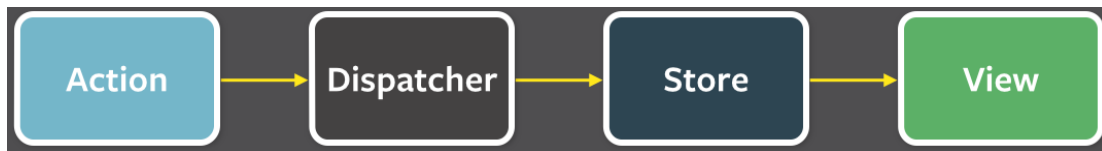


SECTION 7: FLUX

FLUX

- 🔔 Flux is the application architecture created
- 🔔 Unidirectional data flow is central to the Flux pattern
- 🔔 Flux has four roles: actions, stores, the dispatcher and views.
- 🔔 Single directional data flow makes it easy to understand and modify an application as it becomes more complicated.
- 🔔 Two-way data bindings lead to cascading updates, where changing one data model led to another data model updating, making it very difficult to predict what would change as the result of a single user interaction.



FLUX ROLES: ACTION

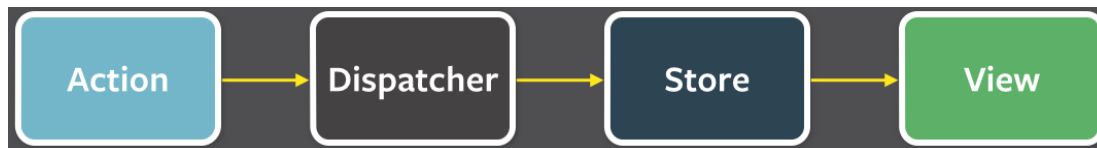
🔔 Flux has four roles: actions, stores, the dispatcher and views.

🔔 Action - an object literal containing the new fields of data and a specific action type

🔔 Different actions are identified by a type attribute

🔔 Action responsibilities can be defined as following:

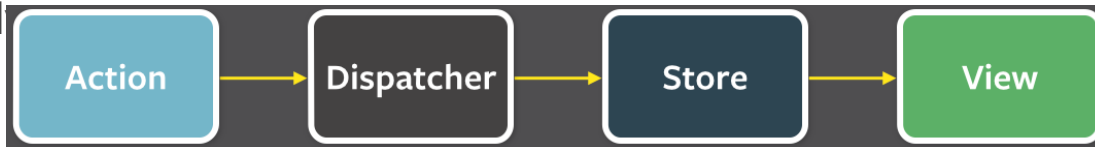
- ▶ Actions are simple objects with a type property and some data. For example, an action could be `{“type”: “IncreaseCount”, “local_data”: {“delta”: 1}}`



FLUX ROLES: STORE

🔔 Store responsibilities can be defined as following:

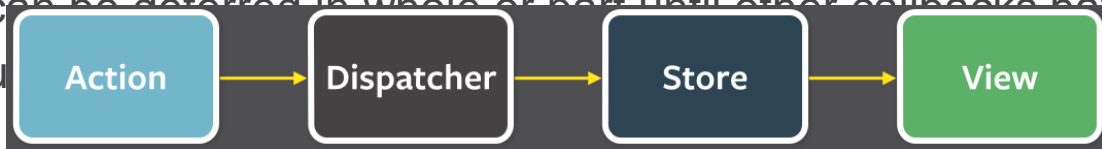
- Stores contain the application's state and logic.
- The best abstraction is to think of stores as managing a particular domain of the application.
- They aren't the same as models in MVC since models usually try to model single objects, while stores in Flux can store anything
- Store is updated with callback from dispatcher. Callback receives the action as a parameter
- After the stores are updated, they broadcast an event declaring that their state has changed, so the views may query the new state and update themselves



FLUX ROLES: DISPATCHER

🔔 Dispatcher responsibilities can be defined as following:

- The Dispatcher acts as a central hub. The dispatcher processes actions (for example, user interactions) and invokes callbacks that the stores have registered with it.
- The dispatcher isn't the same as controllers in the MVC pattern—usually the dispatcher does not have much logic inside it and you can reuse the same dispatcher across projects
- This is also different from generic pub-sub systems in two ways:
 - Callbacks are not subscribed to particular events. Every payload is dispatched to every registered callback.
 - Callbacks can be deferred in whole or part until other callbacks have been executed



