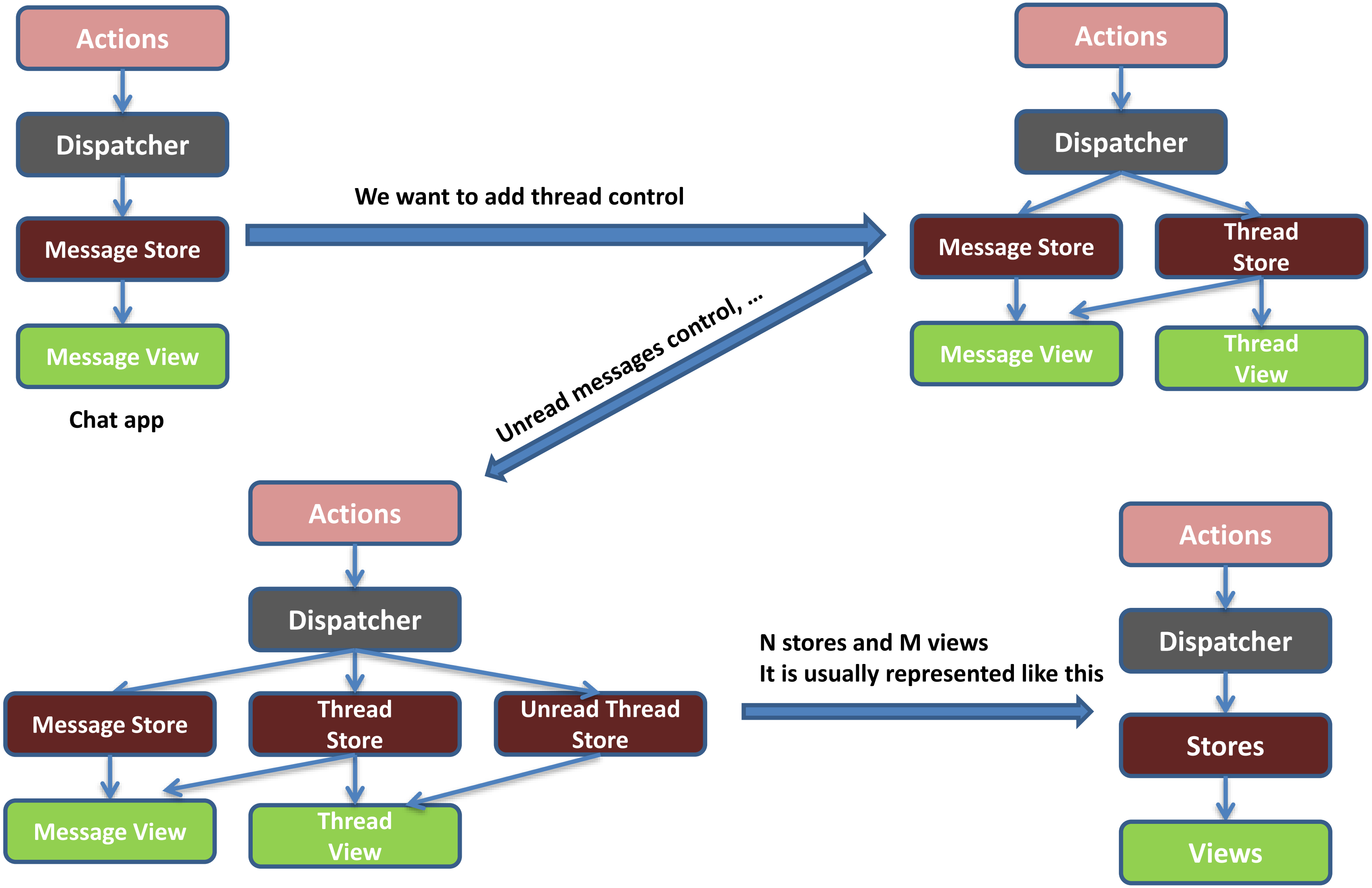


Architecture – explanation and components

- The user by interacting with the views generates **actions**, that are passed to the dispatcher. Also the server can generate actions.
- This actions are objects that are created with some methods action creators
- The dispatcher calls the callback/s that the stores registered according to the logic that they execute
- The stores emit a **“change” event**
- The views receive this event that was previously registered and that tells them that something has changed and they ask for the new state to the store
- The views perform a setState and are updated. The setState causes a new render of the component and their descendants



Flux – how it scales



Actions

Dispatcher

Message Store

Message View

Chat app

We want to add thread control

Actions

Dispatcher

Message Store

Thread Store

Message View

Thread View

Unread messages control, …

Actions

Dispatcher

Message Store

Thread Store

Unread Thread Store

Message View

Thread View

N stores and M views
It is usually represented like this

Actions

Dispatcher

Stores

Views

Flux

- It is not open source, because it is an architecture, not a software

- Facebook has released their dispatcher and utils
  - http://facebook.github.io/flux/docs/dispatcher.html
  - https://github.com/facebook/flux

- Different Flux implementations:
  - Redux - https://github.com/reduxjs/redux
  - Alt - http://alt.js.org/
  - McFly - https://github.com/kenwheeler/mcfly
  - …

- Or we can implement our own architecture step by step using Facebook dispatcher and tools (it is not very difficult)

Flux

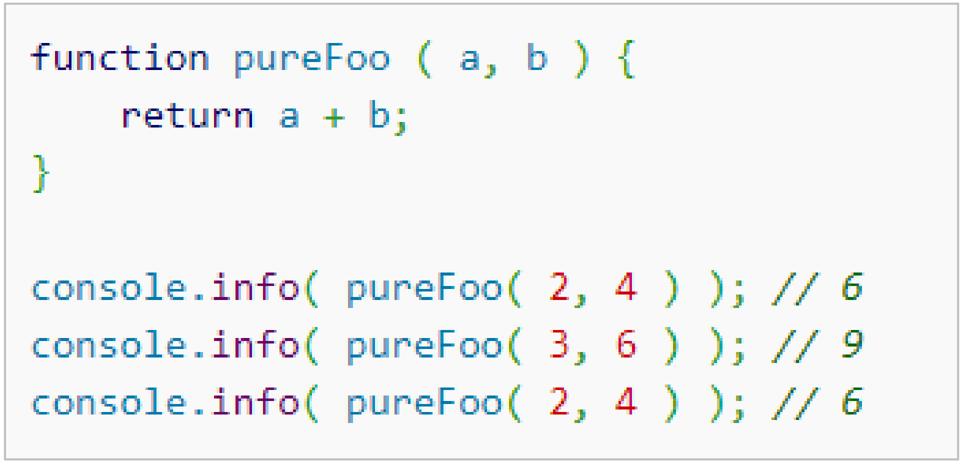
Enrique Barra

Redux Introduction

Enrique Barra

Pure functions

- They operate using only their arguments, not any other element (function, variable, …) outside them
- In a more formal way:
  - Given the same argument values, the pure function will return always the same result
  - The pure function has no side effect



```
function pureFoo ( a, b ) {
    return a + b;
}

console.info( pureFoo( 2, 4 ) ); // 6
console.info( pureFoo( 3, 6 ) ); // 9
console.info( pureFoo( 2, 4 ) ); // 6
```

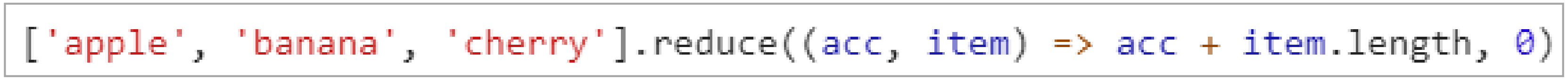
- More info: https://en.wikipedia.org/wiki/Pure_function

Reduce

- It comes from the mapReduce programming model
- The “reducer” function takes the previous output (*acc*) and the next value (item) and calculates the next output

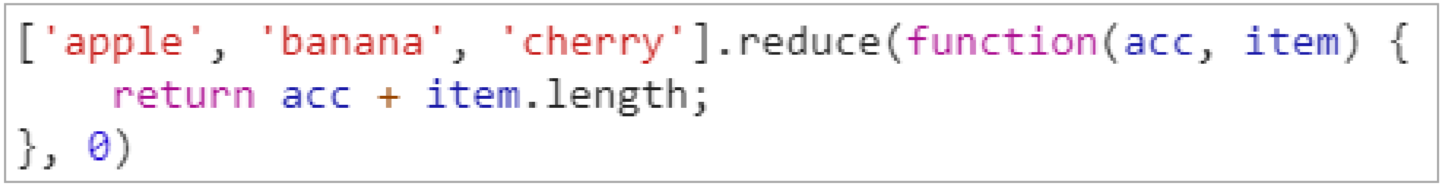
Array.prototype.reduce(**function reducer(acc, item)**, ?initialValue)

- Example that counts the character of all the words in an array:
  - Reduce done with “arrow function”:



```
['apple', 'banana', 'cherry'].reduce((acc, item) => acc + item.length, 0)
```

  - Reduce “traditional way”:



```
['apple', 'banana', 'cherry'].reduce(function(acc, item) {
    return acc + item.length;
}, 0)
```

Redux

- **Redux is a library that implements the Flux pattern (with some variations)**

- Redux is defined as a “Predictable state container for JavaScript apps”

- Helps to write apps that behave in a consistent way and are easy to test

- It is in charge of decoupling the global state of the app from the visual part (i.e. the components)

- Redux is a very small library (2KB approx.). The API is barely 5 functions, and more important it is vanilla JavaScript, so it is framework agnostic, so it can be used with any library or framework such as Angular, Polymer, React, etc.

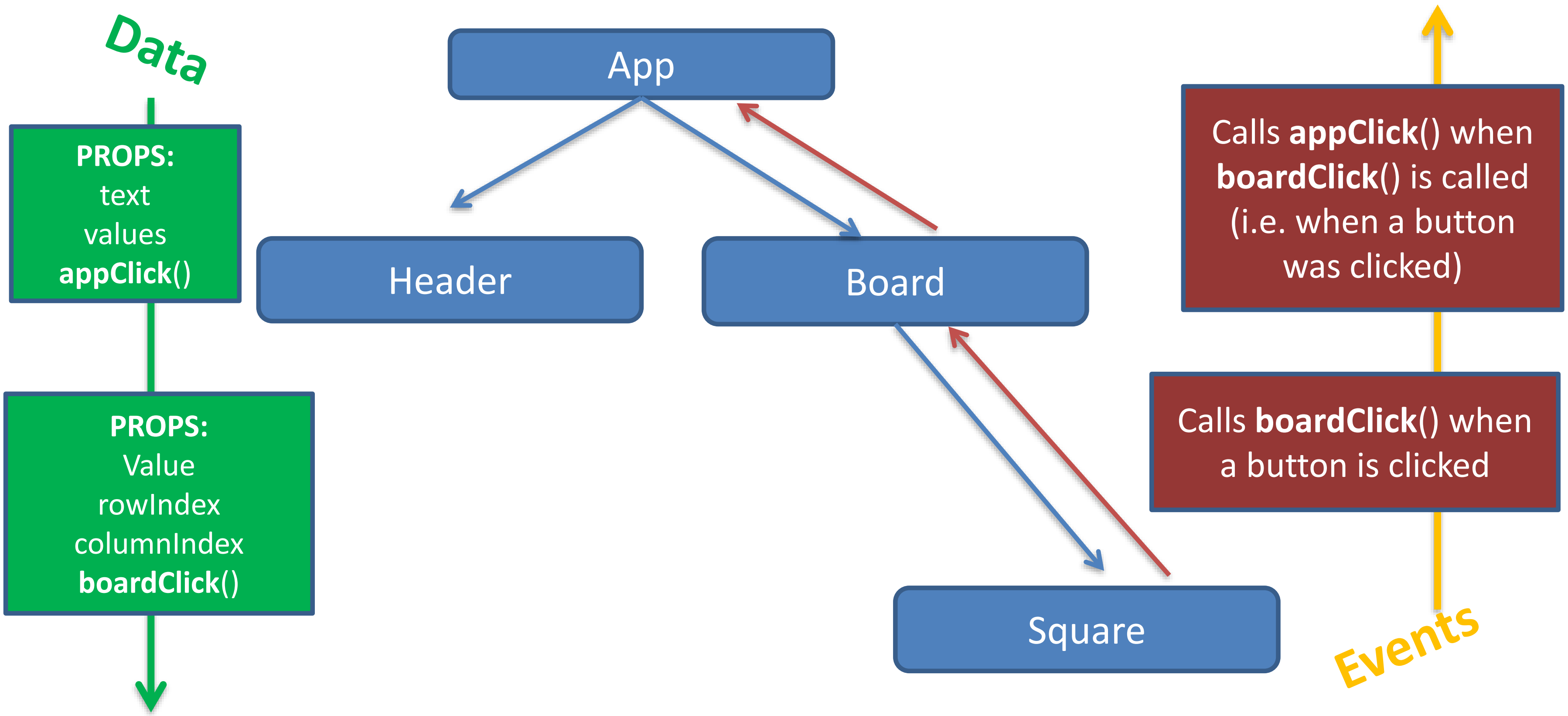
Redux Three principles

- **1. Single source of truth**
  - The state of your whole application is stored in an object tree within a single store
  - Makes easy to implement some functionalities like undo/redo that are usually very difficult to implement
  - Makes easy the communication with the server and the storage of this state

- **2. State is read-only**
  - The only way to change the state is to emit an action, an object describing what happened
  - Everything predictable
  - There are no subtle race conditions to watch out for

