

Redux

Gopalakrishnan Subramani

www.nodesen.se

gs@nodesen.se

+91 9886991146

Introduction

**Redux
State/Data
Management**

**React
View/Presentation**

What is in React

- Component
- Render
- State
- Props
- JSX

What is in Redux

- Action
- Action Creator
- Dispatch
- Reducer
- Store

Redux

- Redux is a framework for managing the state for a web application,
- React components render that state
- A single data store contains the state for your app
- Your application emits an action, that defines something that just happened that will affect the state
- Reducers specify how to change the state when the action is received
- Hot reloading of code changes
- State changes can be tracked, and replayed

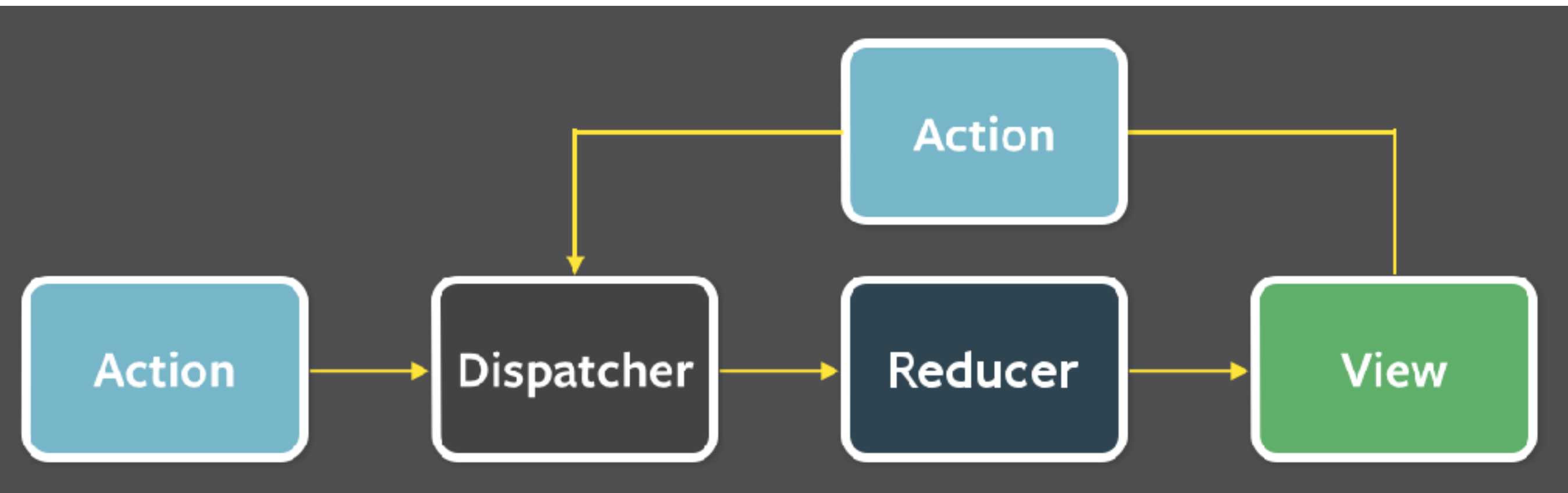
Redux

- Redux is a Flux implementation
- Minimal APIs
- Holds application state
- Allows access to state via `getState()`;
- Allows state to be updated via `dispatch(action)`;
- Registers listeners via `subscribe(listener)` (to update view)
- Handles unregistering of listeners via the function returned by `subscribe(listener)`

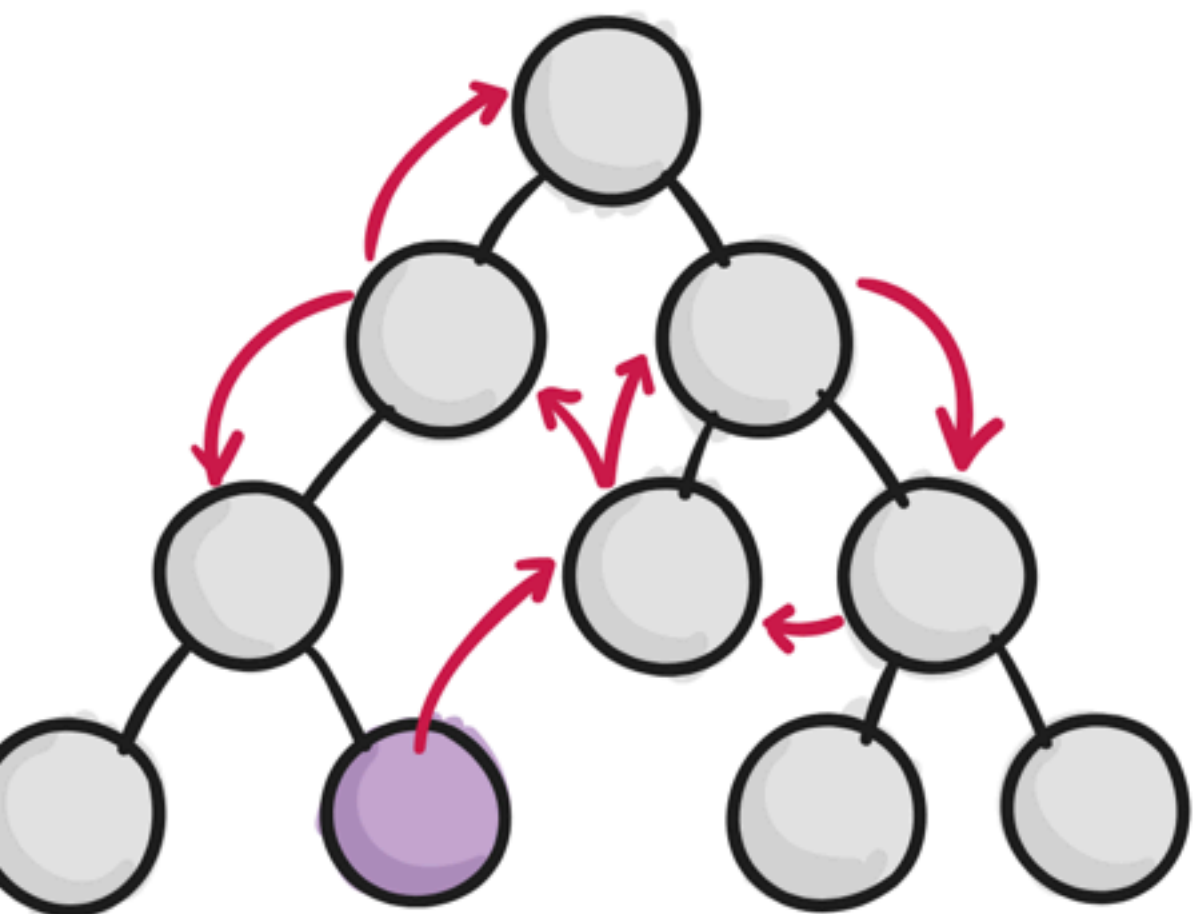
Functional Programming

- Keeping React Components as View only
- De-couple states from React Component
- Calling React view with same input must produce same output (mean, no states maintained at React)
- Predictable outcome, easy to test