FLUX ROLES: VIEWS

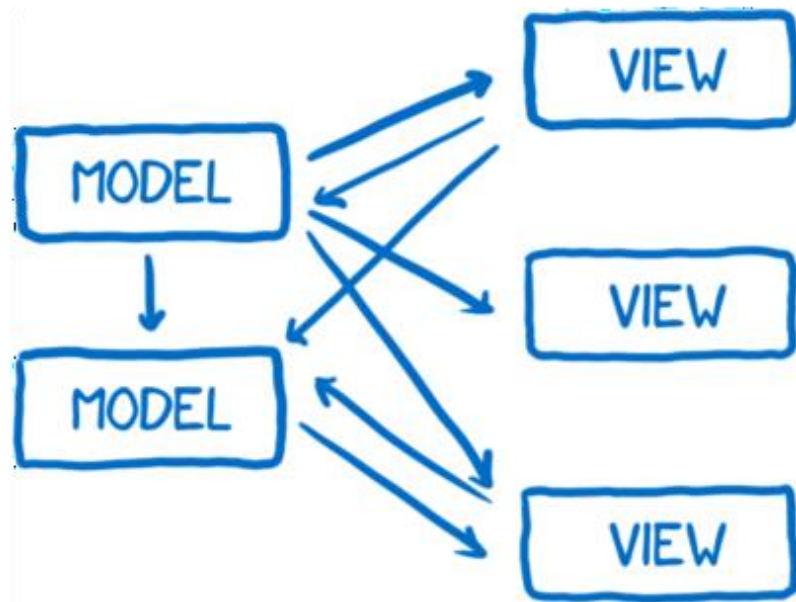
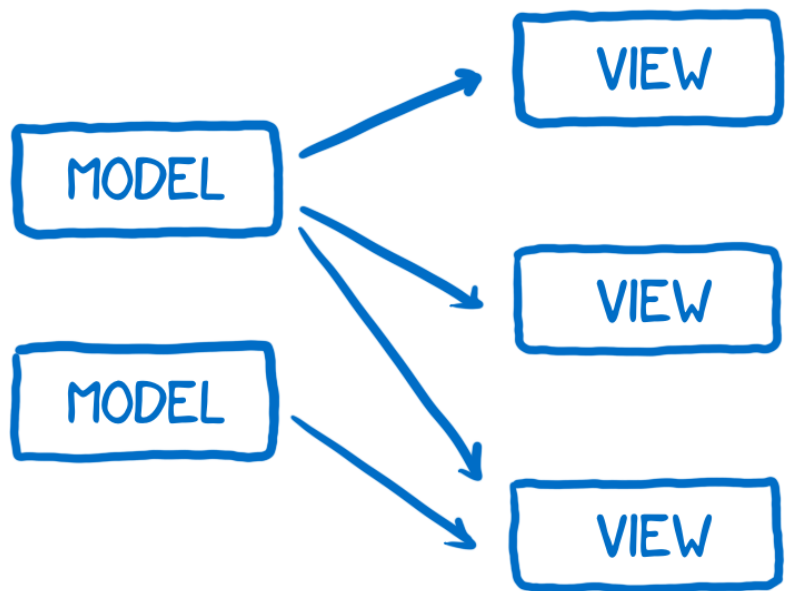
🔔 Views responsibilities can be defined as following:

- Views generally look like usual React Components
- At the same time they are controller-views, also very common in most GUI MVC patterns. They listen for changes from the stores and re-render themselves appropriately.
- Views can also add new actions to the dispatcher, for example, on user interactions.
- The views are usually coded in React, but it's not necessary to use React with Flux
- This works especially well with React's declarative programming style, which allows the communication between the components and the data.



FLUX WHY?