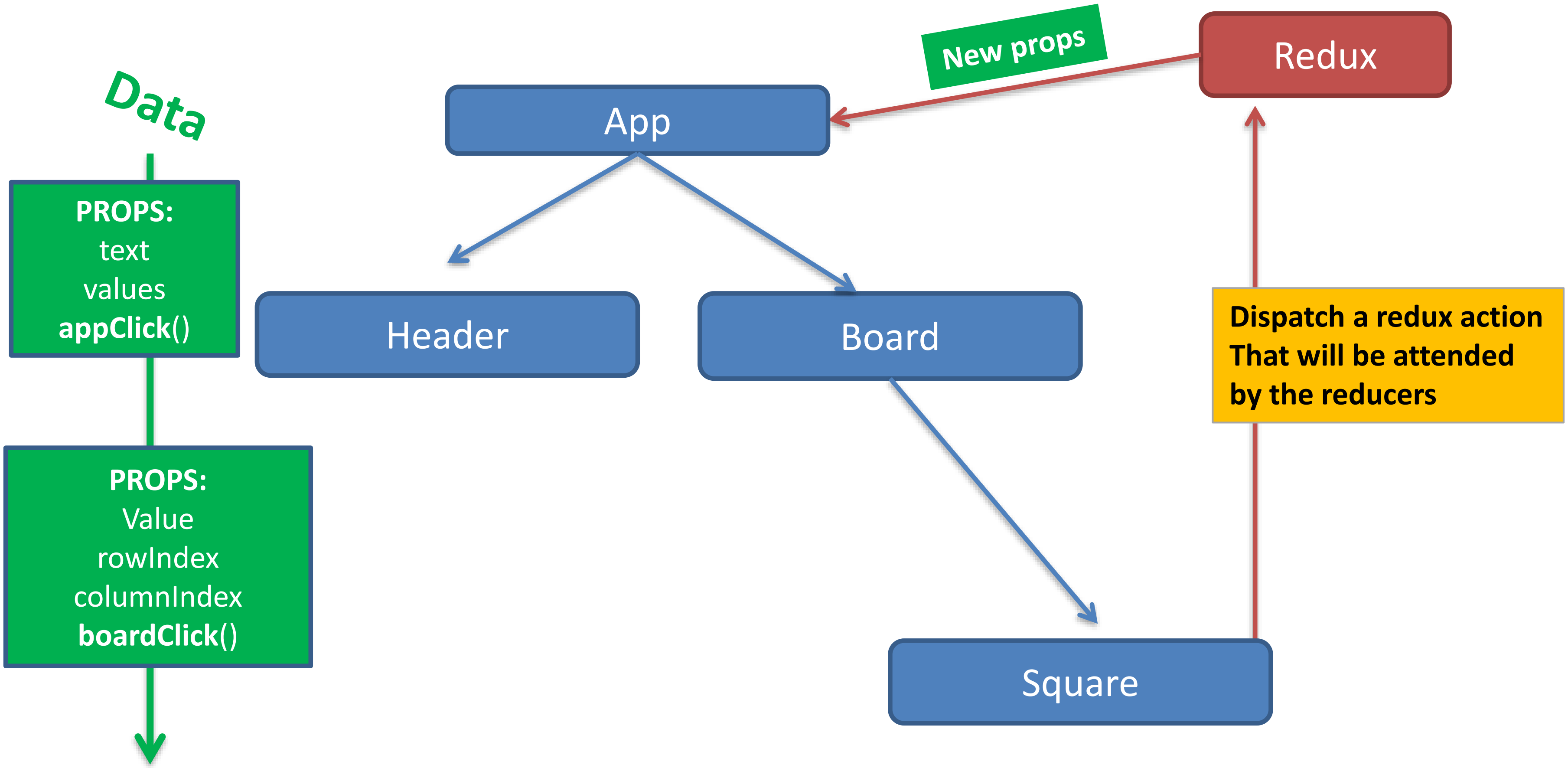
- **3. Changes are made with pure functions**
  - To specify how the state tree is transformed by actions, you write pure reducers
  - Reducers are just pure functions that take the previous state and an action, and return the next state
  - Reducers return new state objects, instead of mutating the previous state
  - Remember that a “reducer” is a function that takes the previous output (*acc*) and the next value (item) and calculates the next output

Without Redux


Data
PROPS:
text
values
appClick()
PROPS:
Value
rowIndex
columnIndex
boardClick()
App
Header
Board
Square
Calls appClick() when boardClick() is called (i.e. when a button was clicked)
Calls boardClick() when a button is clicked
Events

- Without Redux, the child component calls his parent to tell him that an event ocurred
- Then the same process occur up to the top of the app, where the logic takes place

With Redux



- With Redux, the child component **dispatchs** an **action** that will be attended by Redux
- Then Redux will pass new props to the App and it will be re-rendered if needed

Bibliography and resources

- Redux Core Concepts
  http://redux.js.org/docs/introduction/CoreConcepts.html

- Levelling up with React Redux
  https://css-tricks.com/learning-react-redux/

- Learn Redux
  https://learnredux.com/

- Getting started with Redux
  https://egghead.io/courses/getting-started-with-redux

Redux

Redux Introduction

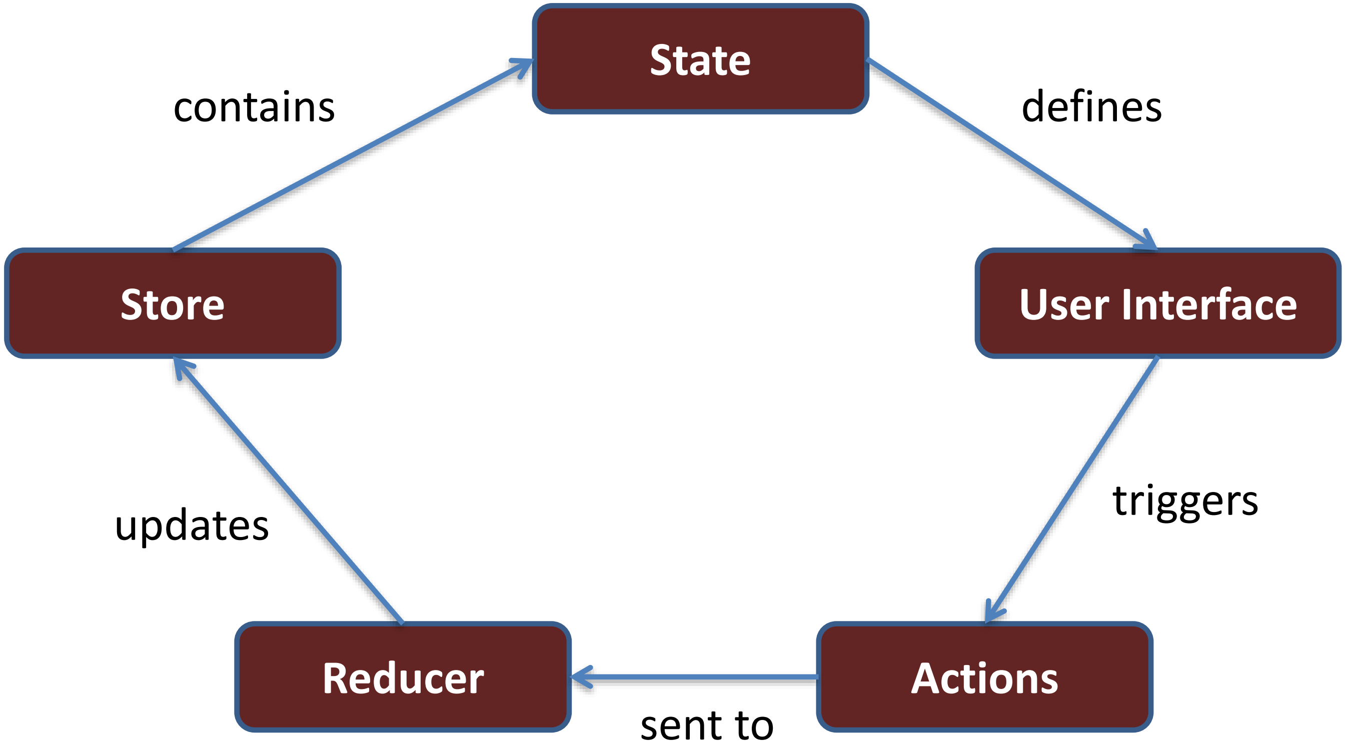
Enrique Barra

Redux

Redux Lifecycle and API

Enrique Barra

Redux lifecycle



State

defines

User Interface

triggers

Actions

sent to

Reducer

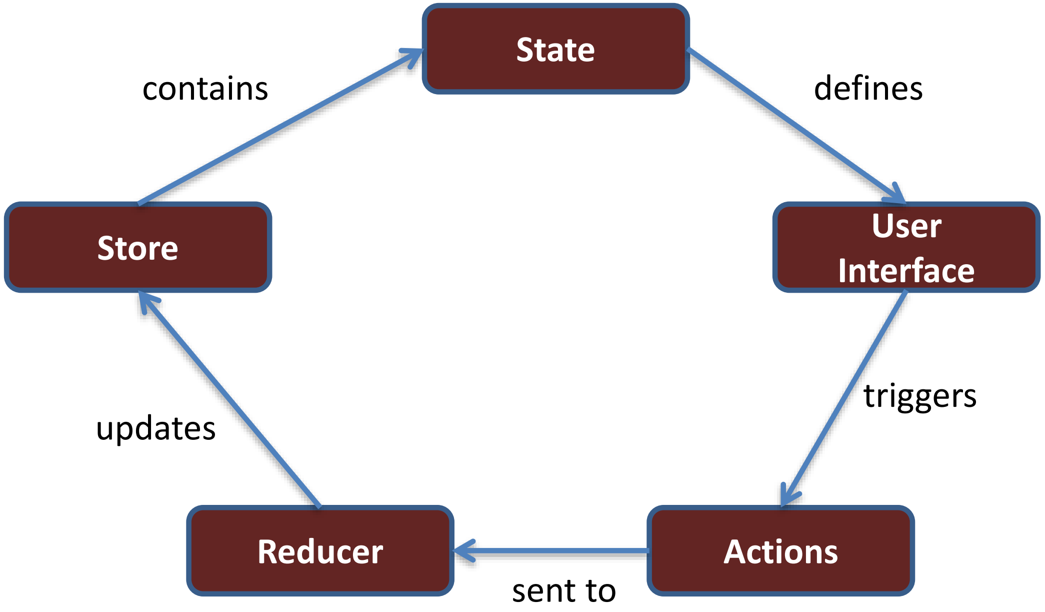
updates

Store

contains

Redux Elements

- State
- Actions
- Reducers
- Store

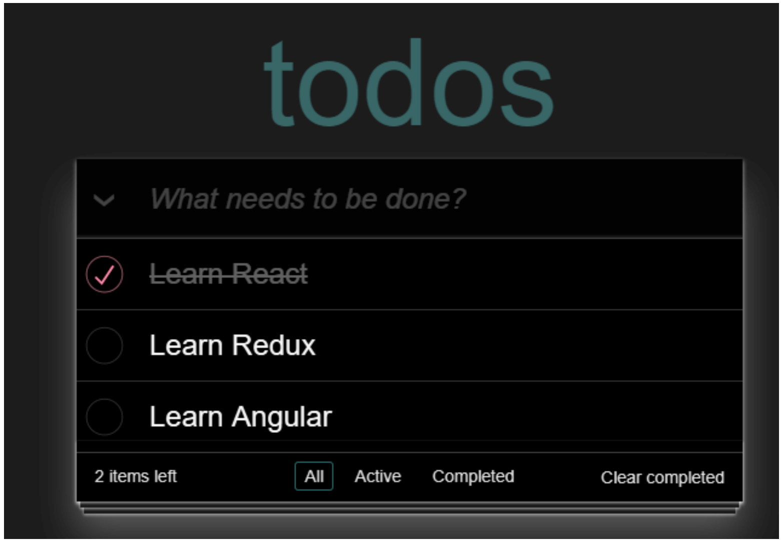


- API (to create the store, combine reducers, etc.)

State (immutable)

- The state of the app is a JS object (immutable, not directly modified)
- Example: TO-DO app

```
{
     todos: [
    { text: ‘Learn React',
      completed: false },
    { text: ‘Learn Redux',
      completed: false },
    { text: ‘Learn Angular',
      completed: true }
     ],
     visibilityFilter:  'SHOW_ALL'
}
```



todos

What needs to be done?

- ☑ ~~Learn React~~
- ○ Learn Redux
- ○ Learn Angular

2 items left

Actions

- The state can **only** be modified triggering actions
- Action: JS object that describes the change to perform on the state
- Actions have a “type” attribute (mandatory) and the rest of the attributes are optional and will depend on what the action does
- Example:

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ACTIVE' }
```

- Usually created with functions called “action creators”

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

Reducers

- **Pure functions** that apply **actions** on the **state**
- They take the previous state and an action and return the new state
- Things that we cannot do inside a reducer:
  - Modify its arguments
  - Perform tasks with side effects such as API calls or route changes
  - Call a non pure function, for example Date.now() or Math.random()
- We will usually structure it in several reducers:
  - Each reducer deals with a specific part of the state => it is easy to manage apps with a lot of information
  - The reducer takes as param a piece of the old state and the action to apply and returns a equivalent part of the new state

Single Reducer structure

State param is the piece of the state that corresponds to this reducer, not the full or global state

In the **action param** comes all the needed information to modify the state, specially the type of action that it is

```
// import {..} …….