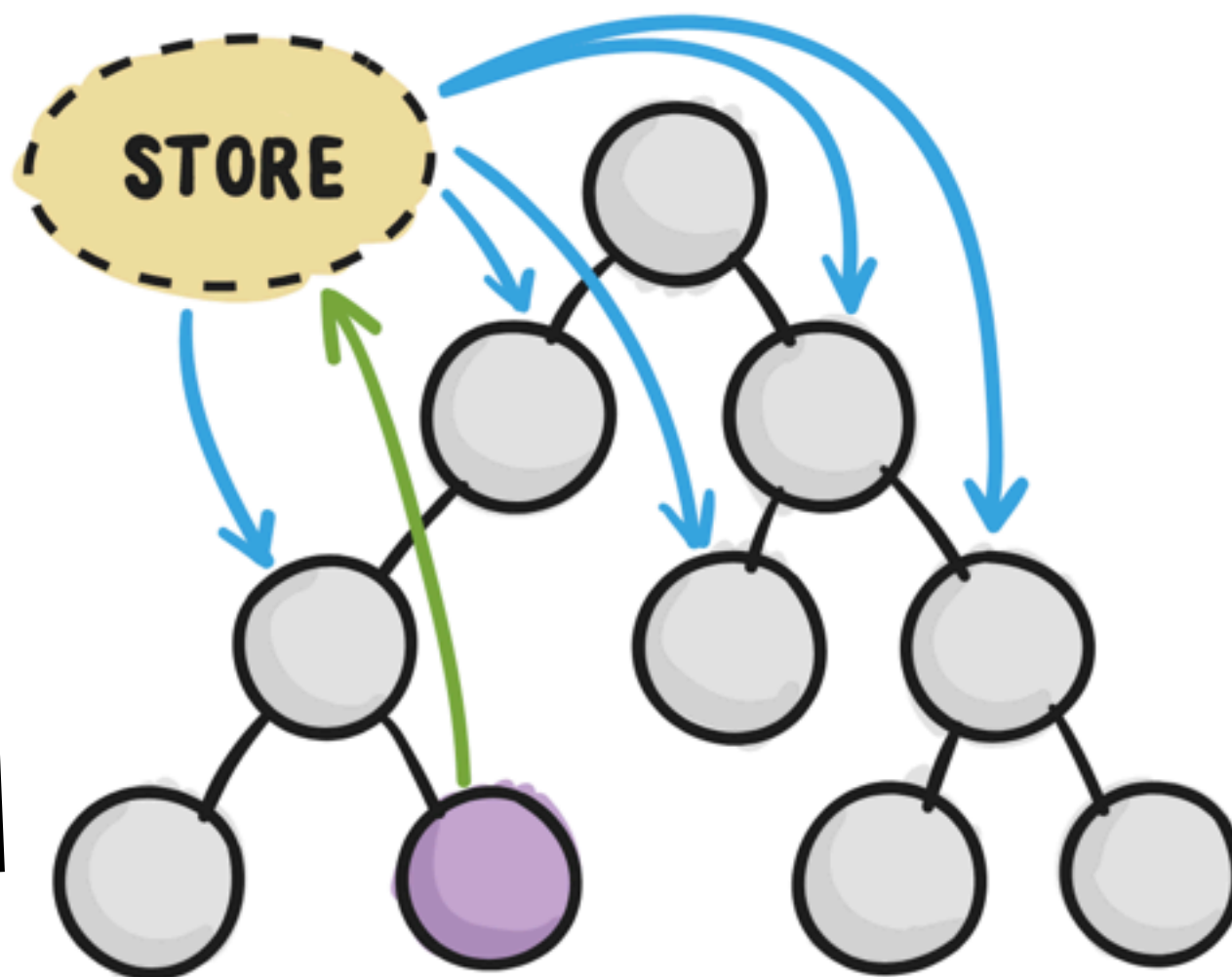
FLUX

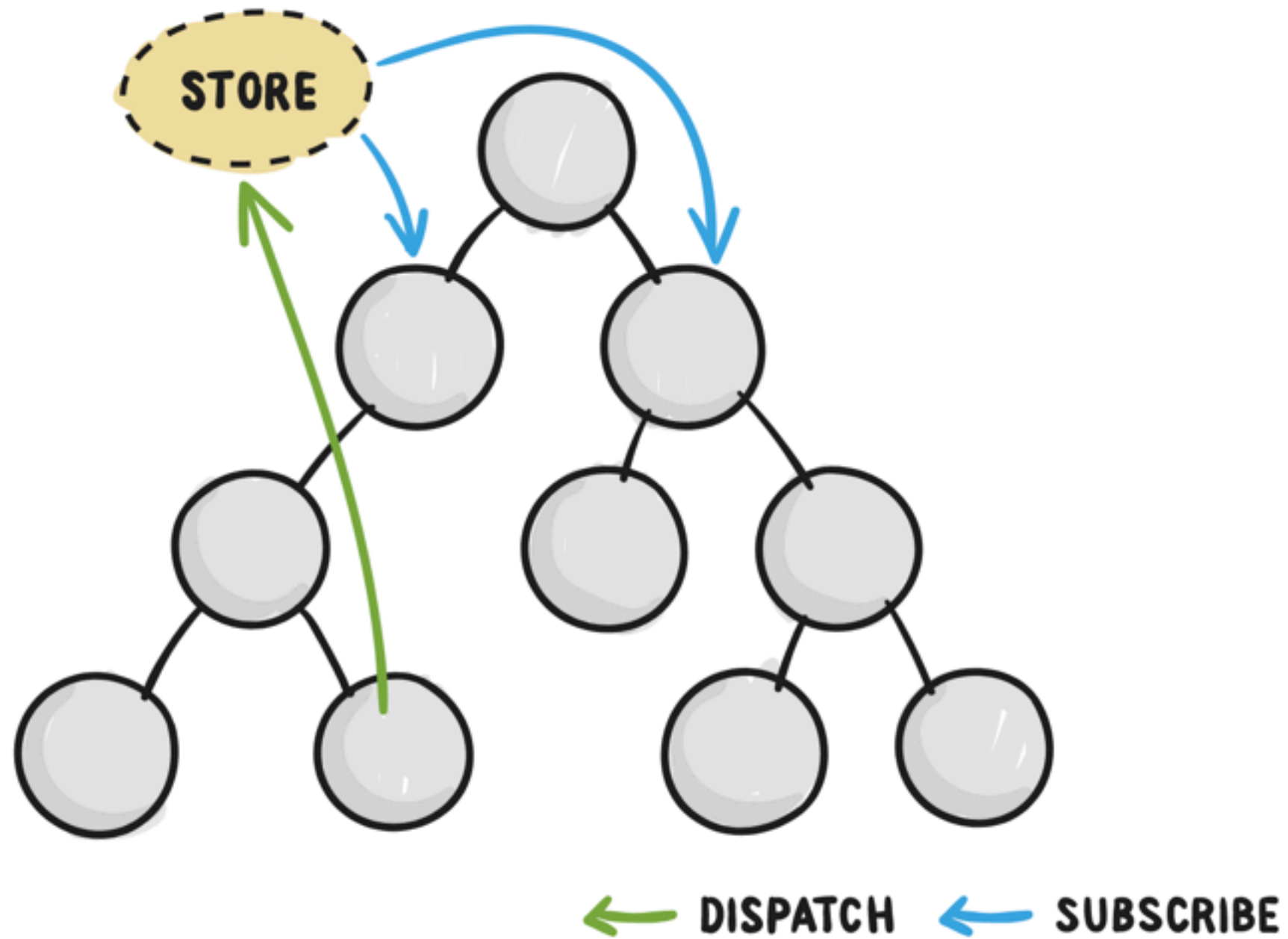


Without Redux



With Redux

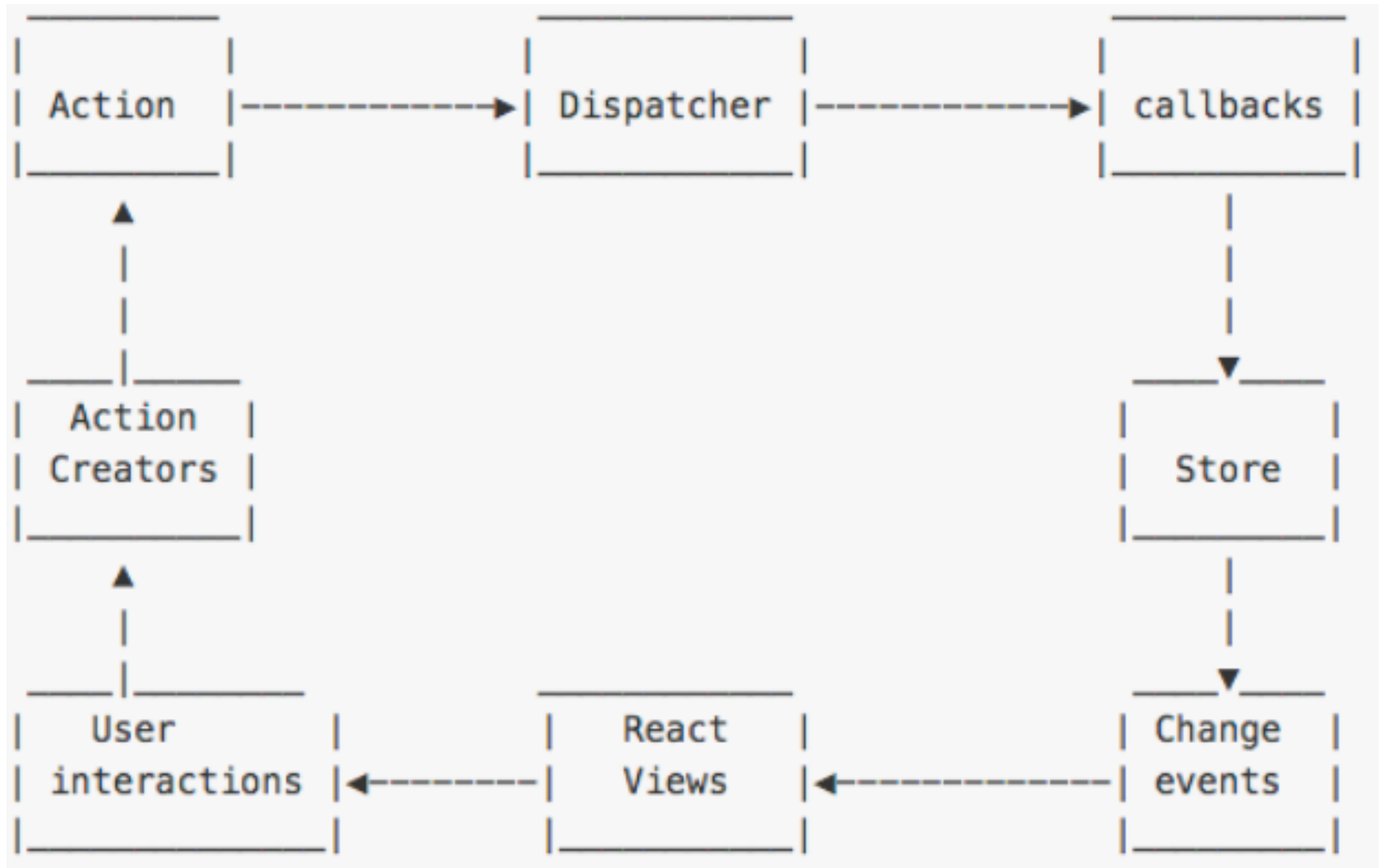




React + Redux

- The props for React components come from the Redux store that tracks the state.
- React components react to user input and emit actions, either directly or indirectly.
- Redux handles the action by running the appropriate reducers which transform the current state into a new state.
- React components react to the new state and update the DOM.
- React components themselves are stateless (most of the time), all of the state is kept in the Redux store, one common place, for simplicity

Redux + React Workflow



Reducers

- Manages application state changes
- Reducers called with current state with action
- Redux stores all the data in one single object
- $(previousState, action) \Rightarrow newState$
- Every Action produces new state, i.e. you should not mutate the state

Reducers

- Given the same arguments, it should calculate the next state and return it.
- No surprises.
- No side effects.
- No API calls.
- No mutations. Just a calculation.

Reducer

```
INITIAL_STATE = 0
```

```
function reducer(state=INITIAL_STATE, action) {
```

```
    switch(action.type) {
```

```
        case "INCREMENT": {
```

```
            return state + action.value
```

```
        }
```

```
        case "DECREMENT": {
```

```
            return state - action.value
```

```
        }
```

```
    default:
```

```
        return state;
```

```
    }
```

```
}
```

Reducer with List

```
const INITIAL_STATE = []
function cartReducer(state = INITIAL_STATE,
                    action) {
  switch(action.type) {
    case "ADD_TO_CART":
      return [...state, action.item]

    case "REMOVE_ITEM_FROM_CART":
      return state.filter (item => item.id !== action.id)

    case "EMPTY_CART":
      return []

    default:
      return state;
  }