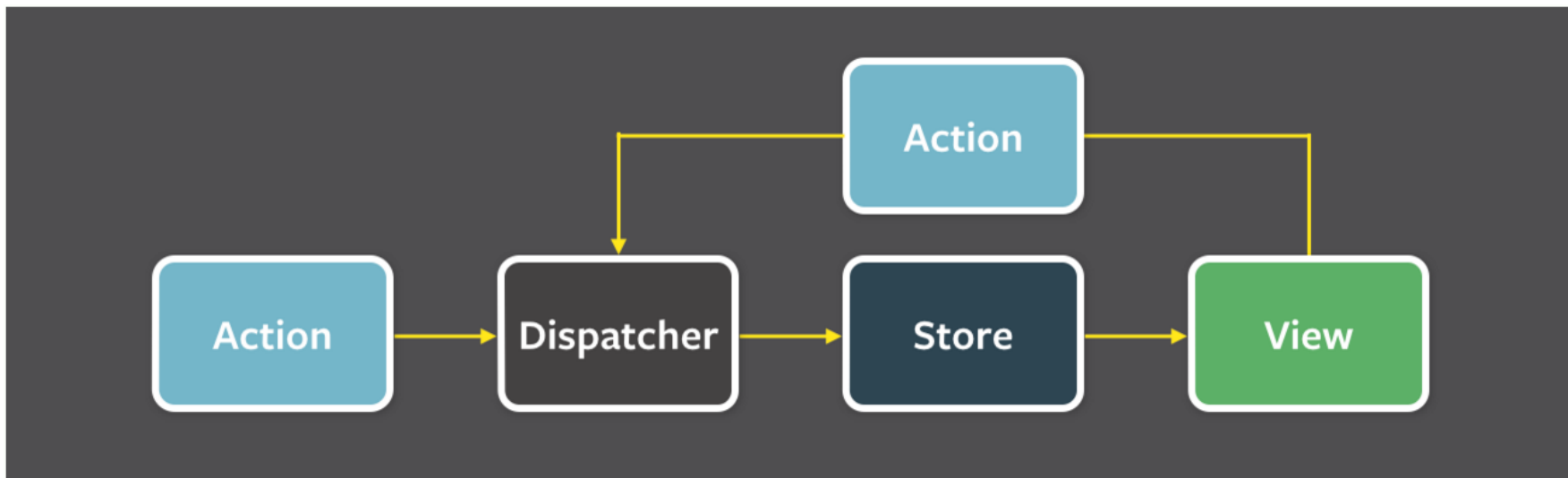
- 🔔 The underlying problem FLUX tries to solve – the way that the data flowed through the application.



FLUX



The views may cause a new action to be propagated through the system in response to user interactions:



FLUX

- 🔔 Generally you have to implement Store by yourself with event notification when changes happen, you have to implement Dispatcher which has all the logic to route Action to Store, and subscribe your component for changes in Store and all that has to follow Flux principles.
- 🔔 But why would you do that if you can choose one of open-sourced libraries which already give you everything you need to optimize your app for FLUX without any complications:

🔔 Facebook Flux

🔔 McFly

🔔 Fluxette

🔔 Fluxible by Yahoo

🔔 Lux

🔔 Fluxxor

🔔 Reflux

🔔 Material Flux

🔔 Freezer

🔔 Alt

🔔 Redux

🔔 Fluxury

🔔 Flummox

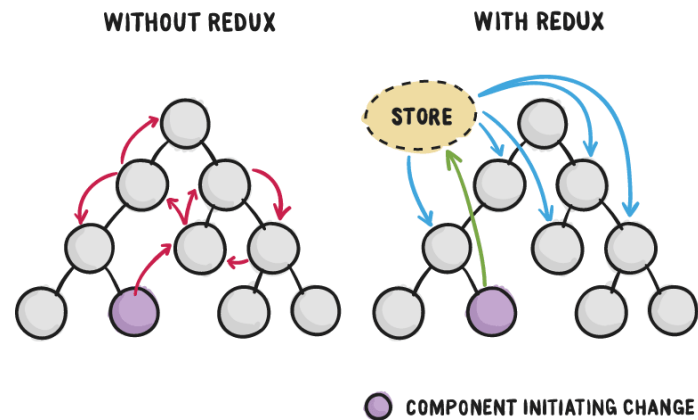
🔔 Redux + Flambeau

🔔 Marty.js

🔔 Nuclear.js

REDUX

- 🔔 Flux is awesome. Single direction data flow rules, you always know the way your data should change.
- 🔔 When you join new project written following FLUX principles you might spend just a little bit of time to figure out what's happening there, because FLUX makes you structure everything very well.
- 🔔 But with pure FLUX there's one problem - code becomes structured, but eventually you have huge bunch of well structured files with significant amount of code and at the end trying to make logic simpler you make code more complex.
- 🔔 Here comes REDUX



IMMUTABILITY

- 🔔 As it turns out, the simplest/fastest way to check whether an object has changed is to check whether it's the same object reference or not.
- 🔔 Instead of doing some sort of deep comparison of properties such as:
 - 🔔 `_.isEqual(object1, object2)`
- 🔔 It'd be a lot faster/simpler if we knew that any time an object changed, we replaced it, instead of editing it in place. Then our check can be simplified to just this:
 - 🔔 `object1 === object2`
- 🔔 That's the basic idea of "immutability". It's not necessarily that you can't change the object
- 🔔 We can implement enforced immutability with tools like Immutable.js. Or even React.addons.update, but you don't need tools for it.
- 🔔 You can just follow the immutability rule: "If you change it, replace it."

