REACT JS

Redux Introduction

Sreekanth M. E.

Freelance Trainer & Consultant

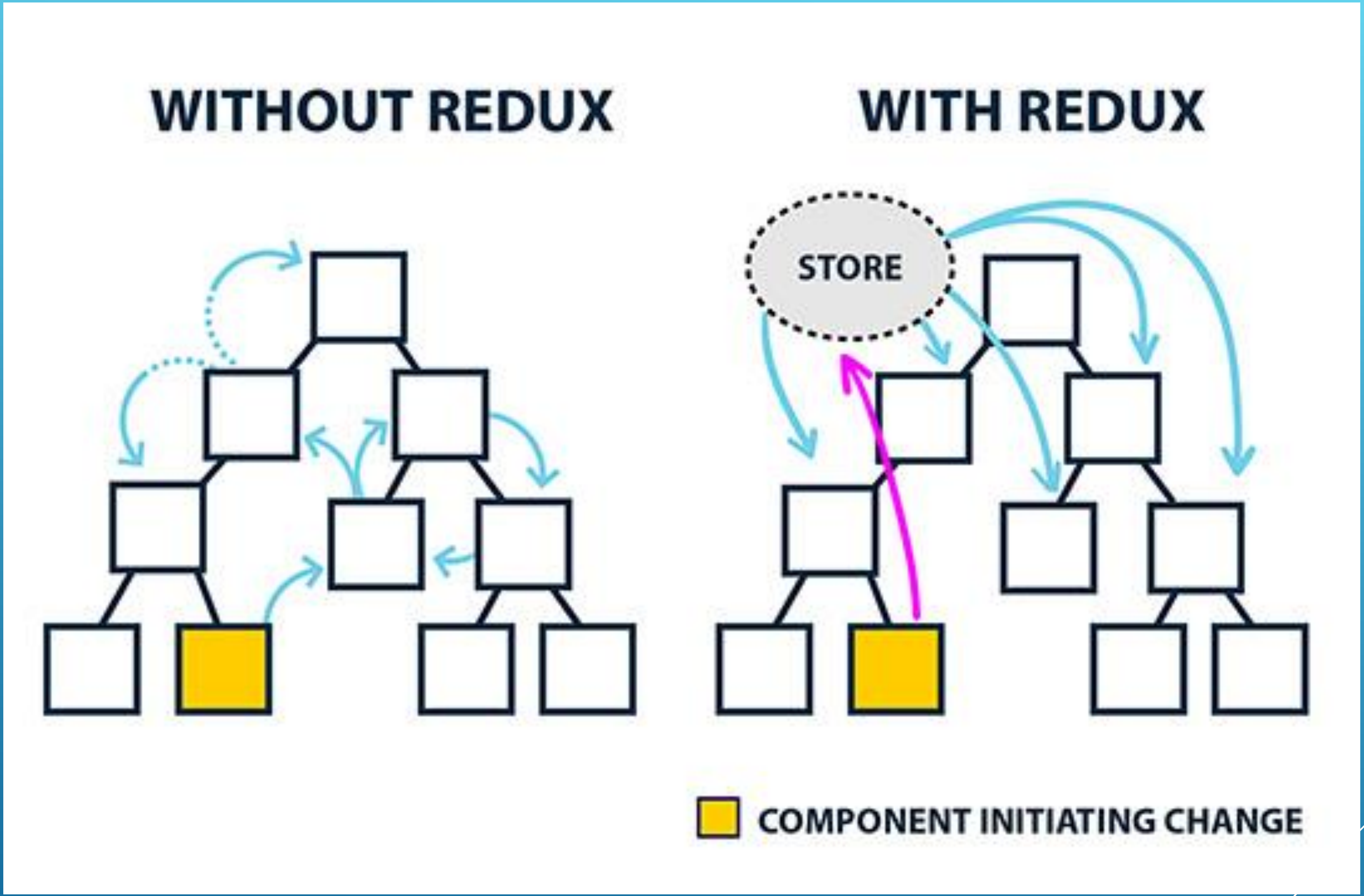
http://www.SreekanthME.com

Redux is a predictable state container for JavaScript apps.