}
```


Store

- Model/Data Management
- Manages the data for the application
- Updates data on events
- Notify the views after changes in data

Store

```
import {createStore} from "redux";  
  
let store = createStore(reducer);  
  
// or with existing seed/initial values  
let seed = 100;  
let store = createStore(reducer, seed);  
  
//to get last known state 100  
console.log(store.getState())
```

combineReducer

- One store can have only one reducer
- What if we have more than one reducer?
- Redux has combineReducer

combineReducer

```
import {createStore, combineReducers} from "redux";

let rootReducer = combineReducers ({
  counter: counterReducer,
  productState: productReducer,
  cartState: cartReducers
})

let store = createStore(rootReducer)

var state = store.getState();
//state.productState
//state.counter
//state.cartState
```

Actions

- Actions are payloads of information that send data from your application to your store.
- They are the only source of information for the store.
- You send them to the store using `store.dispatch()`.

Dispatch

Dispatch is the only way to call reducers

Dispatch dispatches actions to store. Store deliver to all reducers.

Upon receiving actions, reducers should respond with new state, if no state change, return existing state as it is.

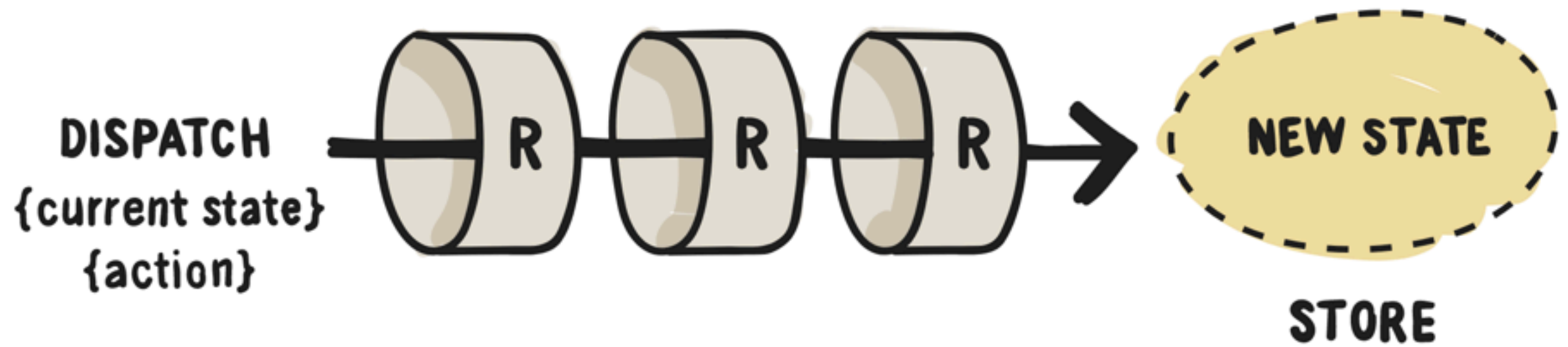
Dispatch

Syntax:

```
store.dispatch({action})
```

```
store.dispatch({  
  type: "INCREMENT",  
  value: 10  
})
```

```
store.dispatch({  
  type: "DECREMENT",  
  value: 5  
})
```





Action Creators

- Action creators are helper method, that create actions

Action Creators

```
function incrementAction(value) {  
  return {  
    type: "INCREMENT",  
    value: value  
  }  
}
```

```
function decrementAction(value) {  
  return {  
    type: "DECREMENT",  
    value  
  }  
}
```

```
}  
store.dispatch(incrementAction(10))  
store.dispatch(decrementAction(5))
```