IMMUTABILITY PURE JS



Array immutability:

[...arrays], concat, slice, splice, filter, map



Objects:

Object.assign(), {...obj}



Examples:

```
return [  
  {  
    key:value  
  },  
  ...state  
]
```

```
state.filter(todo => todo.completed === false)
```

```
Object.assign({}, todo, { text: action.text })
```

```
state.map(todo => Object.assign({}, todo, {  
  completed: !areAllMarked  
}))
```

PURE FUNCTION

- 🔔 What is pure function? Function may be considered as pure if it holds following statements:
- 🔔 Given the same arguments, it should calculate the the same result. No surprises. No side effects. No API calls. No mutations. Just a calculation.
- 🔔 Examples:

```
var values = { a: 1 };  
  
function impureFunction ( items ) {  
  var b = 1;  
  items.a = items.a * b + 2;  
  return items.a;  
}  
  
var c = impureFunction( values )
```

```
var values = { a: 1 };  
  
function pureFunction ( a ) {  
  var b = 1;  
  a = a * b + 2;  
  return a;  
}  
  
var c = pureFunction( values.a )
```

REDUX

- 🔔 Redux is a predictable state container for JavaScript apps
- 🔔 Redux evolves the ideas of Flux, but avoids its complexity
- 🔔 It doesn't help you render stuff, it doesn't tell you how to do routing, etc. It's just about maintaining application state.
- 🔔 Redux main **ideas** is very simple and clear:
 - **Single source of truth** - The state of your whole application is stored in an object tree within a single store.
 - **State is read-only** - The only way to mutate the state is to emit an action, an object describing what happened.
 - **Changes are made with pure functions** - To specify how the state tree is transformed by actions, you write pure reducers.

REDUX



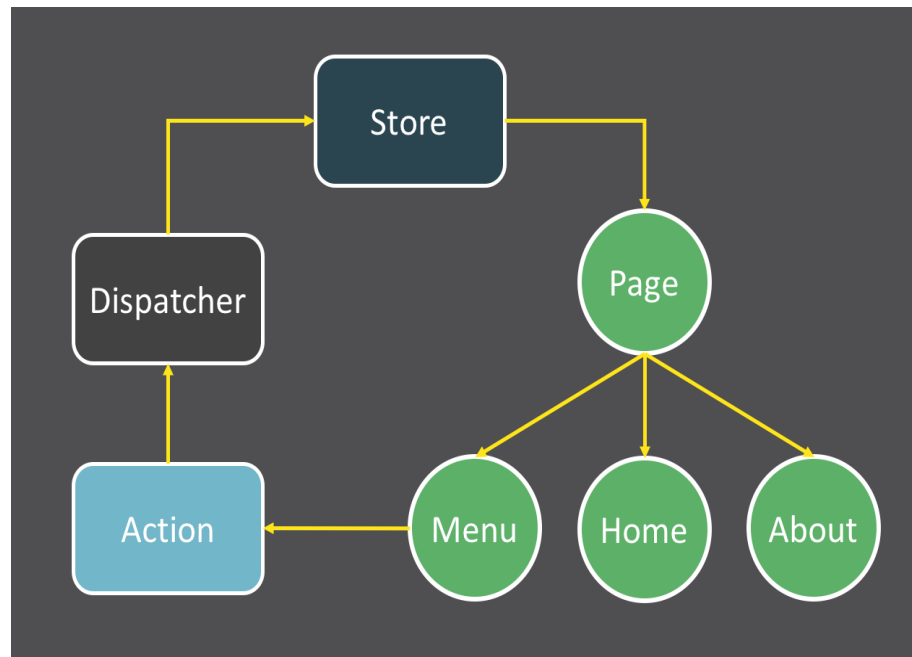
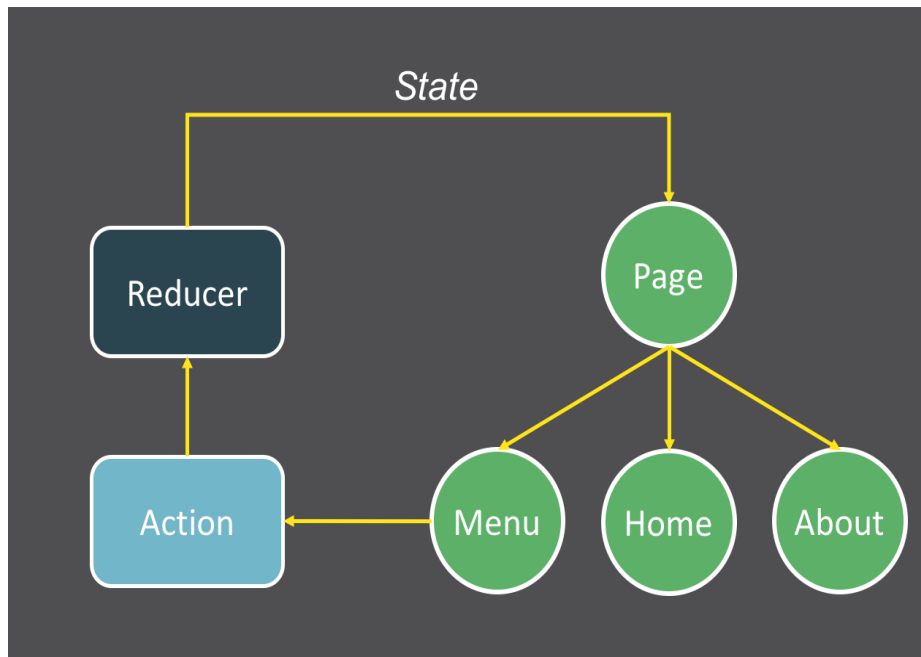
Example

```
function counter(state = 0, action) {  
  switch (action.type) {  
    case 'INCREMENT':  
      return state + 1  
    case 'DECREMENT':  
      return state - 1  
    default:  
      return state  
  }  
}  
  
let store = createStore(counter)  
store.subscribe(() =>  
  console.log(store.getState())  
)  
  
store.dispatch({ type: 'INCREMENT' })//1  
store.dispatch({ type: 'INCREMENT' })//2  
store.dispatch({ type: 'DECREMENT' })//1
```