function myReducer(state = “DEFAULT_STATE”, action) {
  switch (action.type) {
    case 'ACTION_NAME':
      let newState = Object.assign([], state);
      // … Modify newState
      return newState; // Return the modified state
    default:
      return state;
  }
}
export default myReducer;
```

Example of Reducer

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

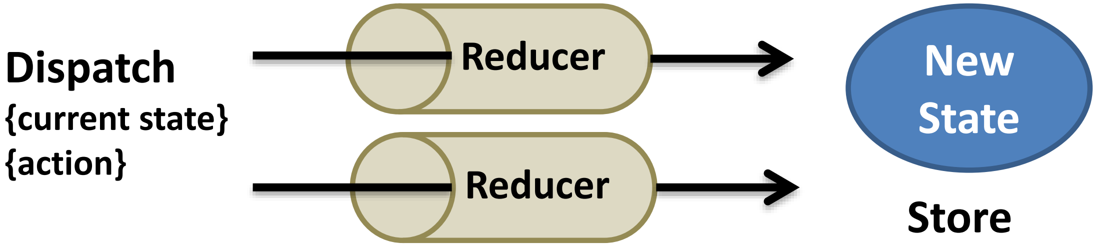
Only affects the piece of state corresponding to the “todos”

In the default case they return the same piece of state

Store

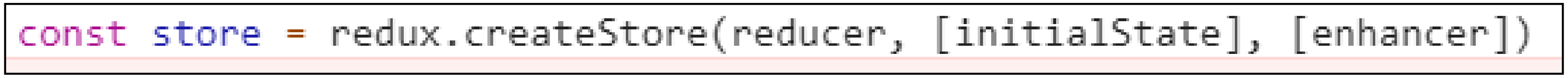
- The store is the object that joins state and reducers
- The store has the next responsibilities:
  - Contains the **full state** of the app
  - Allows the access to the state via **getState()**
  - Allows state updating via **dispatch(action)**
  - Register the listeners via **subscribe(listener)**
  - Manages the removal of the listeners via the return of the function **subscribe(listener)**

- It is important to notice that there is only one store in the Redux app. When we want to split the app logic to manage data we use several reducers instead of several stores



API (I)

- **createStore()**
  - This function creates the central store where the global state is stored
  - The function receives as param a *reducer* and optionally a initial *state* and an *enhancer* that is used to add *middlewares*. It returns the created store.

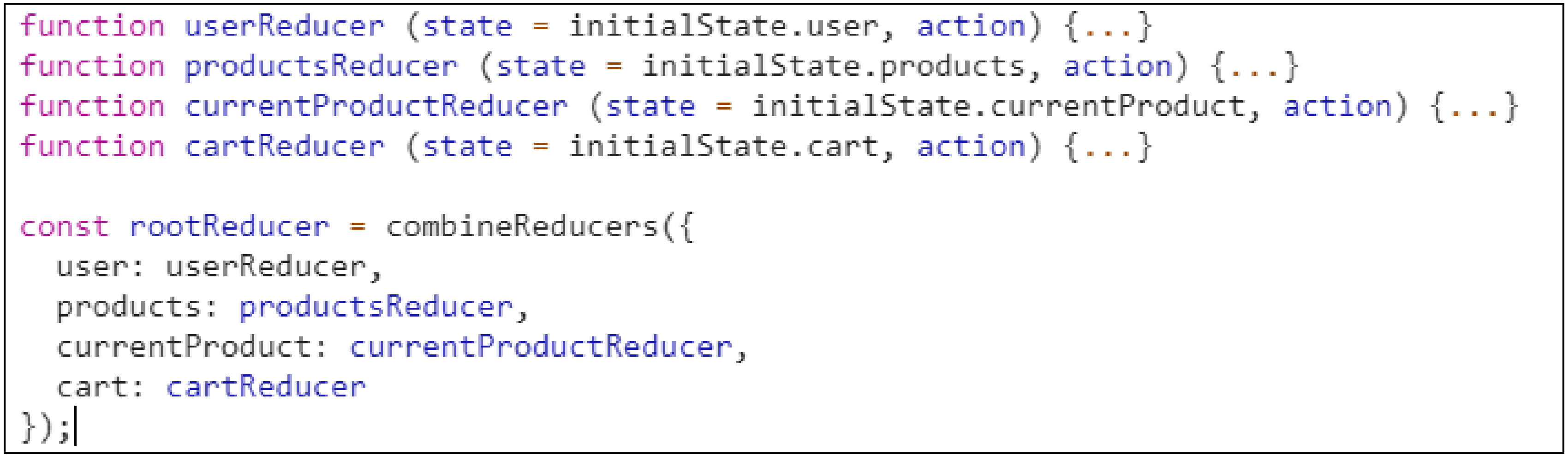


```
const store = redux.createStore(reducer, [initialState], [enhancer])
```

- Methods of the store:
  - **store.getState():** Returns the actual state of the store
  - **store.dispatch(action):** Emits an action, this is the only way to modify the state
  - **store.subscribe(listener):** Allows the subscription to the changes that happen. The listener is called each time that an action is emitted and a piece of the state may be changed

API (II)

- **combineReducers()**
  - In Redux we only have a store to handle the global state. It is a good practice to have several reducers (one for each part/piece of the state) and with this function we can combine them in a single reducer that will be passed as param to the function createStore
- **Why should we have several reducers?** Because this way we can divide our problems in several parts and it is simpler to modularize the app



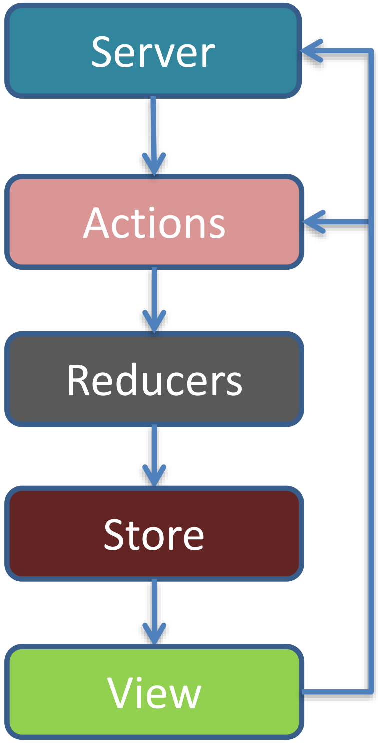
```
function userReducer (state = initialState.user, action) {...}
function productsReducer (state = initialState.products, action) {...}
function currentProductReducer (state = initialState.currentProduct, action) {...}
function cartReducer (state = initialState.cart, action) {...}

const rootReducer = combineReducers({
  user: userReducer,
  products: productsReducer,
  currentProduct: currentProductReducer,
  cart: cartReducer
});
```

API (III)

- There are more methods that we won´t be using in this course:
  - applyMiddleware()
  - bindActionCreators()
  - compose()
- More info: http://redux.js.org/

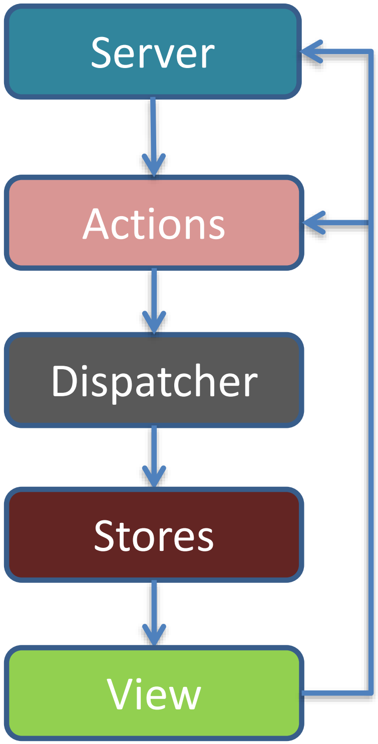
Unidirectional data flow

- 1 someone calls store.dispatch(action)
  - From any part of the app, a component, an Ajax call, a timeout, …
- 2 the Redux store calls the reducer and passes the actual state and the action
  - The reducer can be only one or the result of combineReducers() call
- 3 the reducers are executed and update the state (each one with the piece of the state that corresponds) and generate a new state
- 4 the store in Redux saves the new state and notifies each listener that was registered with store.subscribe(listener)
  - The listeners will be able to call store.getState() to obtain the new state

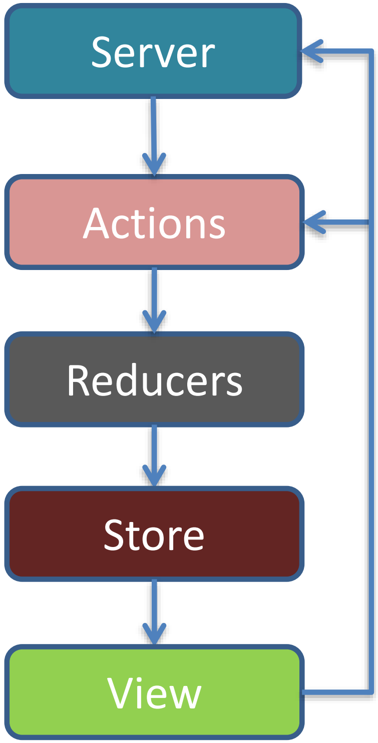


- More info: https://redux.js.org/basics/data-flow

Flux vs. Redux



Flux



Redux

Redux Lifecycle and API

Enrique Barra

React Redux

React and Redux

Enrique Barra

¿How is it integrated with React?

- We need to transfer from Redux to React:
  - The **state**
  - The **actions** that modify the state
- We want also to maintain the immutability of the state and the functions that modify it
  - What is the most similar thing in React => the **props**

Solution: We create a component <ReduxProvider/> that passes the state as props to the root component of the app

We will use a library called **react-redux** (installed via npm)

React state vs. Redux state

- **We can have the whole state in Redux, but we can also leave part of the state in the React components**
- **React state:** state of a component. Each component has access to its state and can pass it to its children as props. Usually contains information only relevant to the view
  - For example if the component is active or has a dropdown opened. We don´t want this state to be persistent, it won´t be saved to the database and when I open my app again I don´t need it, I have a default state for it
- **Redux state:** state of the app that all components can have access. Usually contains information that we want to save between sessions
  - For example each part of the state that we want to be persistent, such as data, etc. We need it when we open the app again

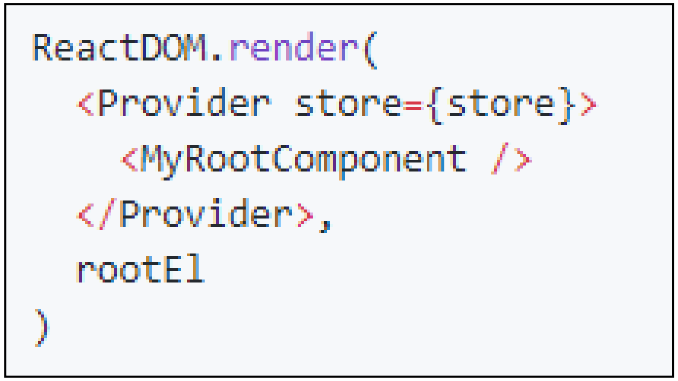
React-redux

- https://github.com/reactjs/react-redux
- Redux is framework agnostic (can be used with Angular, Ember, …) the join with React is provided as part of a library
- This library has an API that provides two things
  - A **provider** component
  - A **connect** method
- API: https://github.com/reactjs/react-redux/blob/master/docs/api.md

React-redux – Provider component

- **<Provider store>**
- React component that will include our app (our root component) and will pass it the state as props

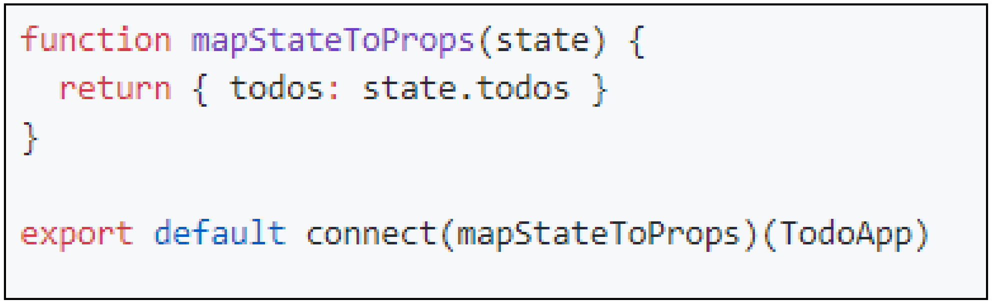
- Provider Props
  - Store (our app store)
  - Children (our app)



```
ReactDOM.render(
  <Provider store={store}>
    <MyRootComponent />
  </Provider>,
  rootEl
)
```

React-redux –connect method

- **connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])**
  - Info at: https://github.com/reactjs/react-redux/blob/master/docs/api.md#connectmapstatetoprops-mapdispatchtoprops-mergeprops-options
- Connect a React component to a Store
- Do not modify the component that is passed but returns a new one to be used (so we will export it instead of our root component)
- Params (all optional):
  - **mapStateToProps:** receives the state and returns the props object that will be passed to the component
  - If we specify a component, it will be subscribed to the store. So each time that the state is modified, mapStateToProps will be called and will receive new props



```
function mapStateToProps(state) {
  return { todos: state.todos }
}

export default connect(mapStateToProps)(TodoApp)
```

React-redux – using it

- We will create a new component called ReduxProvider, it is an intermediate component between index.js and App.jsx
- It renders a React-redux **Provider** with its store
- It will be the rendered component in index.js

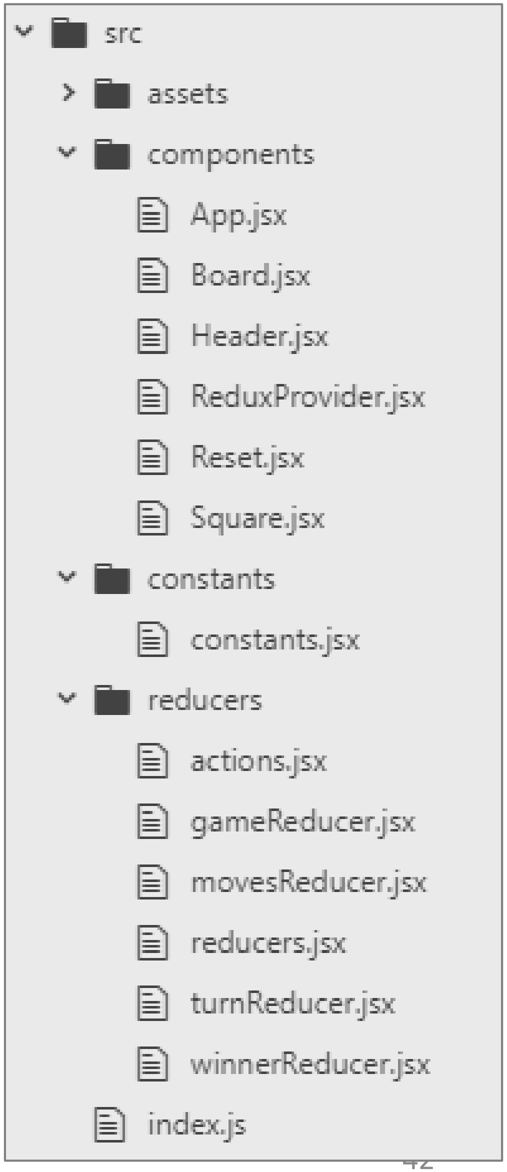
- In App.jsx we will use React-redux **connect** method to connect App with the store using a method mapstatetoprops

Adding Redux to an existing React app

1. Download the dependencies