Redux can be used with React, or with any other view library.

React is just a view library.

With redux, React gets the FLUX pattern for state management.


WITHOUT REDUX
WITH REDUX
STORE
COMPONENT INITIATING CHANGE

With Redux, all component level state is completely eliminated and all state is centralized in something called Redux store.

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

For direct browser use: https://unpkg.com/redux@latest/dist/redux.min.js

For simple SPA with a few components, coordinating components is possible with just React state.

But for large SPA with several components, a state management system like Redux becomes essential.

Redux works on three key principles.

1. SINGLE SOURCE OF TRUTH

The state of the entire SPA is stored in a single JS object called store or state tree.

A single state tree makes it easier to debug or inspect an application.

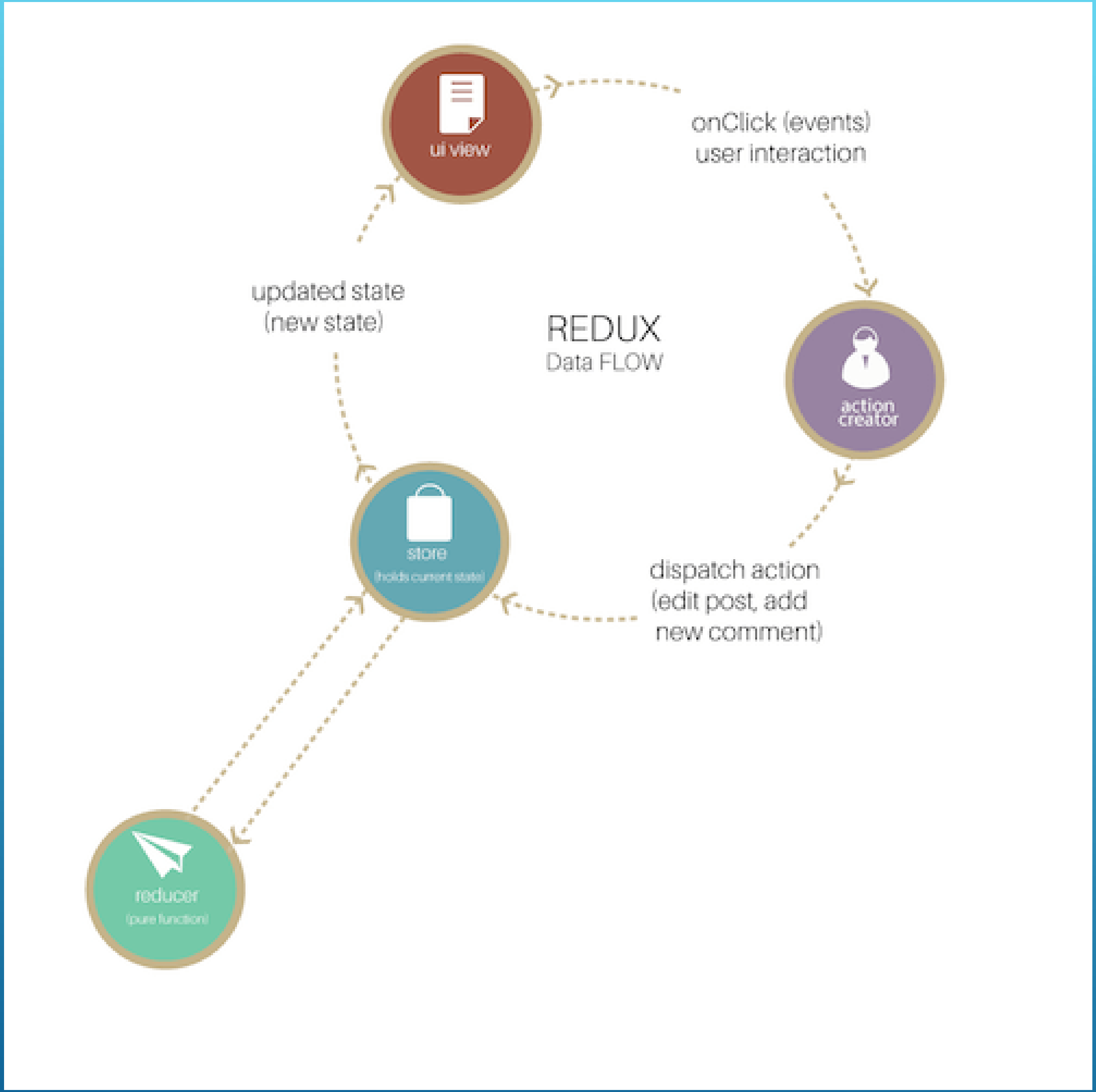
2. STATE IS READ-ONLY

State cannot be modified directly.