REDUX

🔔 Flux Flow vs Redux Flow

🔔 Redux – no Dispatcher, Single Store – State, Functional composition where Flux uses callback registration



REDUX VS FLUX



No dispatcher

```
var flightDispatcher = new FluxDispatcher();

flightDispatcher.dispatch({
  actionType: 'city-update',
  selectedCity: 'paris'
});

flightDispatcher.register(function(payload) {
  if (payload.actionType === 'city-update') {
    CityStore.city = payload.selectedCity;
  }
});
```

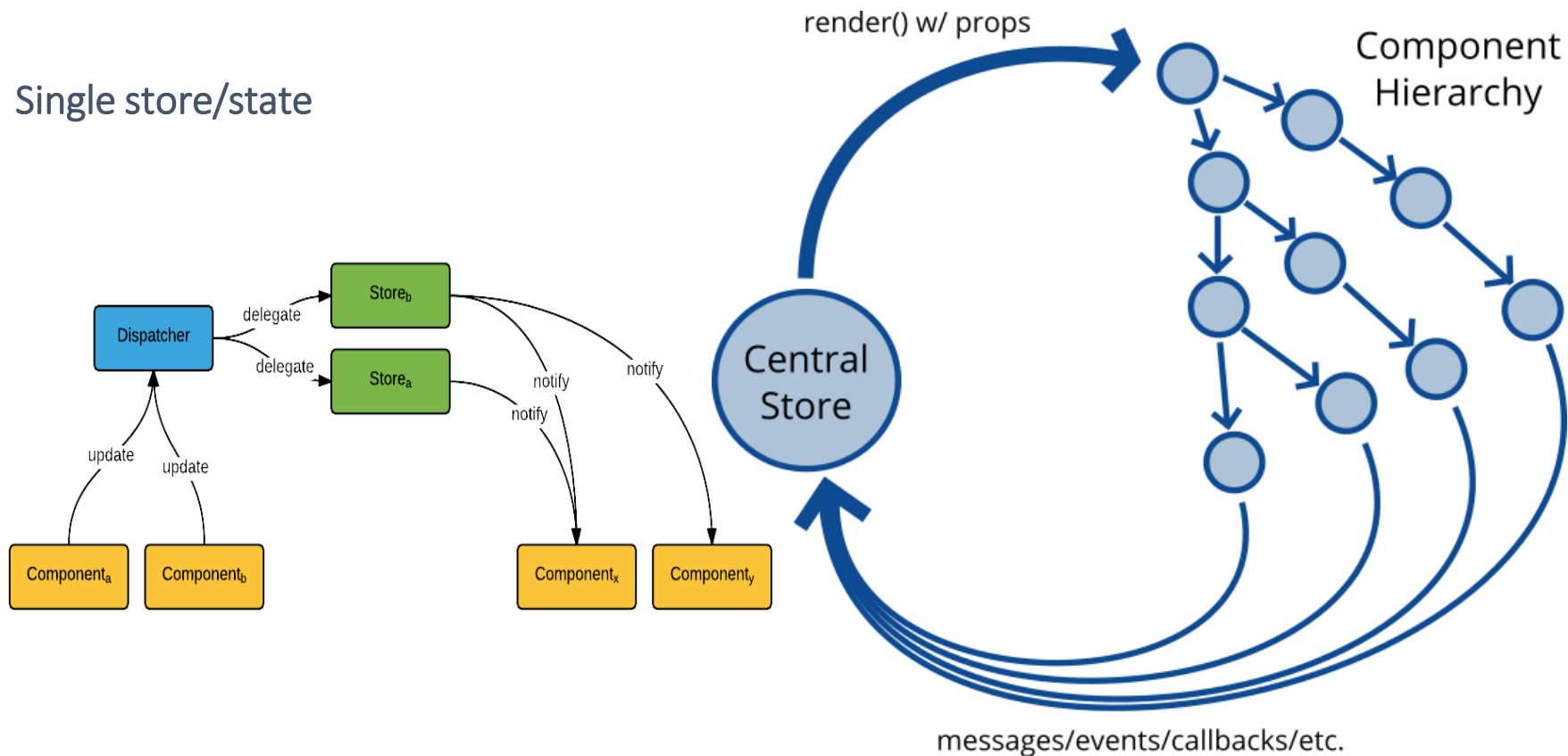
```
export function addTodo(text) {
  return {
    type: ActionTypes.ADD_TODO,
    text: text
  };
}

store.dispatch(addTodo());
```