```
npm install --save-dev react-redux redux
```

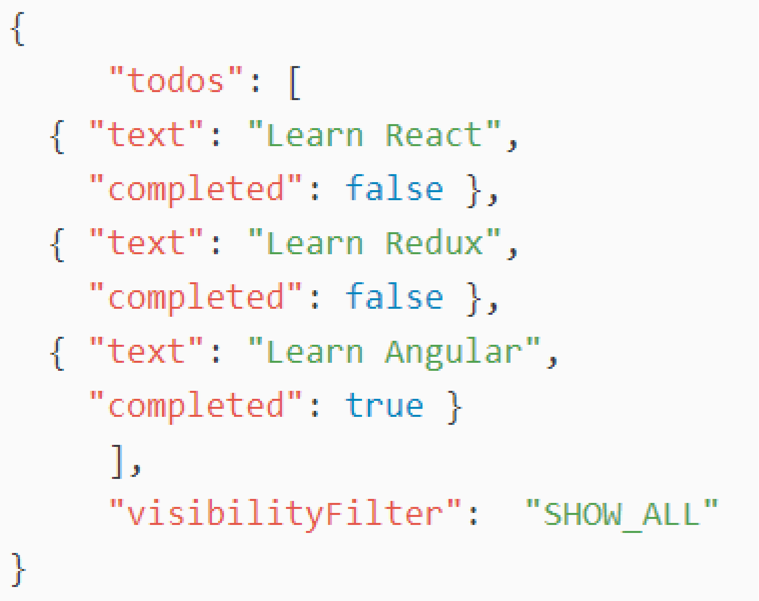
2. Define your app state and put it in *app/constants/constants.jsx*
3. Create the file *app/reducers/actions.jsx*
4. Create the file *app/reducers/reducers.jsx*
5. Create component *app/components/ReduxProvider.jsx*
6. Modify *app/index.js*
7. Modify *app/components/App.jsx*



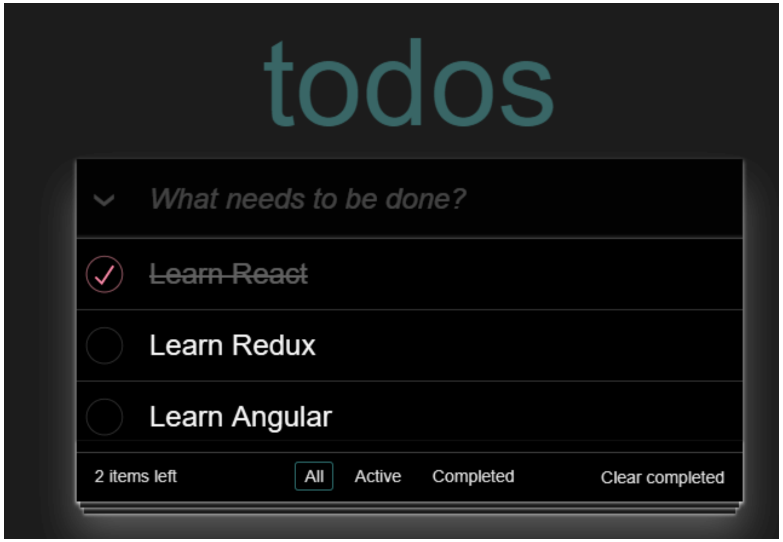
```
src
  assets
  components
    App.jsx
    Board.jsx
    Header.jsx
    ReduxProvider.jsx
    Reset.jsx
    Square.jsx
  constants
    constants.jsx
  reducers
    actions.jsx
    gameReducer.jsx
    movesReducer.jsx
    reducers.jsx
    turnReducer.jsx
    winnerReducer.jsx
  index.js
```

State (immutable)

- The state of the app is a JS object (immutable, not directly modified)
- Example: TO-DO app



```
{
     "todos": [
  { "text": "Learn React",
    "completed": false },
  { "text": "Learn Redux",
    "completed": false },
  { "text": "Learn Angular",
    "completed": true }
     ],
     "visibilityFilter":  "SHOW_ALL"
}
```

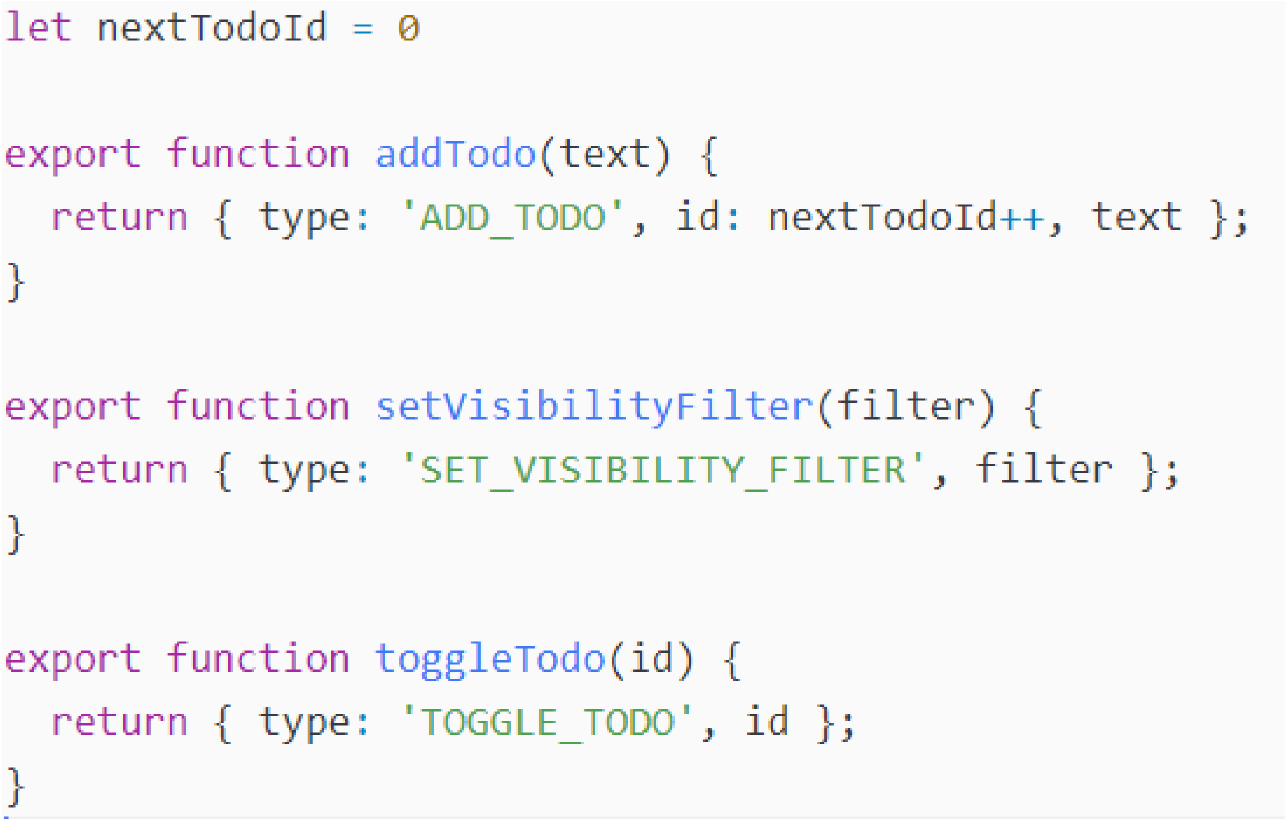


~~Learn React~~

Learn Redux

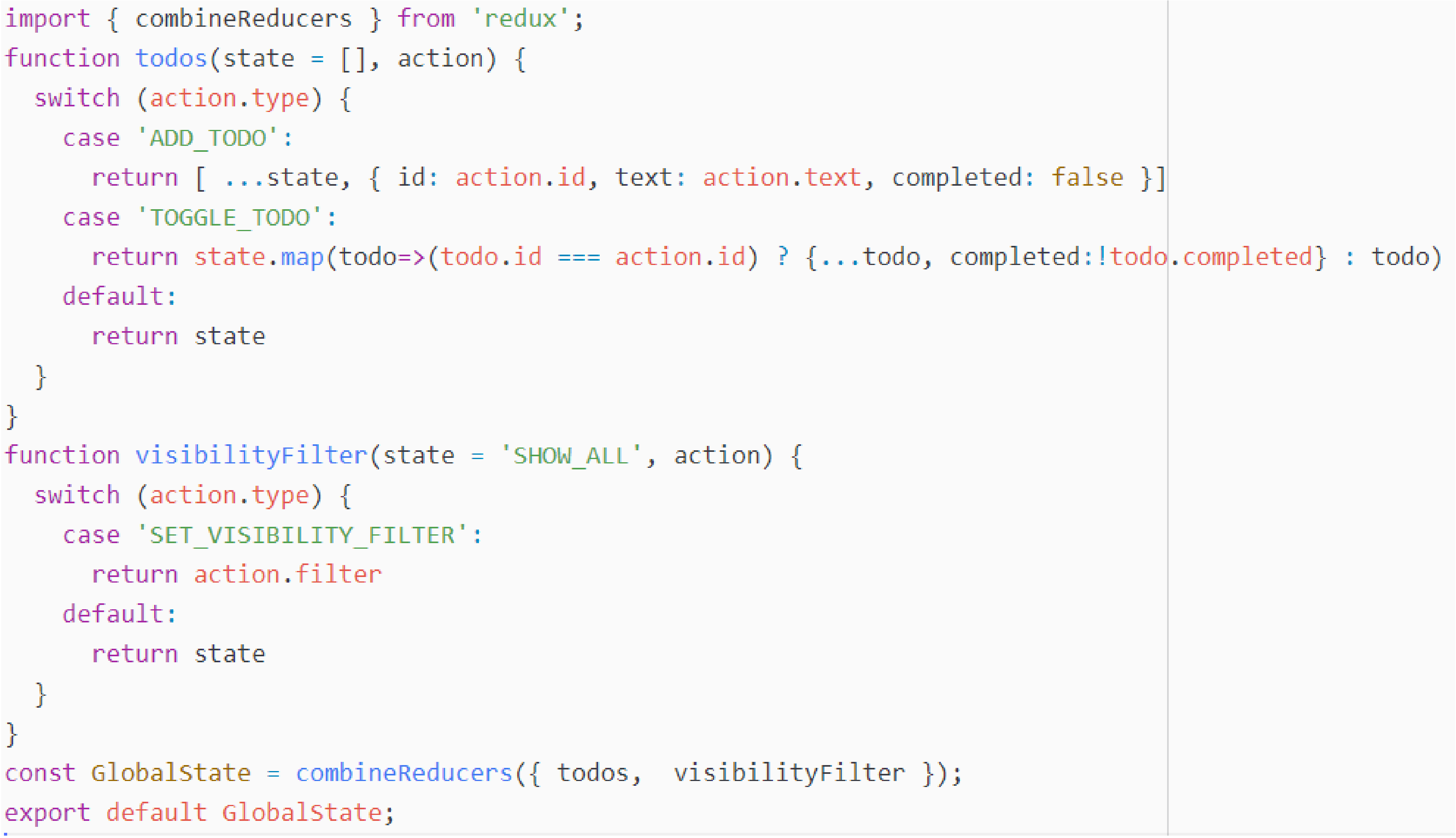
Learn Angular

Step 2: *actions.jsx*



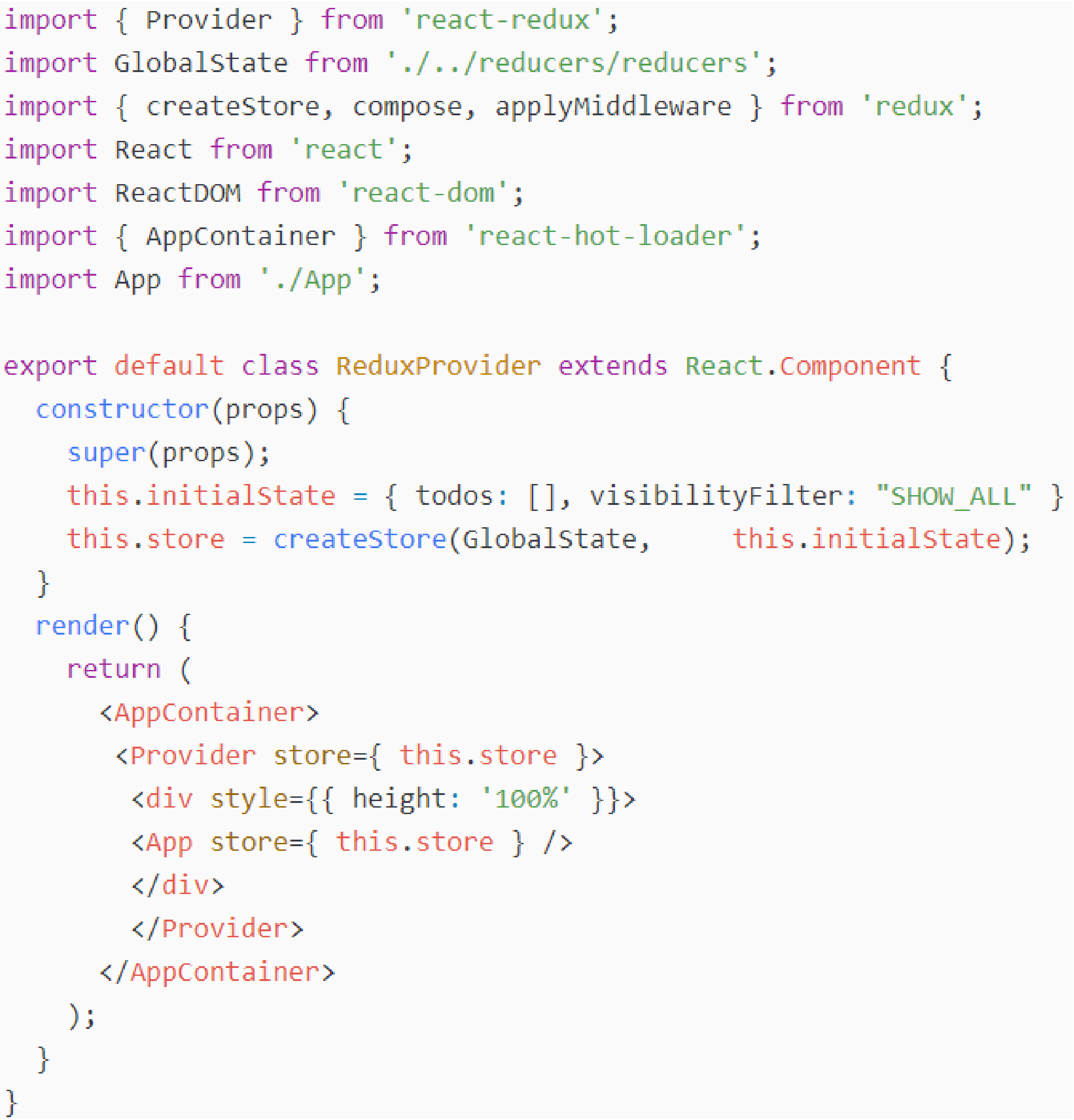
```
let nextTodoId = 0

export function addTodo(text) {
  return { type: 'ADD_TODO', id: nextTodoId++, text };
}