Subscribe

- Any components interested in data from store can subscribe
- Subscribe is be called for every dispatch
- Subscription can be unsubscribed

Subscribe

```
let unsubscribe = store.subscribe ( () => {  
    console.log("updated values", store.getState());  
})
```

```
// at the end, don't fail to unsubscribe  
unsubscribe()
```

With React Component

```
componentDidMount() {  
    this.unsubscribe = store.subscribe( () => {  
        this.setState({  
            result: store.getState()  
        })  
    })  
}
```

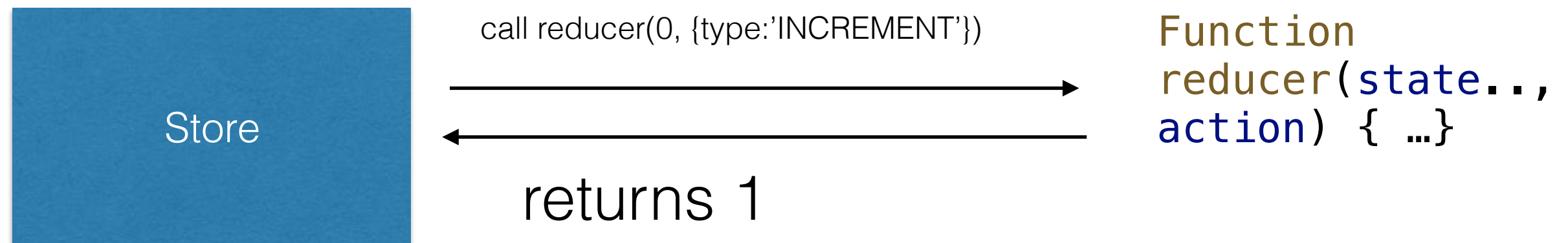
```
componentWillUnmount() {  
    if (this.unsubscribe)  
        this.unsubscribe();  
}
```

```

const INITIAL_STATE = 0;
function reducer(state = INITIAL_STATE,
                  action) {
  switch(action.type) {
    case "INCREMENT":
      return state + 1;

    default:
      return state;
  }
}

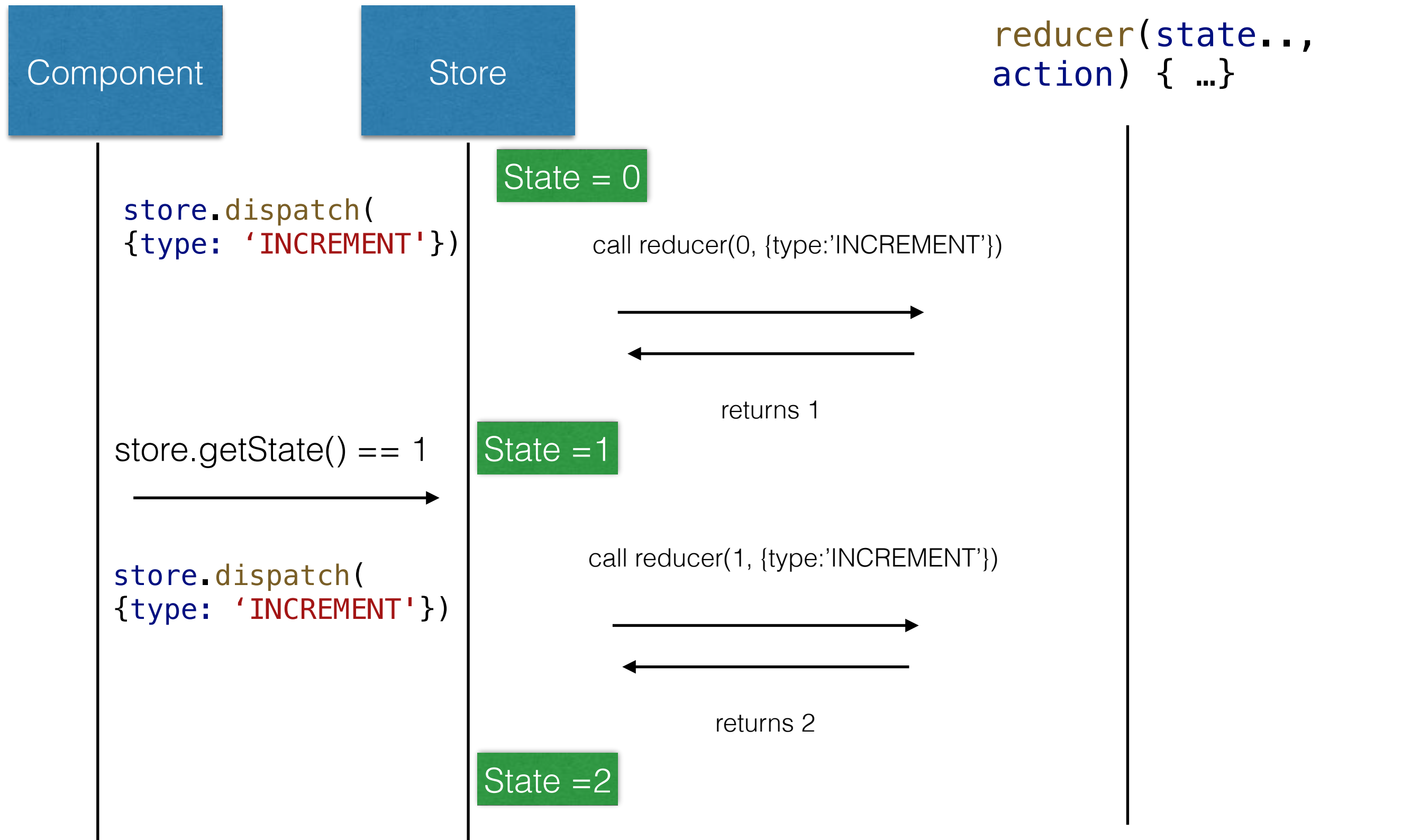
```



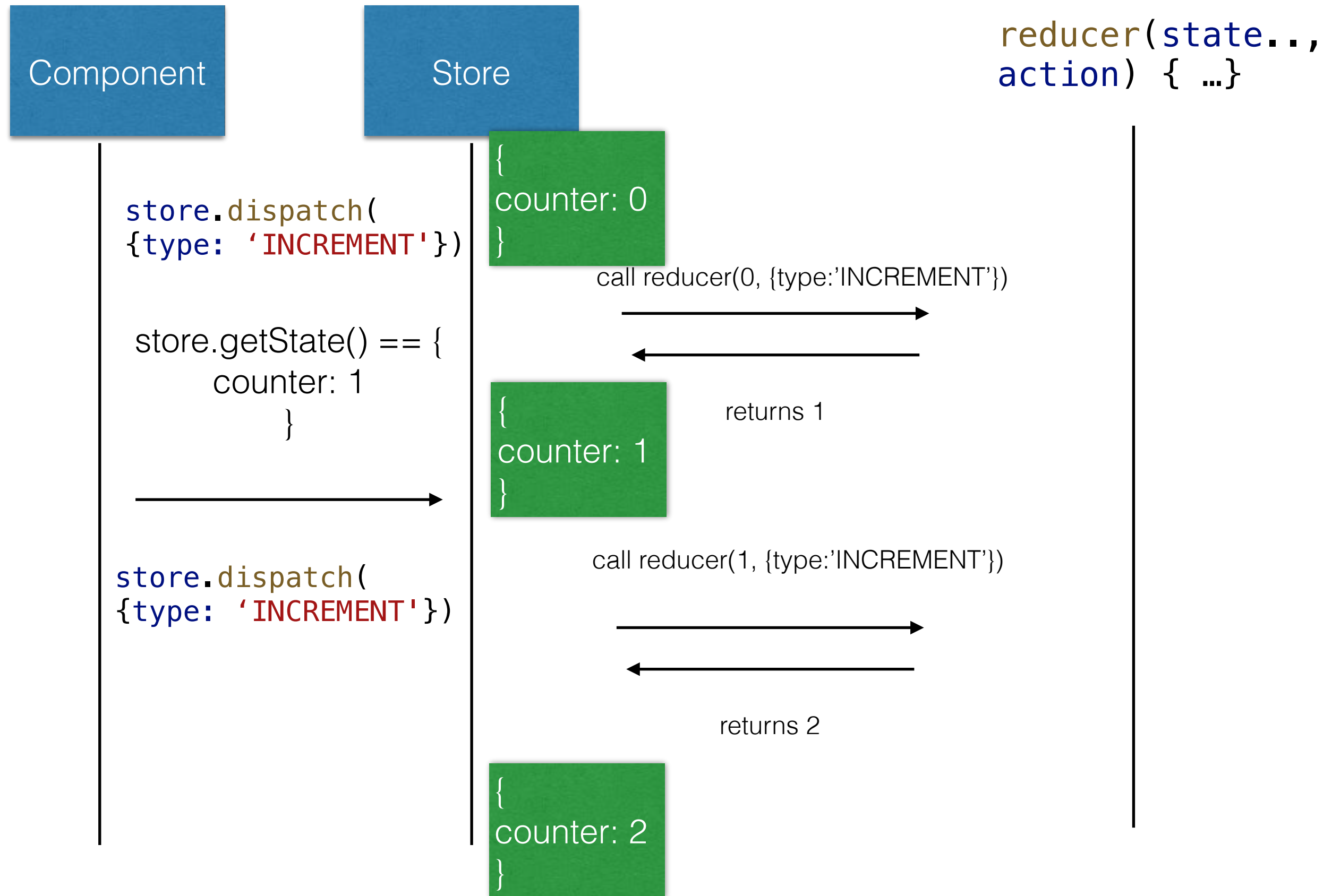
```

store.dispatch({type: 'INCREMENT'});