REDUX VS FLUX



Single store/state



REDUX VS FLUX

- 🔔 Functional composition where Flux uses callback registration
- 🔔 In Flux, stores are flat, but in Redux, reducers can be nested via functional composition, just like React components can be nested.
- 🔔 In Flux, the stores aren't necessarily connected to each other and they have a flat structure. In Redux, the reducers are in a hierarchy
- 🔔 This hierarchy can have as many levels as needed, just like the component hierarchy

REDUX VS FLUX


🔔 Functional composition where Flux uses callback registration Example

```
const todos = (state = [], action) => {  
  switch (action.type) {  
    case 'SOME_ACTION':  
      //DO SOMETHING HERE  
    default:  
      return state;  
  }  
};  
  
const visibilityFilter = (state = 'SHOW_ALL',  
action) => {  
  switch (action.type) {  
    case 'SET_VISIBILITY_FILTER':  
      return action.filter;  
    default:  
      return state;  
  }  
};
```


```
const todoApp = (state = {}, action) => {  
  return {  
    todos: todos(  
      state.todos,  
      action  
    ),  
    visibilityFilter: visibilityFilter(  
      state.visibilityFilter,  
      action  
    )  
  };  
};
```



STORE

 Store in Redux is just a simple object which holds whole state of your app. The store has the following responsibilities:

- Holds application state;
- Allows access to state via **getState()**;
- Allows state to be updated via **dispatch(action)**;
- Registers listeners via **subscribe(listener)**;
- Handles unregistering of listeners via the function returned by `subscribe(listener)`.

 It's important to note that you'll only have a single store in a Redux application. When you want to split your data handling logic, you'll use reducer composition instead of many stores.

```
import { createStore } from 'redux'  
import todoApp from './reducers'  
let store = createStore(todoApp)
```

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()`.
- 🔔 Actions are plain JavaScript objects. Actions must have a `type` property that indicates the type of action being performed. Types should typically be defined as string constants. Once your app is large enough, you may want to move them into a separate module.
- 🔔 Usually it looks like:

```
{  
  type: ADD_TODO,  
  text: 'Build my first Redux app'  
}
```

REDUCERS

- 🔔 Reducer is that combine store and actions together.
- 🔔 Actions describe the fact that something happened, but don't specify how the application's state changes in response. This is the job of a reducer.
- 🔔 The reducer is a pure function that takes the previous state and an action, and returns the next state.

🔔 Why it's

$(\text{previousState}, \text{action}) \Rightarrow$
 newState

- 🔔 It's very important that the **reducer stays pure**. Things you should never do inside a reducer:
 - Mutate its arguments;
 - Perform side effects like API calls and routing transitions;
 - Calling non-pure functions, e.g. `Date.now()` or `Math.random()`.

COMBINE REDUCERS

- 🔔 Few slides ago we were talking about Reducers Composition and why it's necessary.
- 🔔 We have already even created something that looks pretty close to redux utility to combine reducers:

```
const todoApp = (state = {}, action) => {  
  return {  
    todos: todos(  
      state.todos,  
      action  
    ),  
    visibilityFilter: visibilityFilter(  
      state.visibilityFilter,  
      action  
    )  
  };  
};
```



- 🔔 But with built-in helper you can use it this way:

```
import { combineReducers } from 'redux'  
  
export default combineReducers([  
  reducer1,  
  reducer2  
)
```


MIDDLEWARE

- 🔔 Middlewares in Redux, pretty much like in any other technology, let third-party extensions do something useful in between some processes
- 🔔 Redux it provides a third-party extension point between dispatching an action, and the moment it reaches the reducer.
- 🔔 You would might be concerned why do you need it? So here's the deal, let's say you need to log every action and store happened after this action in your app. What would you do?

```
let action = addTodo('Use Redux')  
  
console.log('dispatching', action)  
store.dispatch(action)  
console.log('next state', store.getState())
```

```
function dispatchAndLog(store, action) {  
  console.log('dispatching', action)  
  store.dispatch(action)  
  console.log('next state', store.getState())  
}
```