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

export function toggleTodo(id) {
  return { type: 'TOGGLE_TODO', id };
}
```

Step 3: *reducers.jsx*



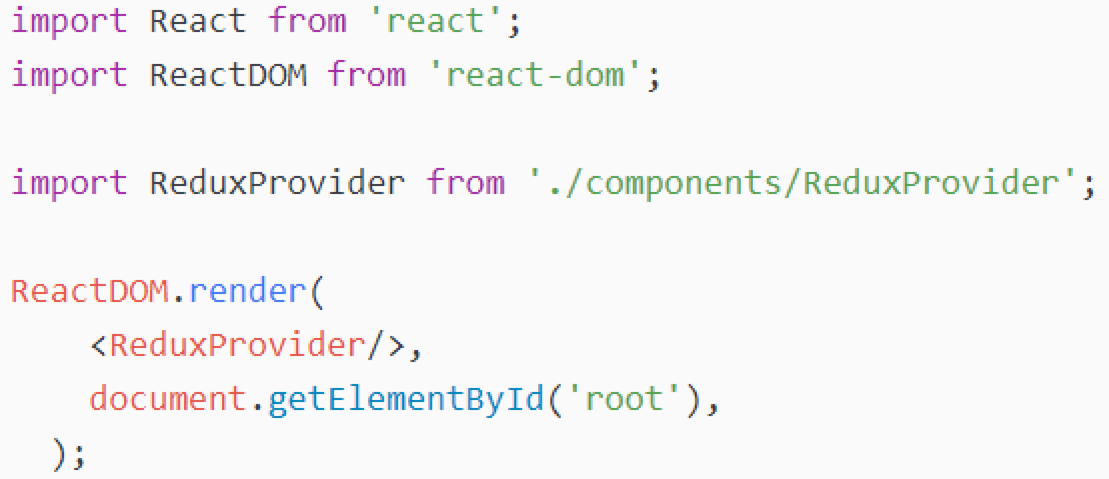
```
import { combineReducers } from 'redux';
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [ ...state, { id: action.id, text: action.text, completed: false }]
    case 'TOGGLE_TODO':
      return state.map(todo=>(todo.id === action.id) ? {...todo, completed:!todo.completed} : todo)
    default:
      return state
  }
}
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}
const GlobalState = combineReducers({ todos,  visibilityFilter });
export default GlobalState;
```

Step 4: *ReduxProvider.jsx*



```
import { Provider } from 'react-redux';
import GlobalState from './../reducers/reducers';
import { createStore, compose, applyMiddleware } from 'redux';
import React from 'react';
import ReactDOM from 'react-dom';
import { AppContainer } from 'react-hot-loader';
import App from './App';

export default class ReduxProvider extends React.Component {
  constructor(props) {
    super(props);
    this.initialState = { todos: [], visibilityFilter: "SHOW_ALL" }
    this.store = createStore(GlobalState,     this.initialState);
  }
  render() {
    return (
      <AppContainer>
       <Provider store={ this.store }>
        <div style={{ height: '100%' }}>
        <App store={ this.store } />
        </div>
        </Provider>
      </AppContainer>
    );
  }
}
```

Step 5: *index.js*



```
import React from 'react';
import ReactDOM from 'react-dom';

import ReduxProvider from './components/ReduxProvider';

ReactDOM.render(
    <ReduxProvider/>,
    document.getElementById('root'),
  );
```

Step 6: *App.jsx*

Add dependencies:

```
import { connect } from 'react-redux';
import { addTodo, setVisibilityFilter, toggleTodo } from './../reducers/actions';
```

Change the component declaration:

~~export default~~ `class App extends React.Component {`

Connect the <App/> component with Redux

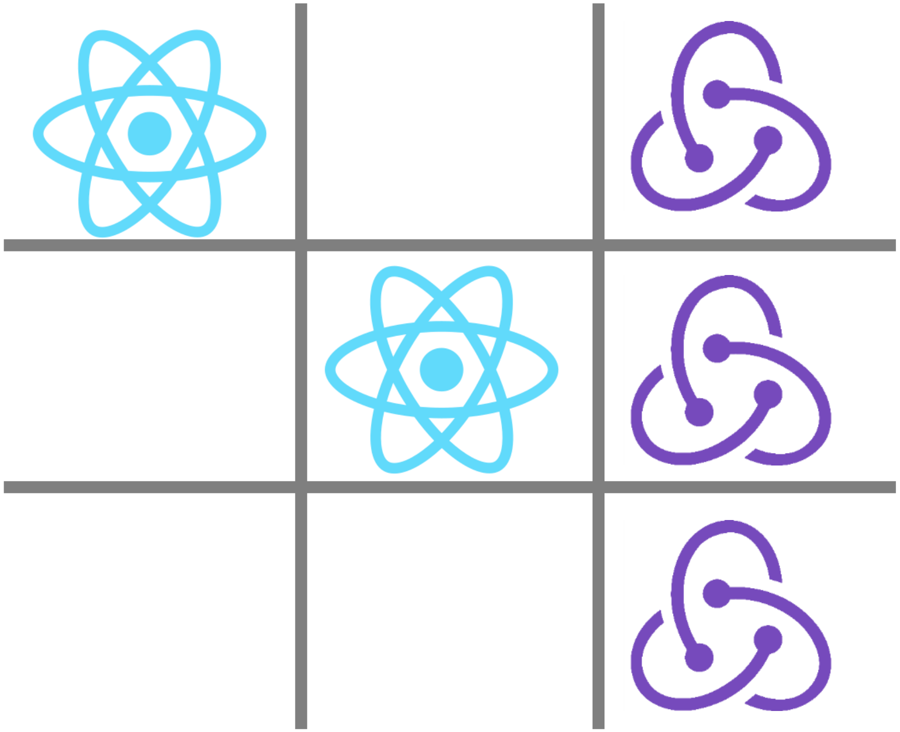
```
function mapStateToProps(state) {
    return {
        todos: state.todos,
        visibilityFilter: state.visibilityFilter
    };
}
export default connect(mapStateToProps)(App);
```

Now we hace access to `this.props.todos` and `this.props.visibilityFilter` in <App/>

To call an action: `this.props.dispatch(setVisibilityFilter(‘SHOW_ACTIVE’))`

React and Redux

Enrique Barra



Adding Redux to the Tic Tac Toe

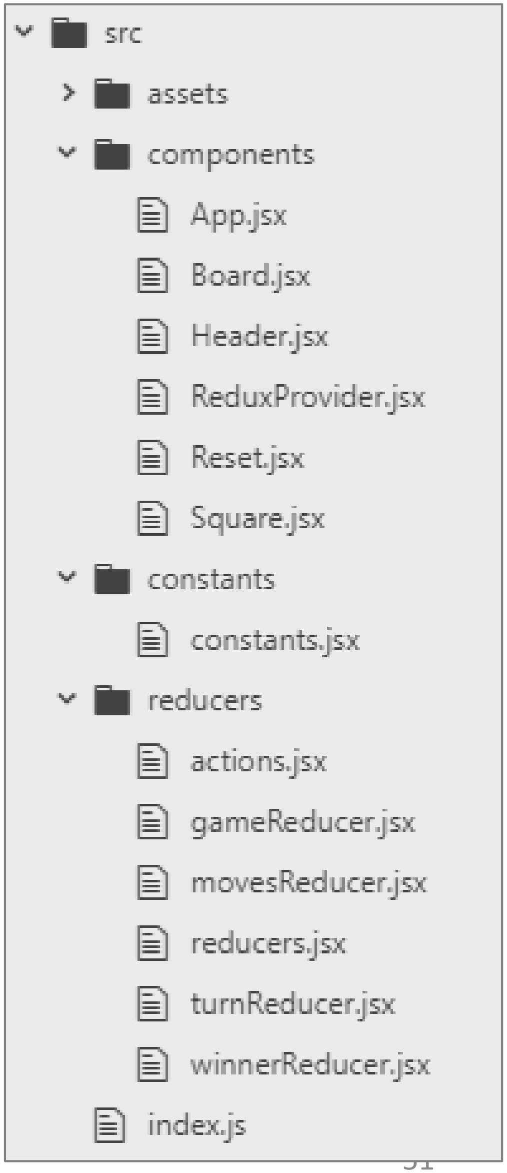
Enrique Barra

Adding Redux to an existing React app

1. Download the dependencies

```
npm install --save-dev react-redux redux
```

2. Define your app state and put it in *app/constants/constants.jsx*
3. Create the file *app/reducers/actions.jsx*
4. Create the file *app/reducers/reducers.jsx*
5. Create the rest of reducers *app/reducers/…jsx*
6. Create component *app/components/ReduxProvider.jsx*
7. Modify *app/index.js*
8. Modify *app/components/App.jsx*



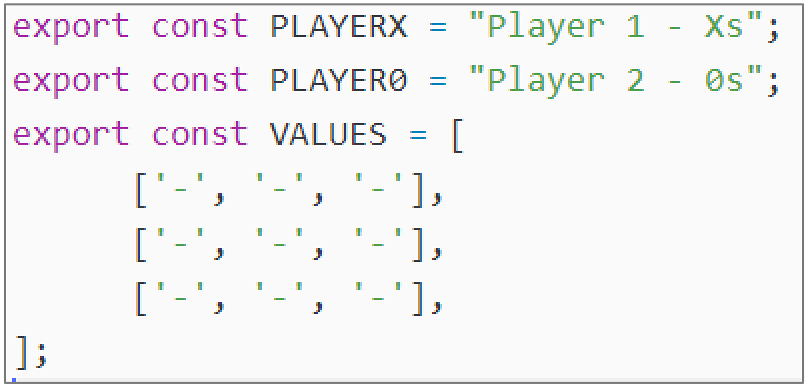
```
src
  assets
  components
    App.jsx
    Board.jsx
    Header.jsx
    ReduxProvider.jsx
    Reset.jsx
    Square.jsx
  constants
    constants.jsx
  reducers
    actions.jsx
    gameReducer.jsx
    movesReducer.jsx
    reducers.jsx
    turnReducer.jsx
    winnerReducer.jsx
  index.js
```

2. Define your app state

app/constants/constants.jsx

- In small apps it is usually the same as the state in App.jsx
  - Board: Array
  - Turn: String

- Define initial state (we will put those values in app/constants/constants.jsx)
  - Empty board
  - Turn of playerx

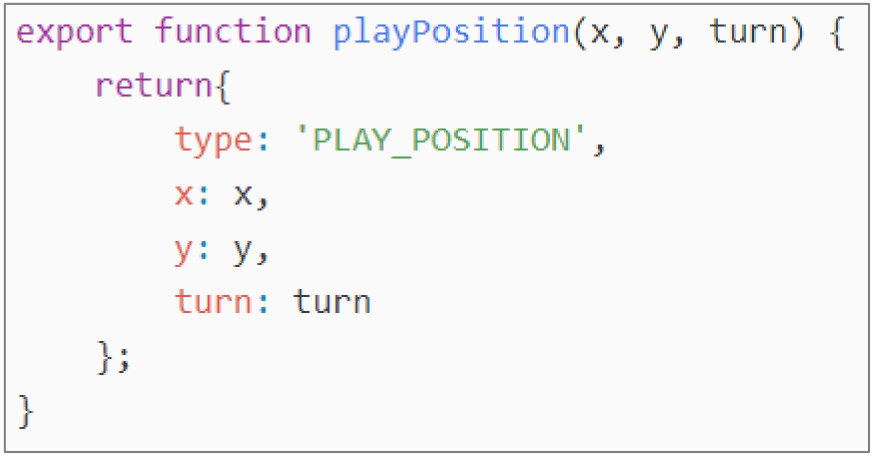


```
export const PLAYERX = "Player 1 - Xs";
export const PLAYER0 = "Player 2 - 0s";
export const VALUES = [
      ['-', '-', '-'],
      ['-', '-', '-'],
      ['-', '-', '-'],
];
```

3. Create the file

app/reducers/actions.jsx

- First action **PLAY_POSITION** each time a player plays a square
- What info does it need?
  - The player that did it
  - The square that was clicked

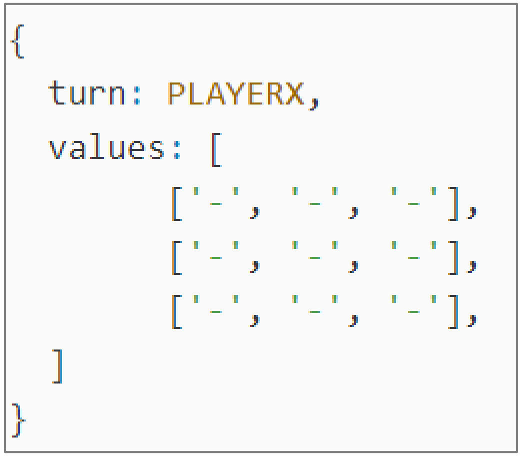


```
export function playPosition(x, y, turn) {
    return{
        type: 'PLAY_POSITION',
        x: x,
        y: y,
        turn: turn
    };
}
```

4. Create the file

app/reducers/reducers.jsx

- The state has two parts (turn and values)
- We will use a reducer for each part
- So inside reducers folder we create:
  - turnReducer.jsx (to manage turn)
  - gameReducer.jsx (to manage values)

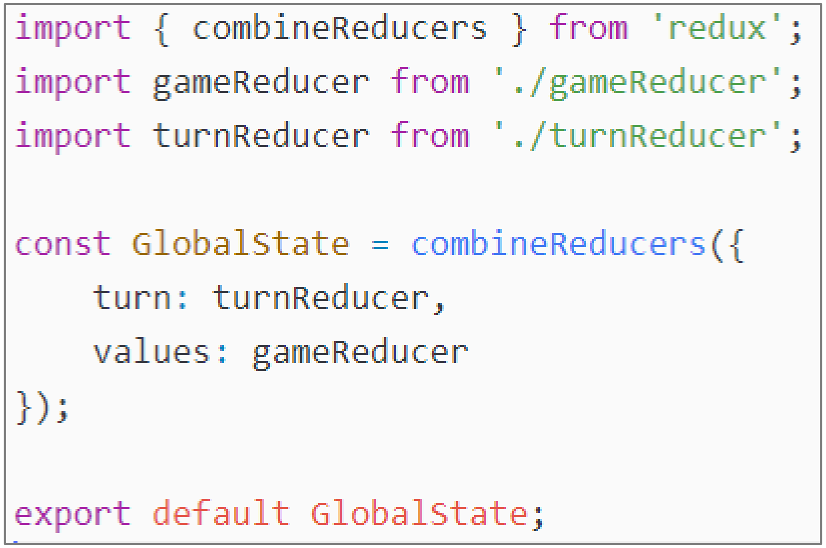