```

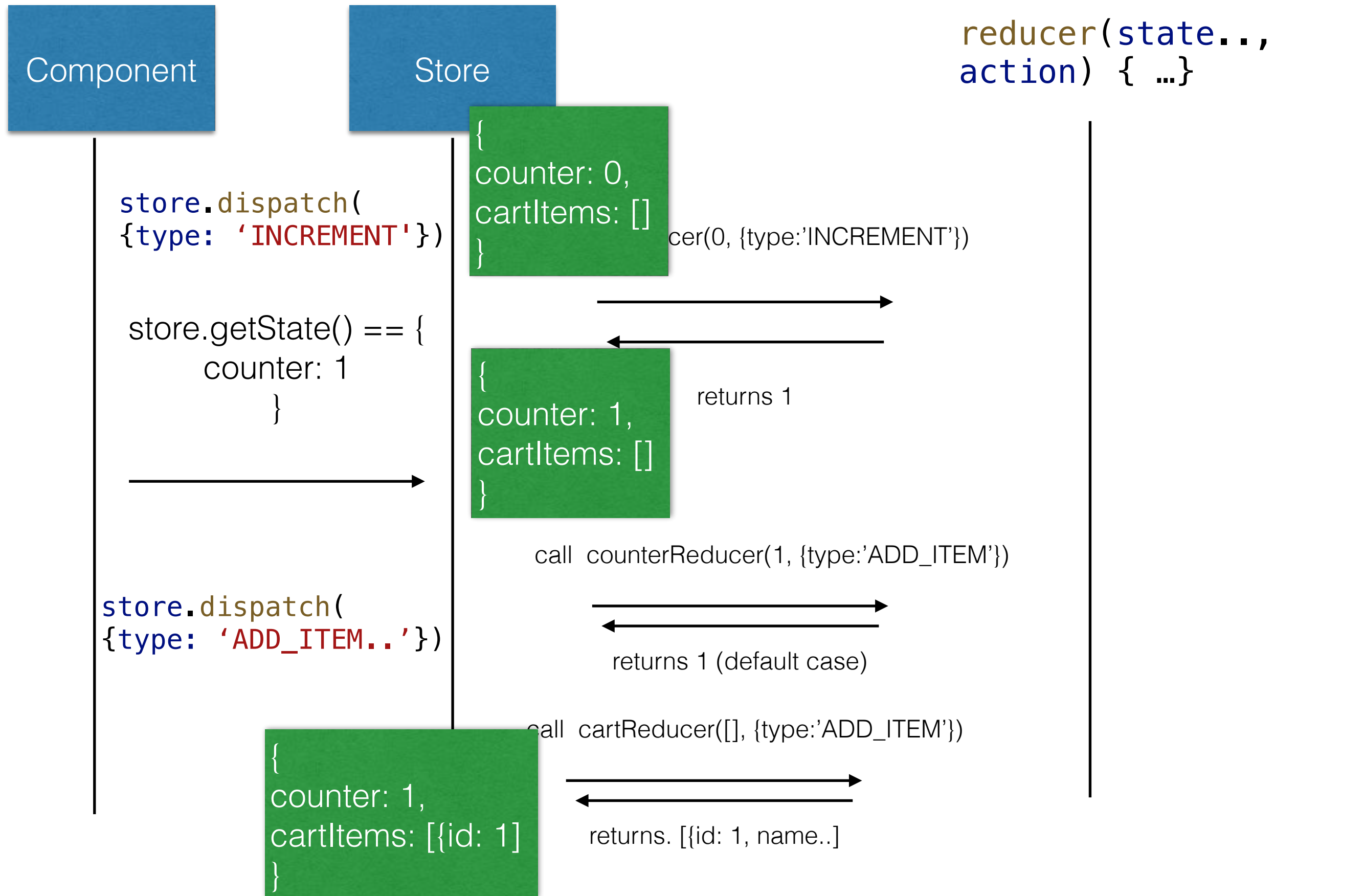


combine Reducer

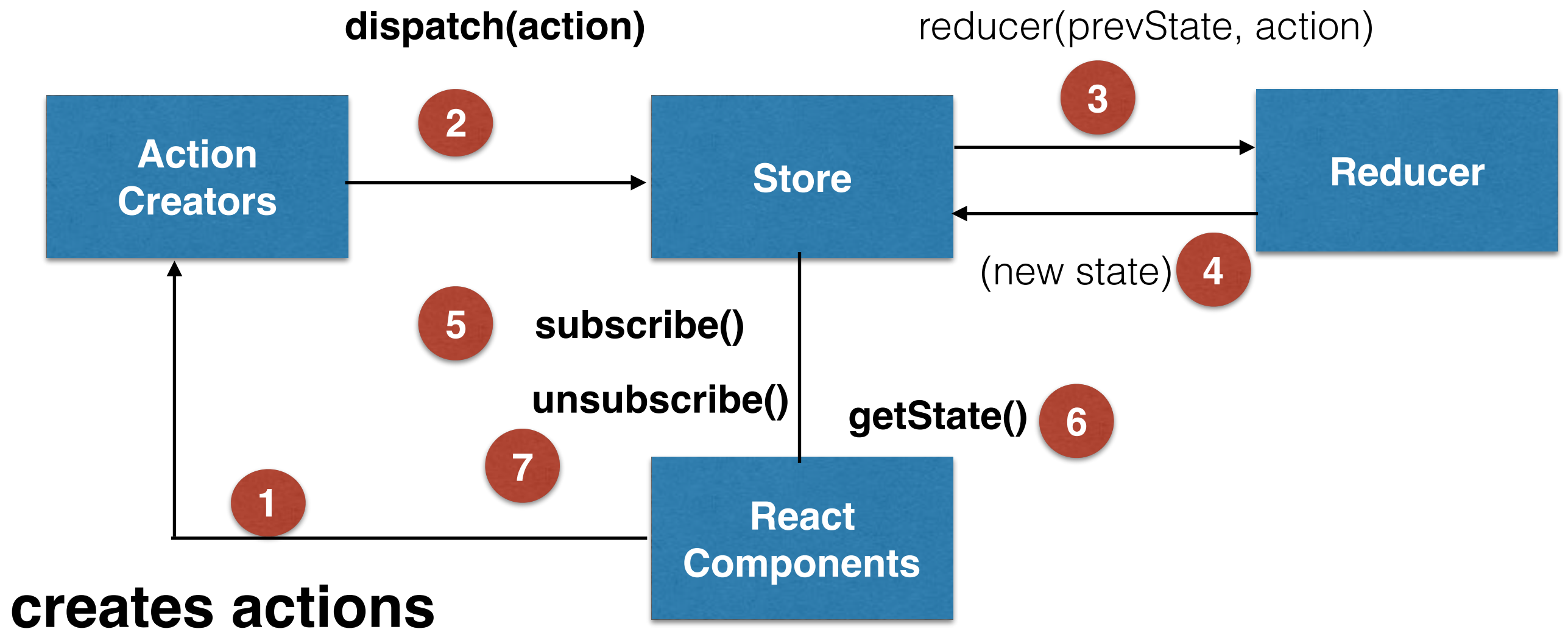


```
let rootReducer = combineReducers({  
  //stateName: reducer fn  
  counter: counterReducer,  
  cartItems: cartReducer  
  //  
})
```