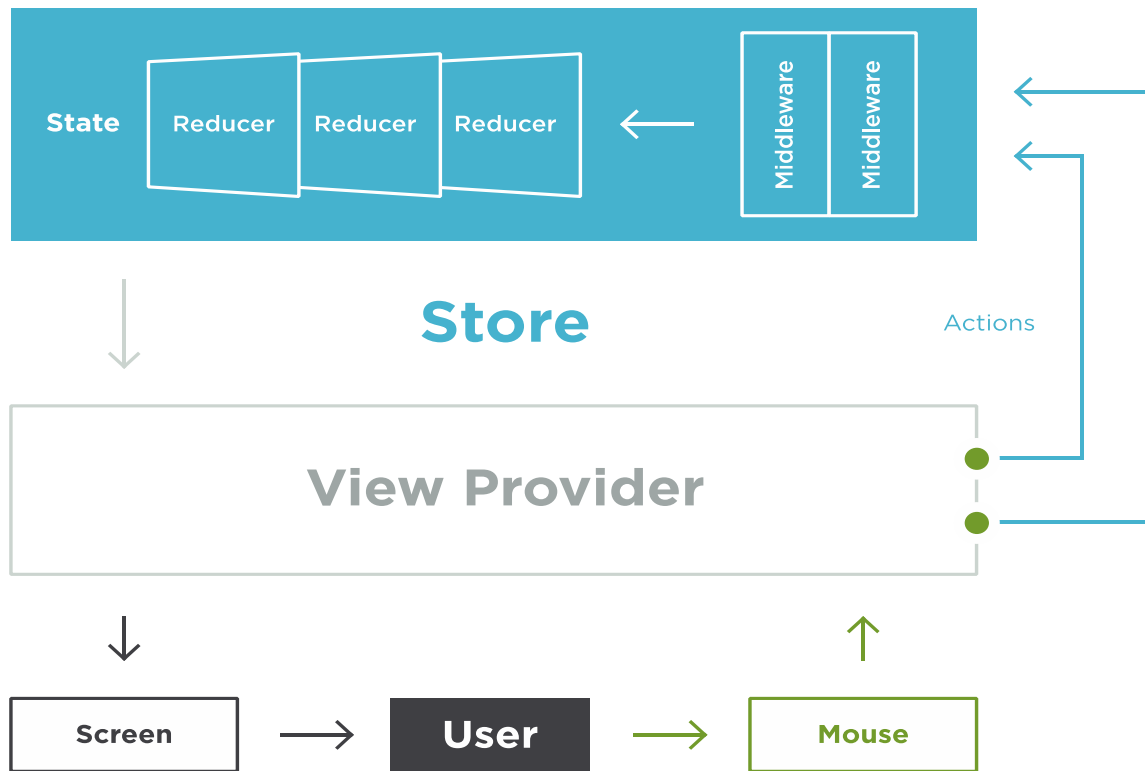
```
let next = store.dispatch  
store.dispatch = function dispatchAndLog(action) {  
  console.log('dispatching', action)  
  let result = next(action)  
  console.log('next state', store.getState())  
  return result  
}
```

MIDDLEWARE

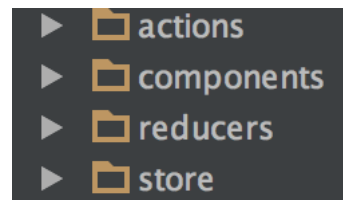
- 🔔 You can have different middlewares in your app: logging, crash reporting, async operations, routing and other. Everywhere you need to put something in between the action and reducer which get this action you can use it.
- 🔔 Lets take a look at the most common example where you would like to use middleware. There's thunkMiddleware which lets you dispatch function instead of actions. It's needed for Async operation mostly. With it you can write Action Creators like that:
- 🔔 1) Store will get startFetching Action
- 🔔 2) And when todos are received, it will get todosReceived result with json data

```
function fetchTodos(url) {  
  return dispatch => {  
    dispatch(startFetchingTodos(url))  
    return fetch(url)  
      .then(response => response.json())  
      .then(json => dispatch(todosReceived(url, json)))  
  }  
}  
store.dispatch(fetchTodos("http://awesomePlaceWithTodos"))
```