```
{
  turn: PLAYERX,
  values: [
         ['-', '-', '-'],
         ['-', '-', '-'],
         ['-', '-', '-'],
  ]
}
```

- In reducers.jsx we will unify/combine all parts to create the global state

4. Create the file

app/reducers/reducers.jsx



```
import { combineReducers } from 'redux';
import gameReducer from './gameReducer';
import turnReducer from './turnReducer';

const GlobalState = combineReducers({
    turn: turnReducer,
    values: gameReducer
});

export default GlobalState;
```

Single Reducer structure

State param is the piece of the state that corresponds to this reducer, not the full or global state

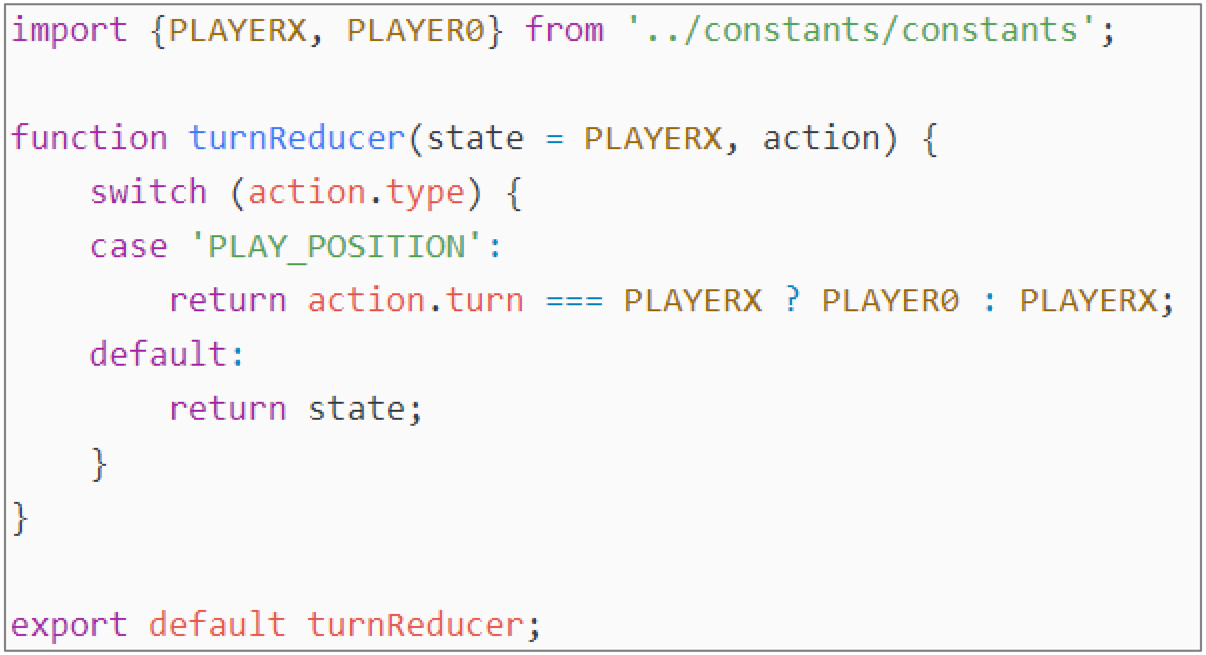
In the **action param** comes all the needed information to modify the state, specially the type of action that it is

```
// import {..} …….

function myReducer(state = “DEFAULT_STATE”, action) {
  switch (action.type) {
    case 'ACTION_NAME':
      let newState = Object.assign([], state);
      // … Modify newState
      return newState; // Return the modified state
    default:
      return state;
  }
}
export default myReducer;
```

5. Create the rest of reducers

app/reducers/turnReducers.jsx

We have to change the turn to the other player (the one that has not moved now)

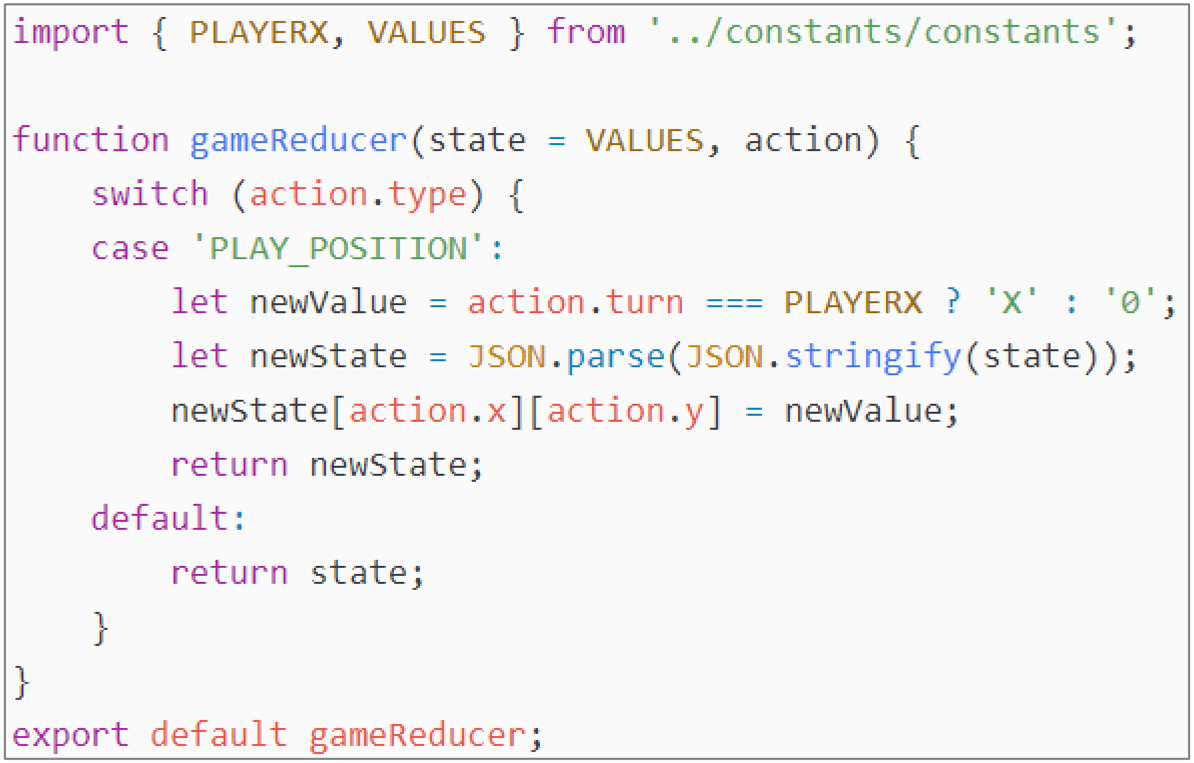


```
import {PLAYERX, PLAYER0} from '../constants/constants';

function turnReducer(state = PLAYERX, action) {
    switch (action.type) {
    case 'PLAY_POSITION':
        return action.turn === PLAYERX ? PLAYER0 : PLAYERX;
    default:
        return state;
    }
}

export default turnReducer;
```

5. Create the rest of reducers

app/reducers/gameReducers.jsx

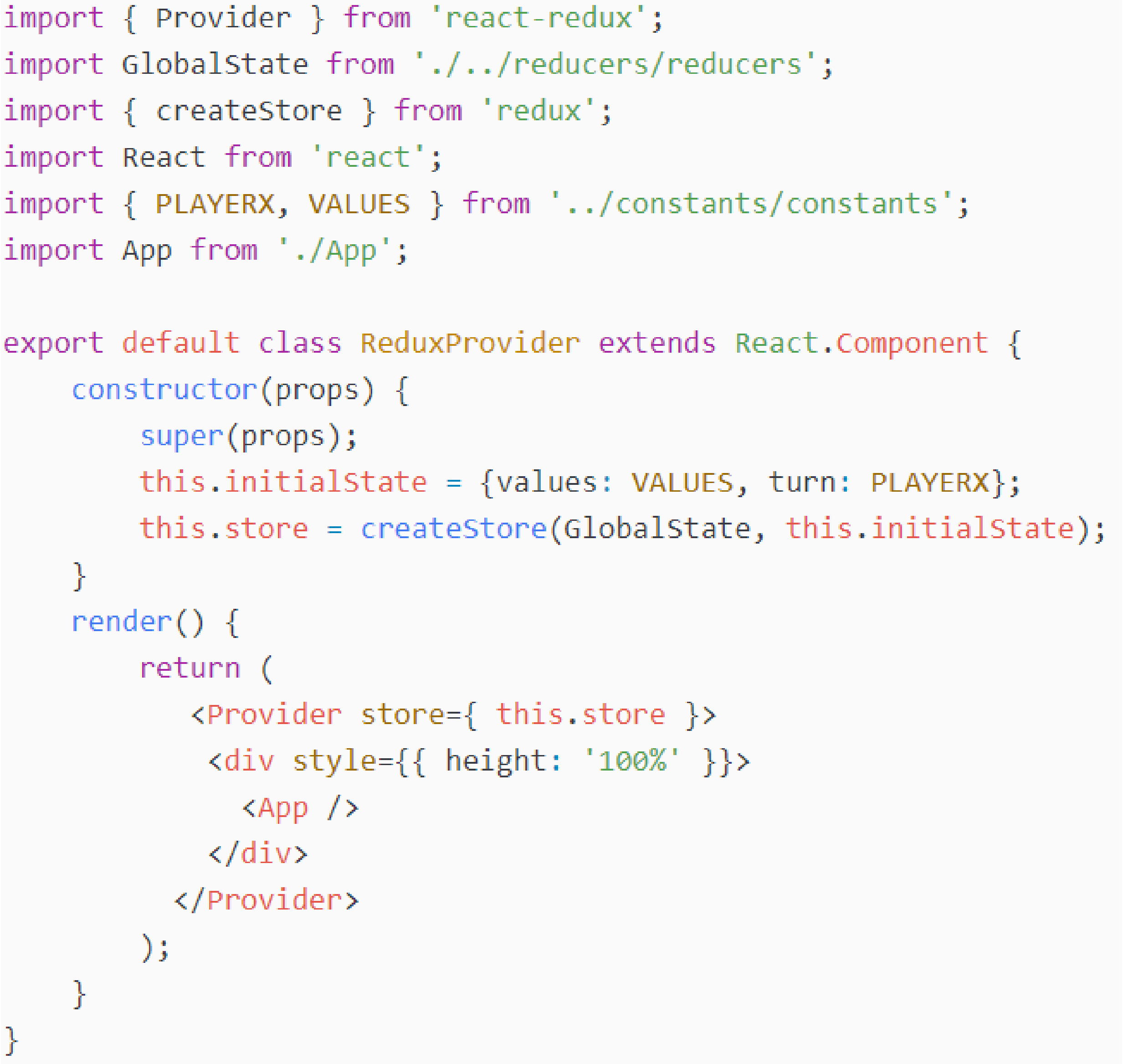
We have to fill in the square with an ‘X’ or a ‘0’



```
import { PLAYERX, VALUES } from '../constants/constants';

function gameReducer(state = VALUES, action) {
    switch (action.type) {
    case 'PLAY_POSITION':
        let newValue = action.turn === PLAYERX ? 'X' : '0';
        let newState = JSON.parse(JSON.stringify(state));
        newState[action.x][action.y] = newValue;
        return newState;
    default:
        return state;
    }
}
export default gameReducer;
```

6. Create component

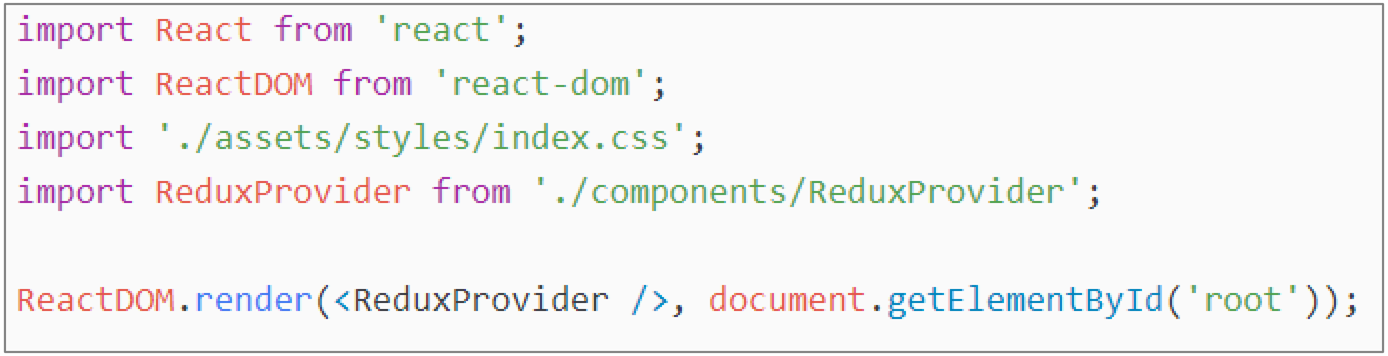
app/components/ReduxProvider.jsx



```
import { Provider } from 'react-redux';
import GlobalState from './../reducers/reducers';
import { createStore } from 'redux';
import React from 'react';
import { PLAYERX, VALUES } from '../constants/constants';
import App from './App';

export default class ReduxProvider extends React.Component {
    constructor(props) {
        super(props);
        this.initialState = {values: VALUES, turn: PLAYERX};
        this.store = createStore(GlobalState, this.initialState);
    }
    render() {
        return (
           <Provider store={ this.store }>
            <div style={{ height: '100%' }}>
              <App />
            </div>
          </Provider>
        );
    }
}
```

7. Modify

app/index.js



```
import React from 'react';
import ReactDOM from 'react-dom';
import './assets/styles/index.css';
import ReduxProvider from './components/ReduxProvider';

ReactDOM.render(<ReduxProvider />, document.getElementById('root'));
```

8. Modify

app/components/App.jsx

Add dependencies

```
import { connect } from 'react-redux';
import {playPosition} from './../reducers/actions';
```

Change the component declaration

```
~~export default~~ class App extends React.Component {
```

Connect the props of the <App/> component with the Redux state

```
function mapStateToProps(state) {
  return {
        values: state.values,
        turn: state.turn
    };
}
export default connect(mapStateToProps)(App);
```

Now we will have access to `this.props.values` and `this.props.turn` in <App/>
Example of call to an action: `this.props.dispatch(playPosition(0,0,PLAYER_X))`

8. Modify

app/components/App.jsx

Delete the initial state from the constructor (now we get it from Redux)

```
class App extends React.Component {
  constructor(props) {
    super(props);
    ~~//this.state = {..}~~
    this.appClick = this.appClick.bind(this);
}
```

Change the render method to use props instead of state