The only way to change the state is to emit an action (an object describing what happened).

3. STATE CHANGES ARE VIA PURE FUNCTIONS

State tree is updated by pure functions named reducers.


ui view
onClick (events)
user interaction
action creator
dispatch action
(edit post, add
new comment)
store
(holds current state)
reducer
(pure function)
updated state
(new state)
REDUX
Data FLOW

ACTIONS

Actions are plain JS objects of information that send data from UI to the store.

Actions MUST have a type property that indicates the type of action being performed.

Other than type, the structure of an action object is really up to the developer. Though, it is typical to have a property named payload.

```
{ type: ‘LIKE_ARTICLE’,
articleId: 42 }


 { type: ‘FETCH_USER_SUCCESS’,
payload: { id: 3, name: ‘Mary’ } }


 { type: ‘ADD_TODO’,
text: ‘Read the Redux docs.’ }
```

Actions are sent to the store using store.dispatch(action) method.

ACTION CREATOR

Action creators are functions that return an action object.

```
function addTodo(textVal) {
  return {
    type: ‘ADD_TODO’,
    text: textVal
  }
}
```

To initiate a dispatch, the action (returned by action creators) is passed to the dispatch() function:

```
store.dispatch(addTodo(text))
store.dispatch(completeTodo(index))
```

Action creator functions can be asynchronous and impure.

REDUCERS

Store is created from one or more reducers.

```
Redux.createStore(reducer)

Redux.createStore(combineReducers({reducer1,
reducer2, …}))
```

Actions received by the store through the store.dispatch() method are passed on to all the reducers from which the store has been created.

Reducers are just pure functions that take the previous state and an action, and return the next state calculated by them in immutable fashion.

```
(previousState, action) => newState
```

Each reducer manages independent parts of the SPA state.

The combineReducers() helper function creates a single reducer that calls every child reducer, and gathers their results into a single state object like so.

```
{
  reducer1: ...
  reducer2: ...
}
```

A reducer manages its own part of the store.

The state parameter of the reducer is different for every reducer, and corresponds to the part of the state it manages.

When a store is created, Redux calls all the reducers to populate the store with the initial state.

So each reducer should return some kind of initial state if the state given to it as the first argument is undefined.

store.subscribe(listener): It needs to be passed a listener or callback. The callback is invoked whenever the state tree changes by an action.

store.getState(): Returns the current state tree of the SPA.

END OF CHAPTER

APPENDIX

REDUX DATA LIFECYCLE:

The data lifecycle in any Redux app follows these 4 steps:

1. You call store.dispatch(action).

An action is a plain object describing what happened. You can call store.dispatch(action) from anywhere in your app, including components and XHR callbacks, or even at scheduled intervals.

2. The Redux store calls the reducer function you gave it.

The store will pass two arguments to the reducer: the current state tree and the action.

Note that a reducer is a pure function. It only computes the next state. It should be completely predictable: calling it with the same inputs many times should produce the same outputs. It shouldn't perform any side effects like API calls or router transitions. These should happen before an action is dispatched.

3. The root reducer may combine the output of multiple reducers into a single state tree.

How you structure the root reducer is completely up to you. Redux ships with a combineReducers() helper function, useful for “splitting” the root reducer into separate functions that each manage one branch of the state tree.

4. The Redux store saves the complete state tree returned by the root reducer.

This new tree is now the next state of your app! Every listener registered with store.subscribe(listener) will now be invoked; listeners may call store.getState() to get the current state.

Now, the UI can be updated to reflect the new state.