TO SUMMARIZE



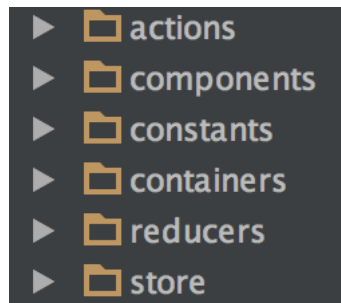
REDUX FOLDER STRUCTURE



- 🔔 Generally redux folders structure is next: Actions, Components, Reducers and Store
- 🔔 Actions here usually contains file/files with all the actions/action creators you going to use in you app
- 🔔 Components obviously contains all the components
- 🔔 Reducer contains reducers hierarchy
- 🔔 And store usually contains one file with with use of redux createStore and some middleware you use

REDUX FOLDER STRUCTURE

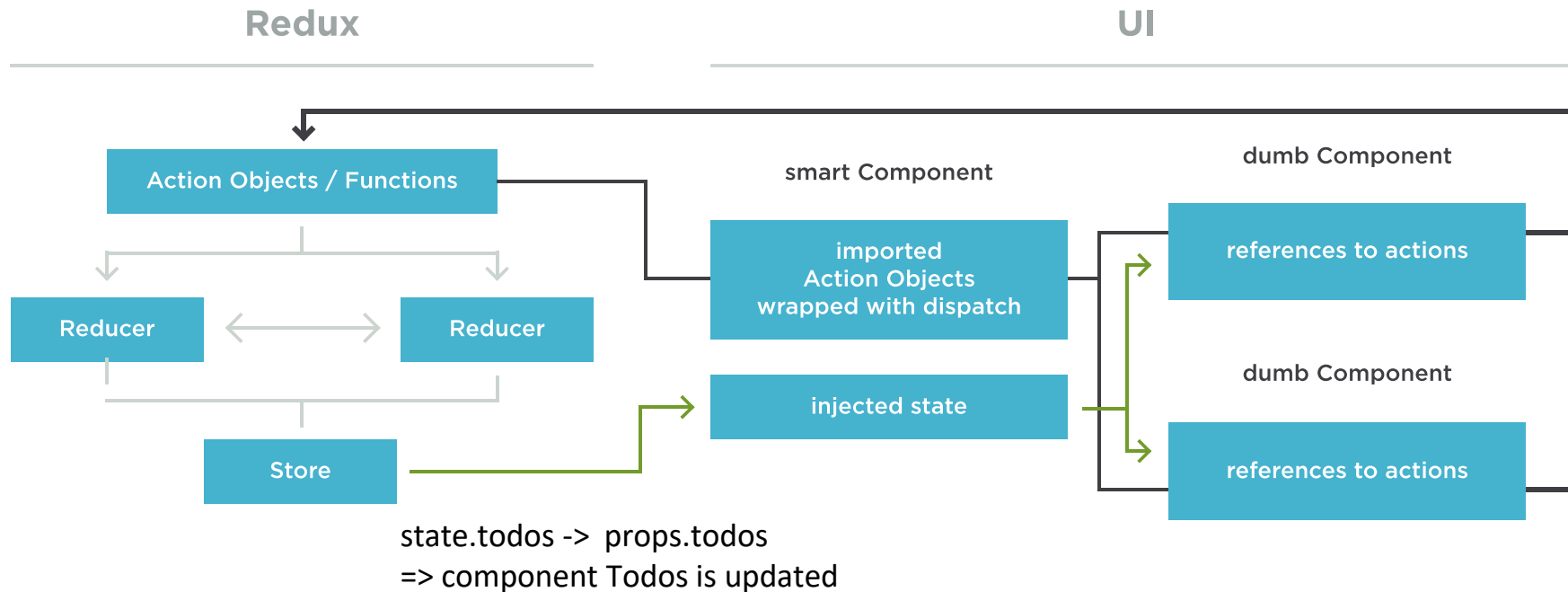
- 🔔 Structure you have seen is absolutely fine, but you can do it better and add more structure and responsibility separation. What is the difference:
- 🔔 Constants folder supposed to have list of constants, usually for action types to make sure reducers and components share the same variable.
- 🔔 Components in this structure are encapsulated React components that are driven solely by props and don't talk to Redux. They should stay the same regardless of your router, data fetching library, etc.
- 🔔 Containers are React components that are aware of Redux, Router, etc. They are more coupled to the app.



REDUX FOLDER STRUCTURE

Sync Flow

Action > Reducer > SmartContainer > DumbComponent



REDUX MEMORY CONSUMPTION

- 🔔 Having immutable state doesn't mean that it is completely recreated every time.
- 🔔 On the example parts of the state 1 and 2 are not changed, and it doesn't need additional memory
- 🔔 Only 3 was changed to 4, so this part will need new memory
- 🔔 However, after all components will be updated, part 3 will be garbage collected, and this memory will free up