```
 render() {
    let text = "Turn of " + ~~this.state.turn~~ this.props.turn;
    return (
      <div>
        <Header text={text} />
        <Board values={~~this.state.values~~ this.props.values} appClick={this.appClick}/>
       </div>
    );
  }
```

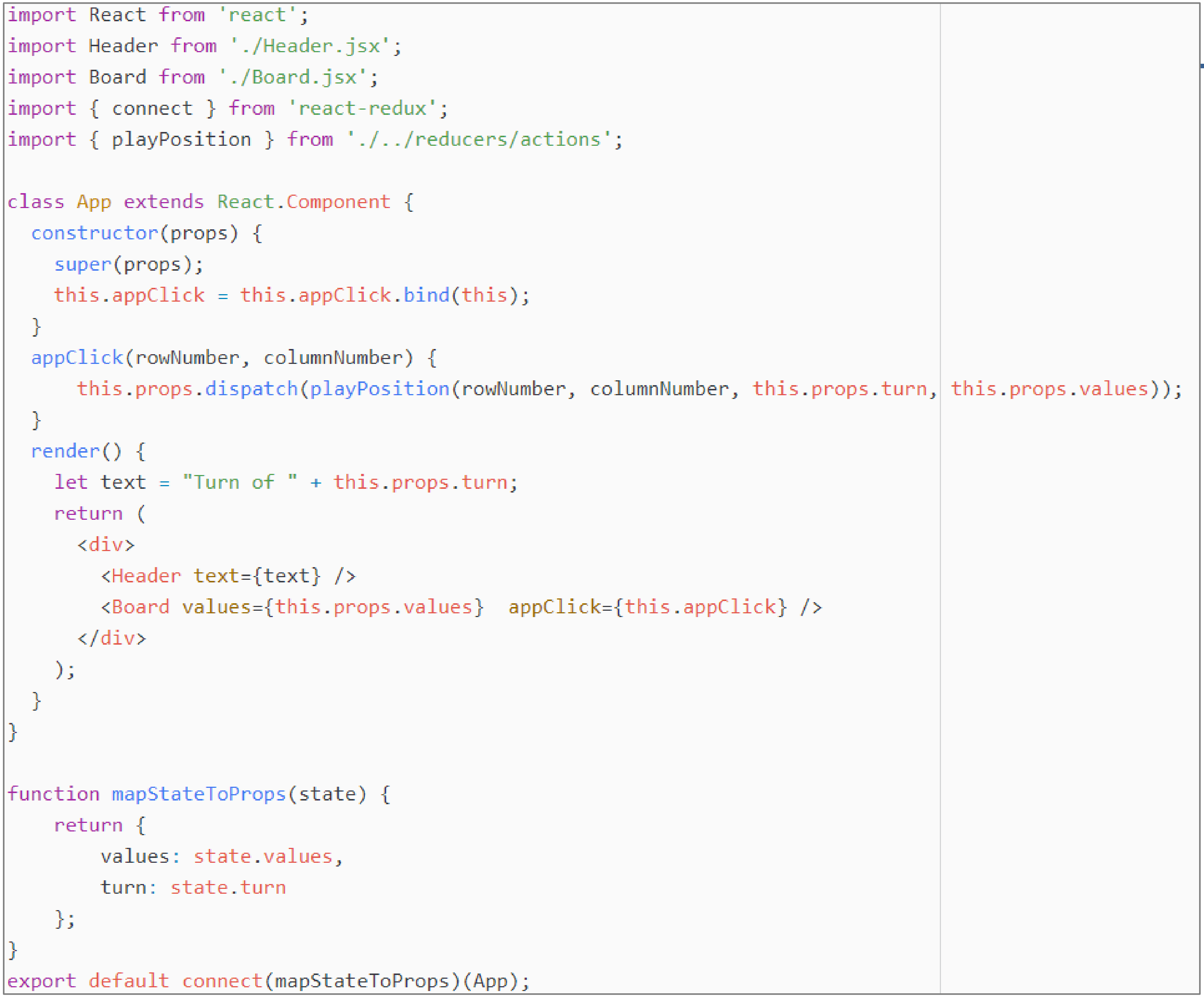
8. Modify

app/components/App.jsx

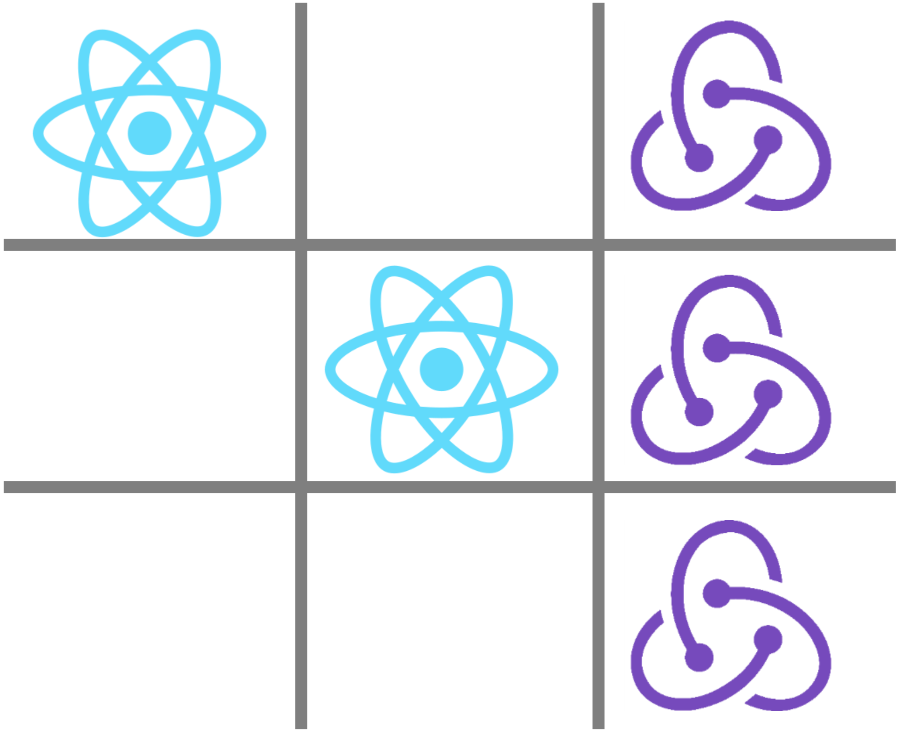
Modify the method *appClick()*, now all the logic goes to the reducers

```
 appClick(rowNumber, columnNumber) {
      ~~let valuesCopy = JSON.parse(JSON.stringify(this.state.values));~~
     ~~let newMovement = this.state.turn === PLAYERX ? 'X' : '0';~~
     ~~valuesCopy[rowNumber][columnNumber] = newMovement;~~
     ~~this.setState({~~
          ~~turn: this.state.turn === PLAYERX ? PLAYER0 : PLAYERX,~~
          ~~values: valuesCopy,~~
          ~~moves: this.state.moves +1 });~~
     this.props.dispatch(playPosition(rowNumber, columnNumber, this.props.turn));
}
```

8. Modify - *app/components/App.jsx*

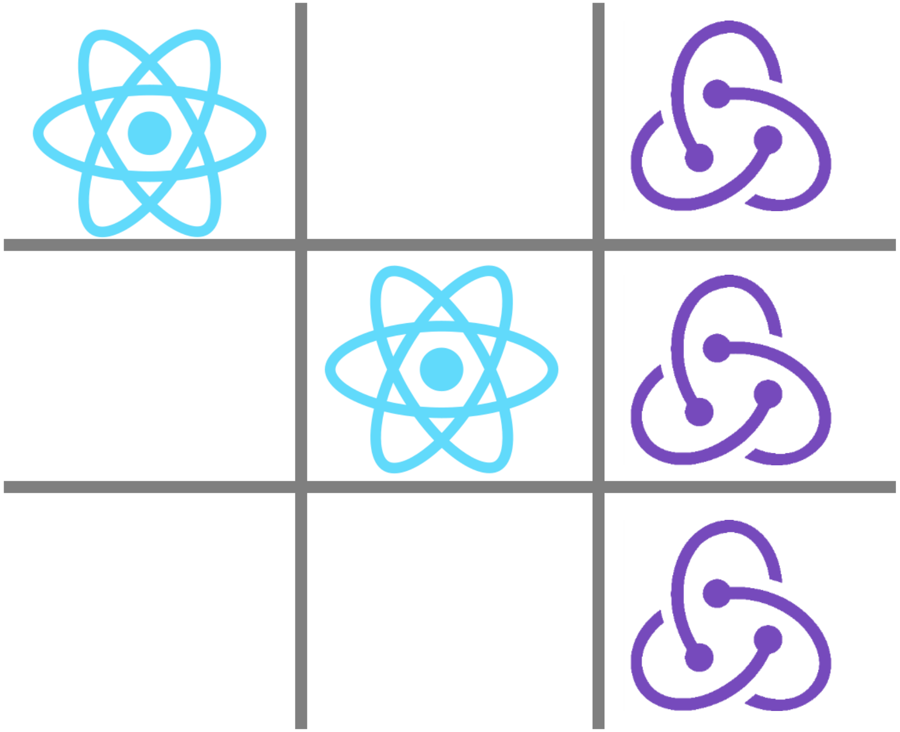


```
import React from 'react';
import Header from './Header.jsx';
import Board from './Board.jsx';
import { connect } from 'react-redux';
import { playPosition } from './../reducers/actions';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.appClick = this.appClick.bind(this);
  }
  appClick(rowNumber, columnNumber) {
      this.props.dispatch(playPosition(rowNumber, columnNumber, this.props.turn, this.props.values));
  }
  render() {
    let text = "Turn of " + this.props.turn;
    return (
      <div>
        <Header text={text} />
        <Board values={this.props.values}  appClick={this.appClick} />
      </div>
    );
  }
}

function mapStateToProps(state) {
    return {
        values: state.values,
        turn: state.turn
    };
}
export default connect(mapStateToProps)(App);
```



Adding Redux to the Tic Tac Toe

Enrique Barra



Adding Reset action to the Tic Tac Toe

Enrique Barra

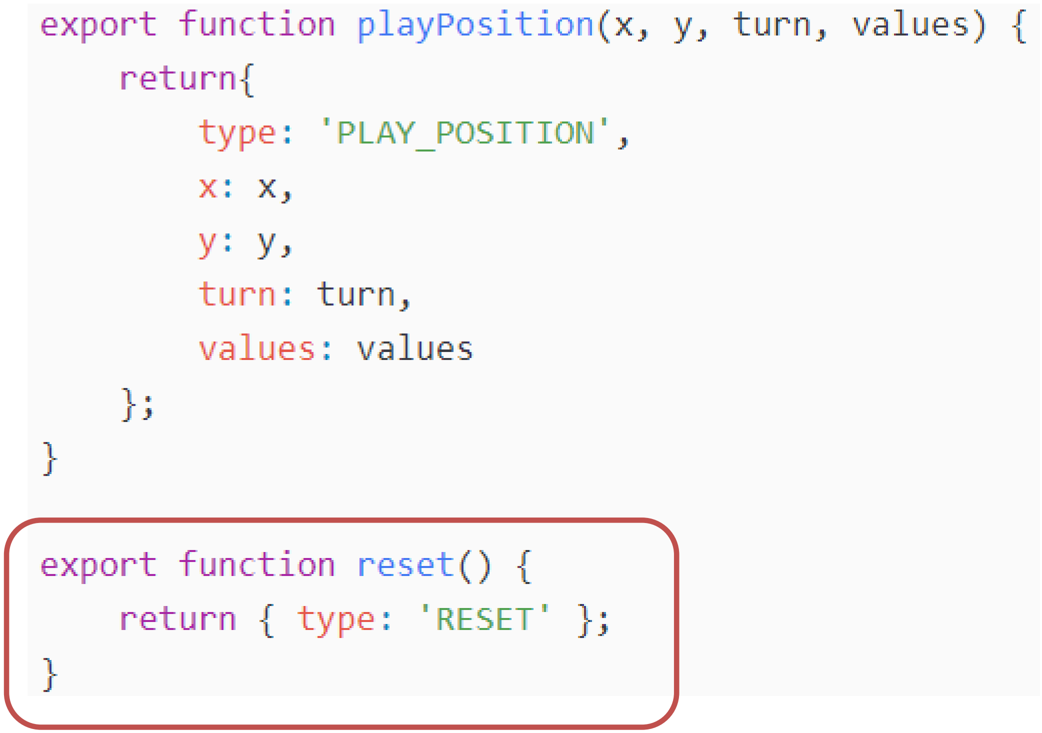
9. Extra - New reset action

- We want the reset button to restart the game
- We define a new action and modify the reducers so they consider the **RESET** case and not only **PLAY_POSITION**
- It won´t need attributes

9. Extra - New reset action

app/reducers/actions.jsx

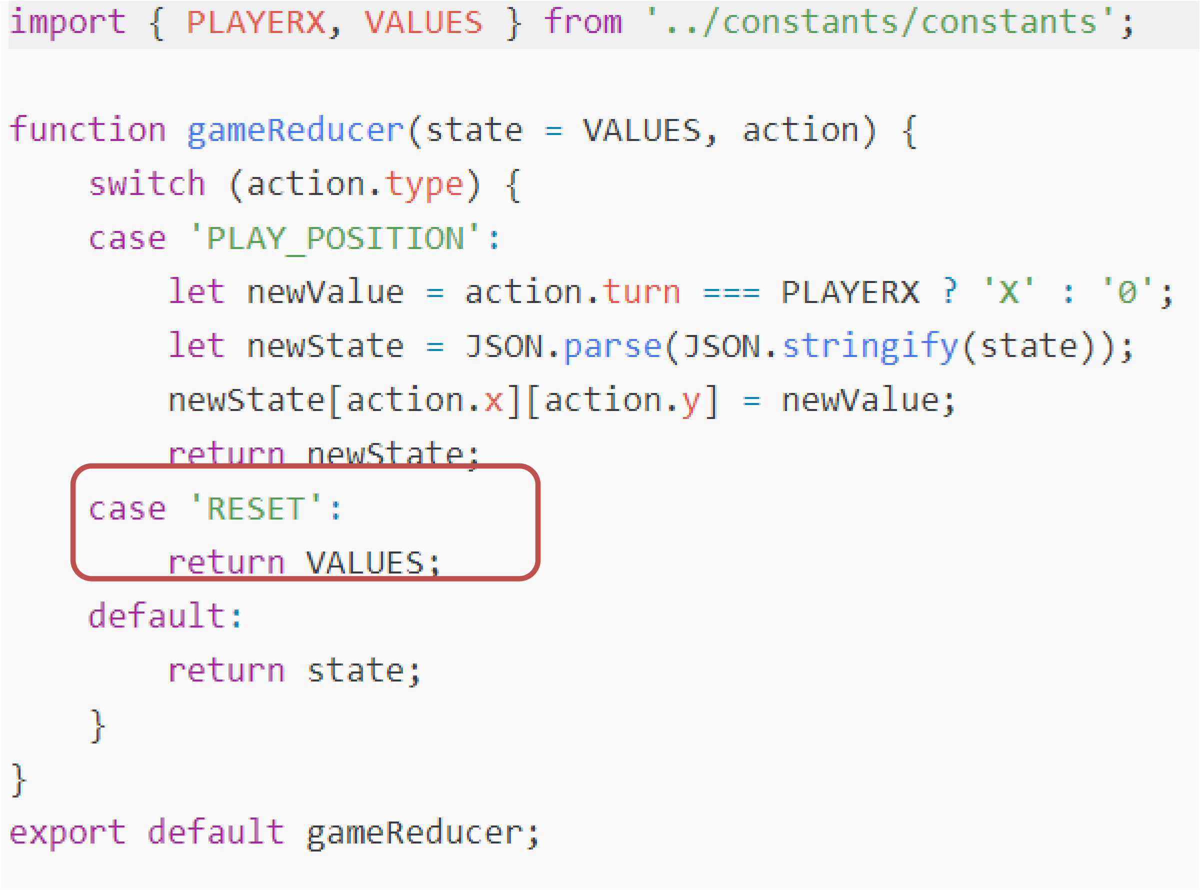
Create new action called “reset”