combine Reducer



Redux Flow



Middleware

- Middleware are added between Dispatch and Reducers
- Often useful to log the data, perform actions with promises, modify data on the fly
- Middleware can stop action going to reducer as well

Example

```
const logger = store => next => action => {  
  console.log('dispatching', action)  
  let result = next(action)  
  console.log('next state', store.getState())  
  return result  
}
```

next(...) calls next middleware/reducer in the chain

Redux-thunk

```
npm install redux-thunk --save
```

For handling async actions and promises

Store with Middleware

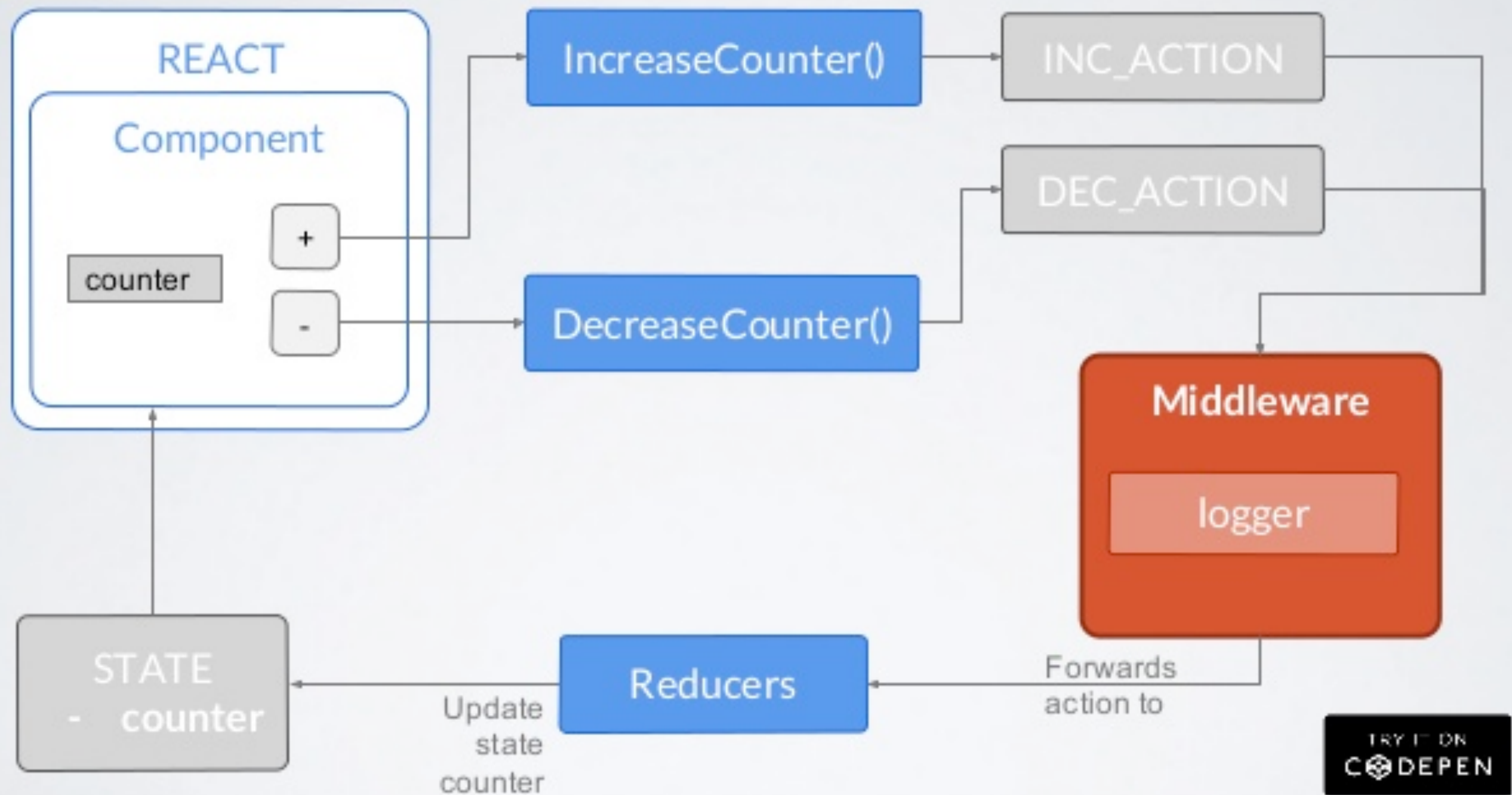
```
import {createStore, applyMiddleware} from "redux";  
import thunk from "redux-thunk"
```

```
let store = createStore(math_reducer,  
                          initialState,  
                          applyMiddleware(thunk,  
                                          logger));
```