```
export function playPosition(x, y, turn, values) {
    return{
        type: 'PLAY_POSITION',
        x: x,
        y: y,
        turn: turn,
        values: values
    };
}

export function reset() {
    return { type: 'RESET' };
}
```

9. Extra - New reset action
app/reducers/gameReducer.jsx

We have to empty the board

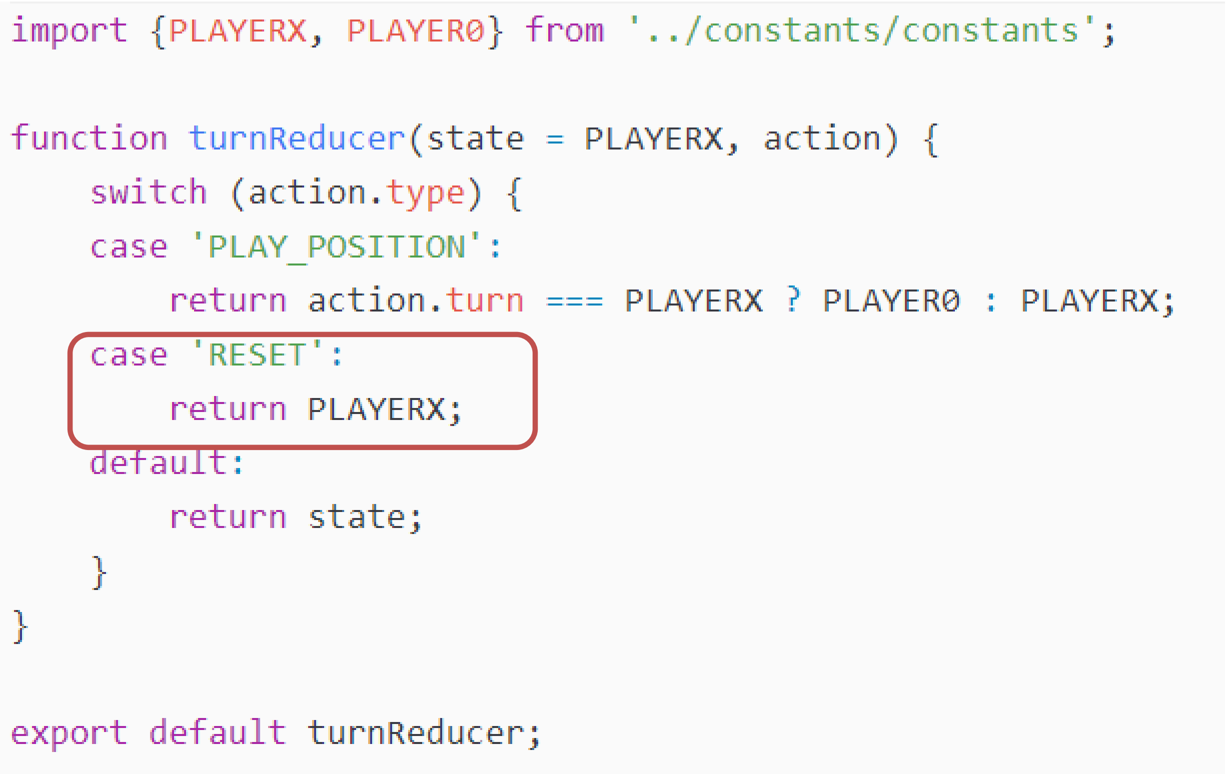


```
import { PLAYERX, VALUES } from '../constants/constants';

function gameReducer(state = VALUES, action) {
    switch (action.type) {
    case 'PLAY_POSITION':
        let newValue = action.turn === PLAYERX ? 'X' : '0';
        let newState = JSON.parse(JSON.stringify(state));
        newState[action.x][action.y] = newValue;
        return newState;
    case 'RESET':
        return VALUES;
    default:
        return state;
    }
}
export default gameReducer;
```

9. Extra - New reset action

app/reducers/turnReducer.jsx

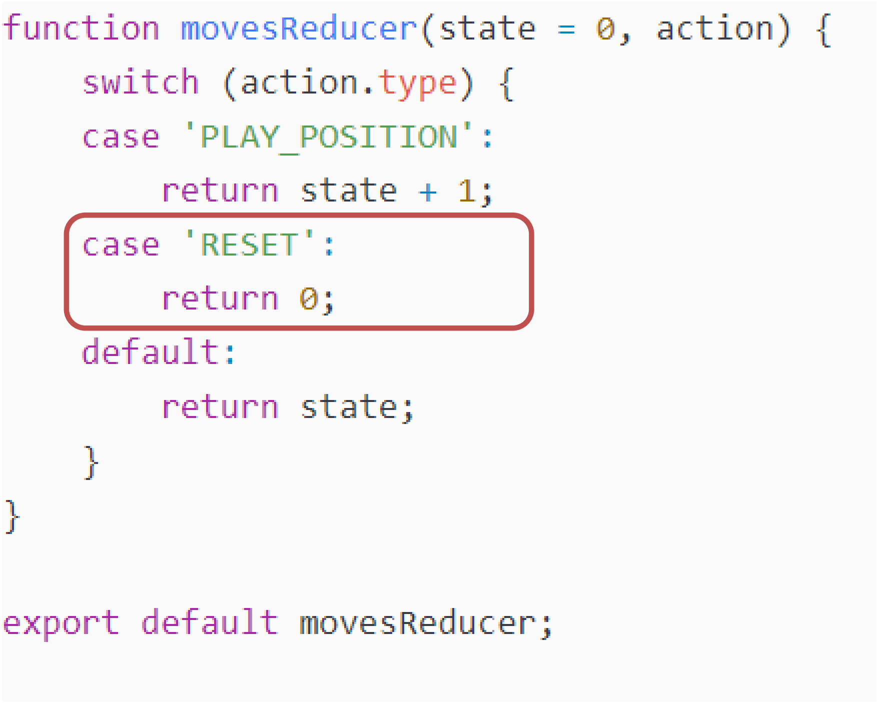
We have to set the turn to the initial player



```
import {PLAYERX, PLAYER0} from '../constants/constants';

function turnReducer(state = PLAYERX, action) {
    switch (action.type) {
    case 'PLAY_POSITION':
        return action.turn === PLAYERX ? PLAYER0 : PLAYERX;
    case 'RESET':
        return PLAYERX;
    default:
        return state;
    }
}

export default turnReducer;
```

9. Extra - New reset action
app/reducers/movesReducer.jsx

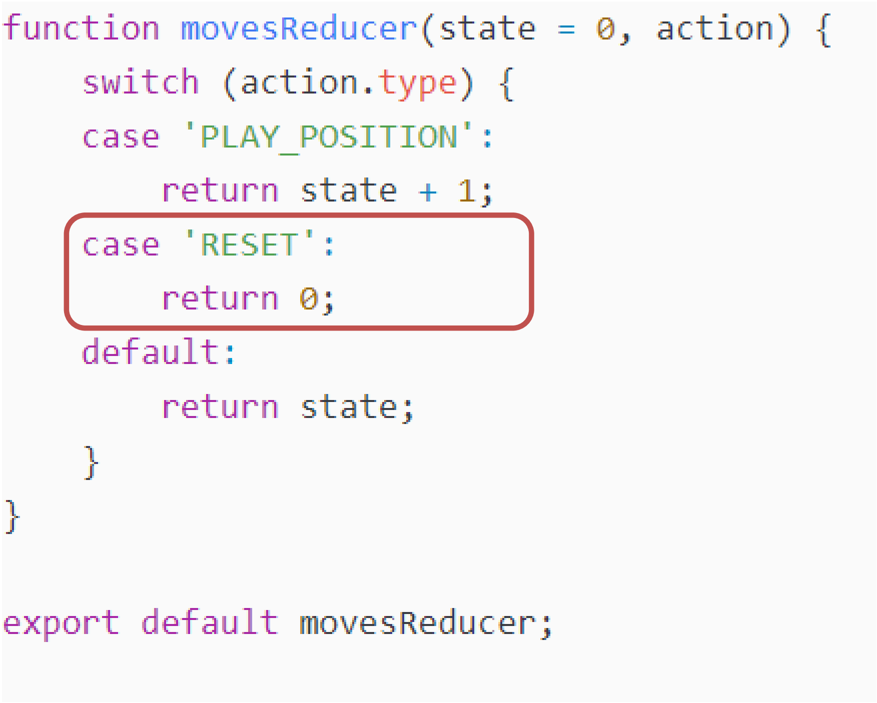
We have to reset the moves to 0



```
function movesReducer(state = 0, action) {
    switch (action.type) {
    case 'PLAY_POSITION':
        return state + 1;
    case 'RESET':
        return 0;
    default:
        return state;
    }
}

export default movesReducer;
```

9. Extra - New reset action
app/reducers/movesReducer.jsx

We have to reset the moves to 0



```
function movesReducer(state = 0, action) {
    switch (action.type) {
    case 'PLAY_POSITION':
        return state + 1;
    case 'RESET':
        return 0;
    default:
        return state;
    }
}

export default movesReducer;
```

9. Extra - New reset action

app/components/App.jsx

Import the new action

```
import { playPosicion, reset } from './../reducers/actions';
```

Modify resetClick() to dispatch the action

```
 resetClick() {
    this.props.dispatch(reset());
 }
```

8. Extra new reset action - *app/components/App.jsx*

```
import React from 'react';
import Header from './Header.jsx';
import Board from './Board.jsx';
import Reset from './Reset.jsx';
import { connect } from 'react-redux';
import { playPosition, reset } from './../reducers/actions';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.appClick = this.appClick.bind(this);
    this.resetClick = this.resetClick.bind(this);
  }
  appClick(rowNumber, columnNumber) {
      this.props.dispatch(playPosition(rowNumber, columnNumber, this.props.turn, this.props.values));
  }
  resetClick() {
      this.props.dispatch(reset());
  }
  render() {
    let text = "Turn of " + this.props.turn;
    return (
      <div>
        <Header text={text} winner={this.props.winner}/>
        <Board values={this.props.values}  appClick={this.appClick} winner={this.props.winner}/>
       <Reset resetClick={this.resetClick}></Reset>
      </div>
    );
  }
}

function mapStateToProps(state) {
    return {
       values: state.values,
        turn: state.turn,
    };
}
export default connect(mapStateToProps)(App);
```

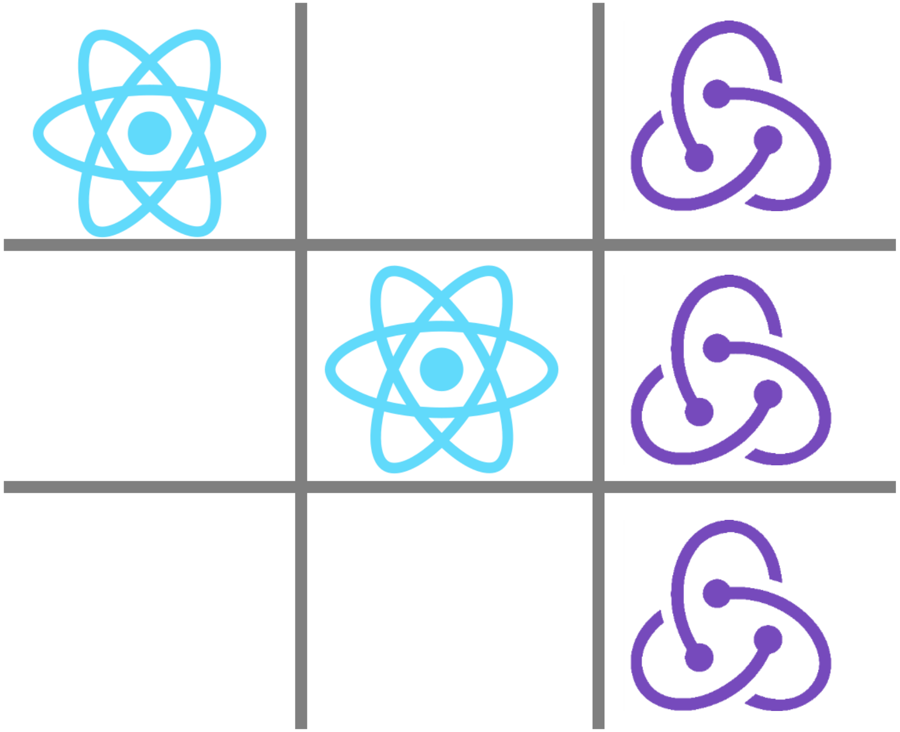
9. Extra - New reset action

summary

Each time we add a new action:

1. Define the action in *actions.jsx*: *type* and params
2. Modify the reducers so they take into account the new action (with a new case statement) if needed
3. Launch the new action from App.jsx or the component that dispatchs it

THE END



Adding Redux to the Tic Tac Toe

Enrique Barra