Redux life-cycle with middlewares



Simplest example - logger



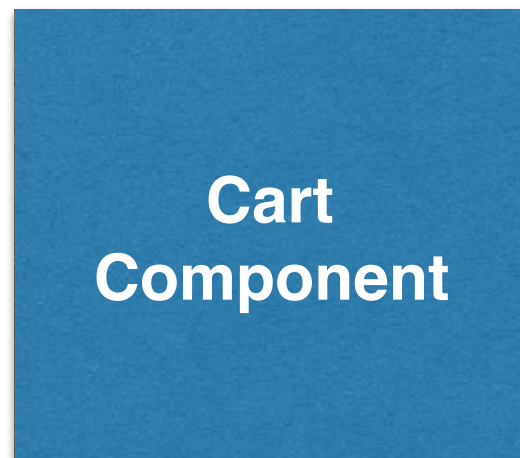
React + Redux

- react-redux library
- mapStateToProps
- mapDispatchToProps
- connect

React Redux

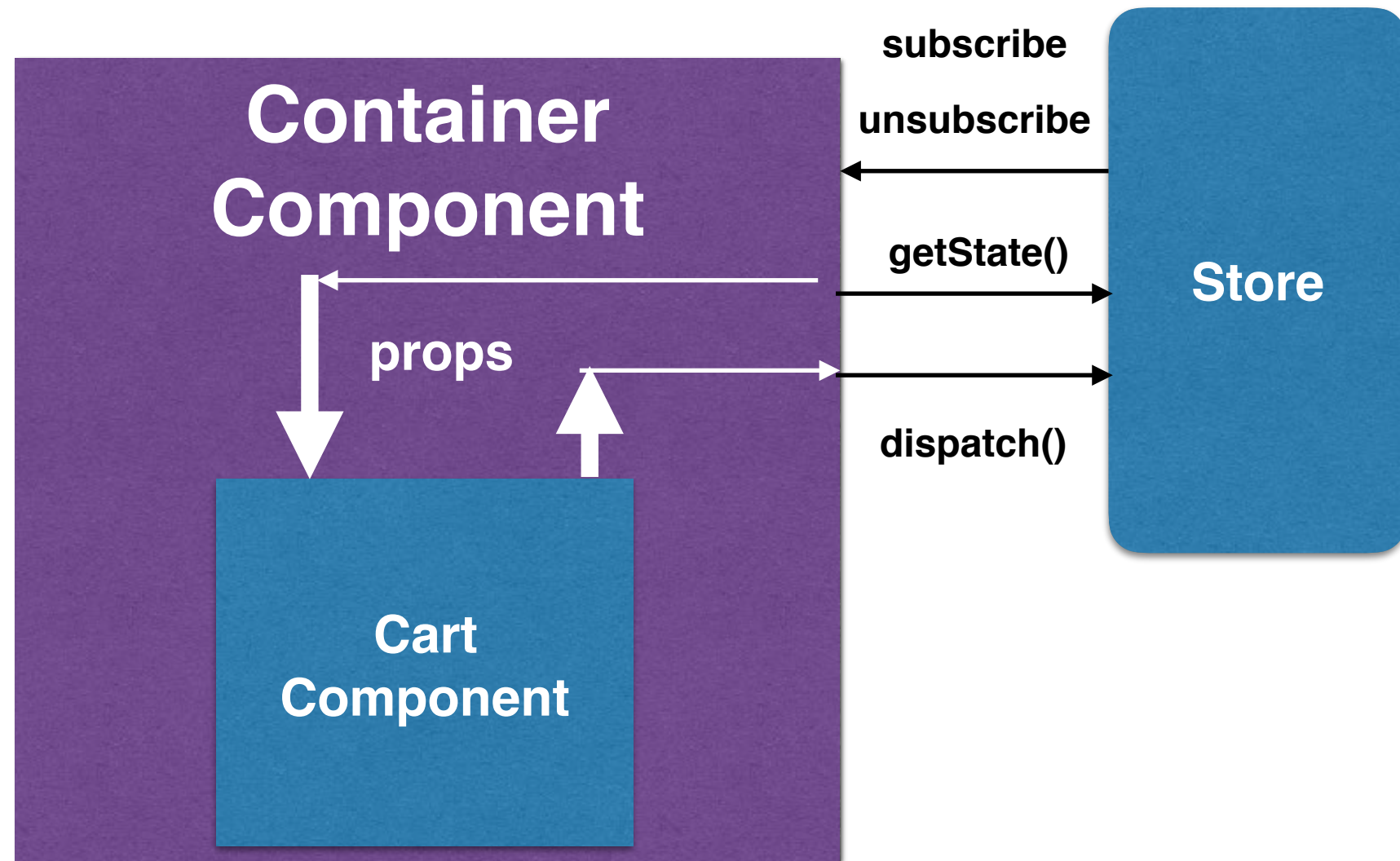
- Connects React and Redux
- Wrap the React components within its **container component**
- **Provides clear segregation between React and Redux, keep them independent**

React World



Your
Component

Redux World



Containers

React-Redux

- npm install react-redux —save
- Connect React and Redux

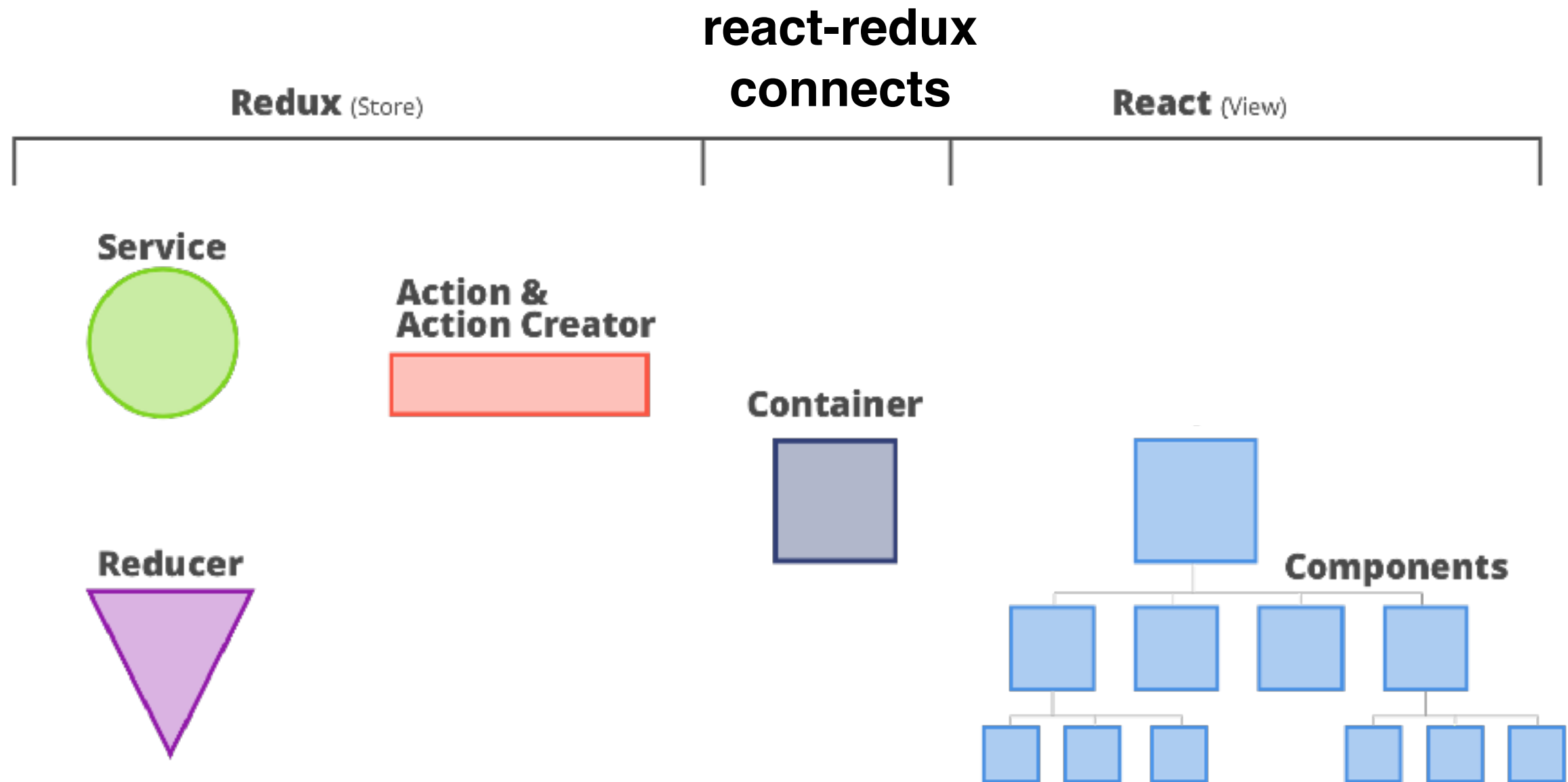
```
import {Provider} from "react-redux";
```

```
render( (  
    <Provider store={store}>  
        <App />  
    </Provider>  
), document.getElementById("root"));
```

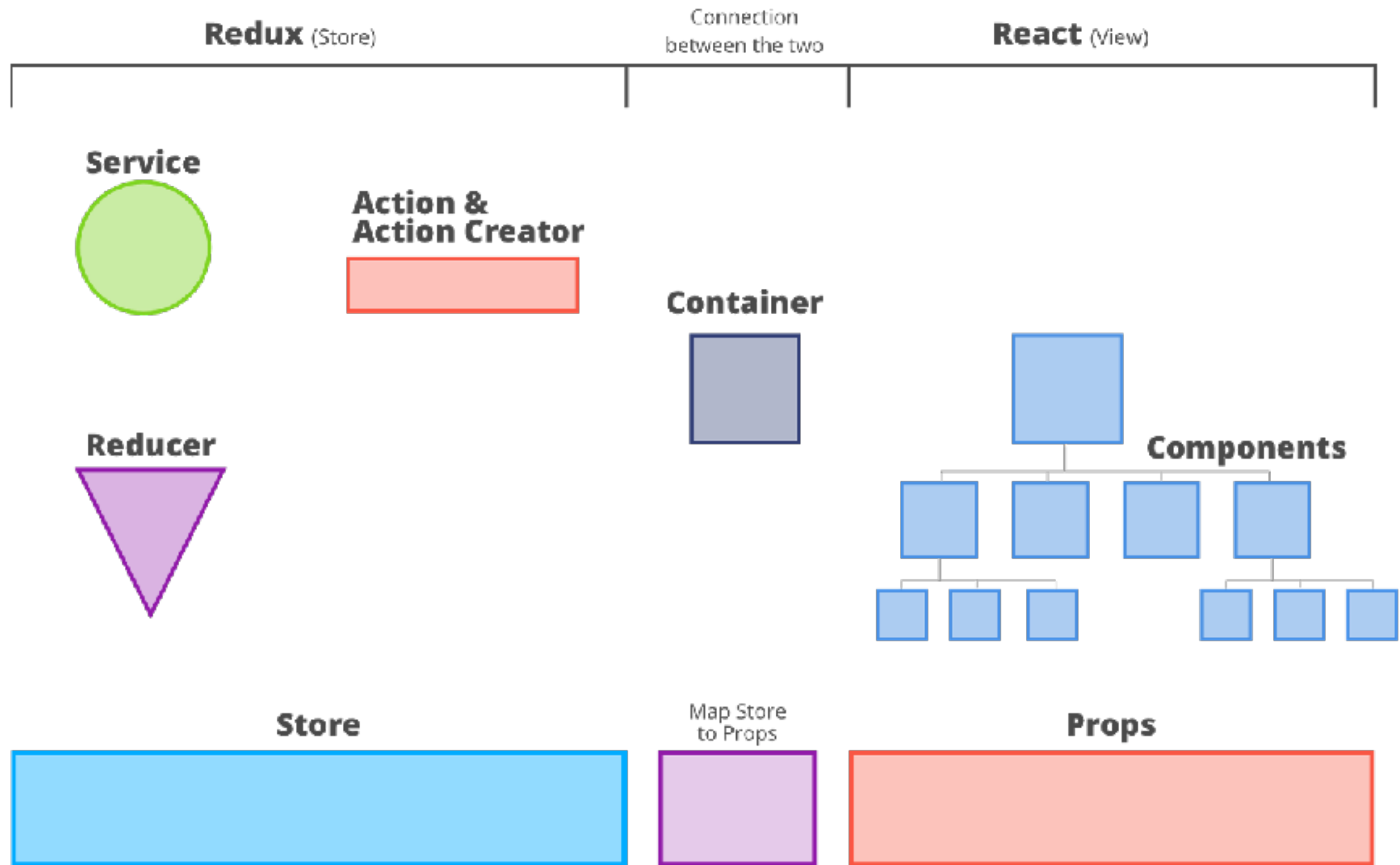
React-Redux

- React-Redux maps Redux State to Component Params or Presentation Views functional arguments
- React-Redux maps Redux Store Dispatch to props
- Finer Abstraction over Redux while using React Components

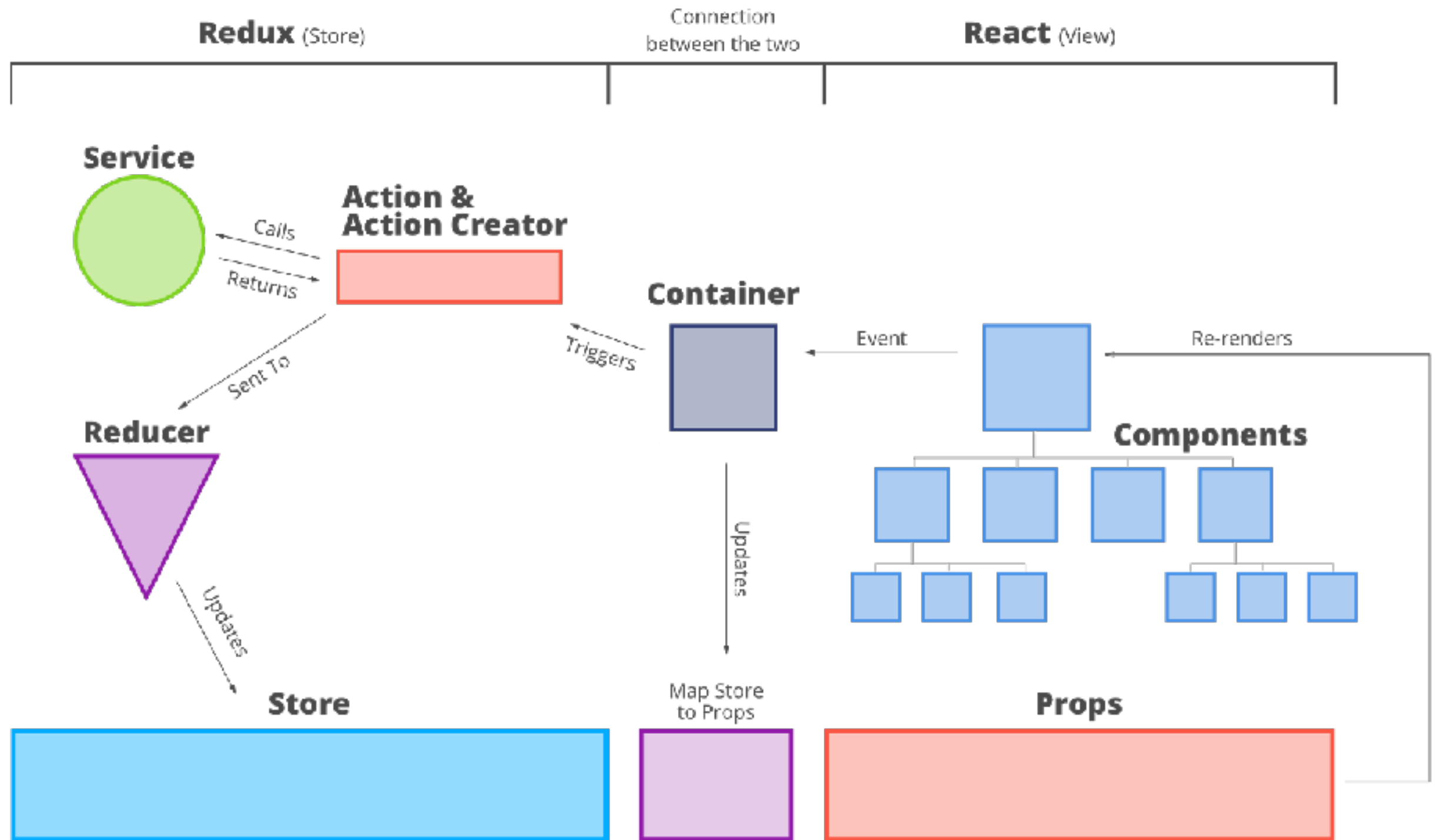
React Redux



React Redux



React Redux



```
import React, {PropTypes} from "react";
import {connect} from "react-redux";

class Cart extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.items.length}</h1>
        <button onClick={this.props.emptyCart}>
          Empty Cart
        </button>
      </div>
    )
  }
}

Cart.propTypes = {
  items: PropTypes.array
}
```

```
const mapStatesToProps = (state) => {  
  return {  
    items: state.cartState  
  }  
}  
  
const mapDispatchToProps = (dispatch) => {  
  return {  
    emptyCart: () => {  
      dispatch({type: 'EMPTY'});  
    }  
  }  
}
```

```
CartContainer = connect(mapStatesToProps,  
  mapDispatchToProps)(Cart);
```

```
export default CartContainer;
```