REACT JS

Redux Prerequisites

Sreekanth M. E.

Freelance Trainer & Consultant

http://www.SreekanthME.com

IMMUTABILITY

In programming, a value which is allowed to change over time is called mutable.

An immutable value is the exact opposite – after it has been created, it can never change.

In JS, primitive datatypes (like Strings and Numbers) are immutable by design.

But Objects (including arrays and functions) are mutable.

ARRAY FUNCTIONS

Mutating Functions: push, pop, shift, unshift, sort, reverse, splice and delete

Non-Mutating Functions: concat, slice, map, reduce, and filter

OBJECT CHANGE: NON-MUTATING OPTIONS

- Object.assign() method
- Spread operator

OBJECT.ASSIGN()

The Object.assign() method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It returns the target object.

Object.assign(target, source[s])

target: Target object.

Source[s]: Source object(s).

Return value: New object.

SPREAD OPERATOR

The spread operator (…) is used for array/object construction.

Spread expands an array into its elements and an object into its properties.

PURE FUNCTIONS

A pure function is a function which produces no side effects.

FUNCTION WITH NO SIDE EFFECTS MEANS:

- Does not mutate its arguments (array/object).
- Does not depend or change global variables.
- No input/output.
- No DB or Network (API) call.
- No call to Date.now() or Math.random().

PURE FUNCTION CHARACTERISTICS:

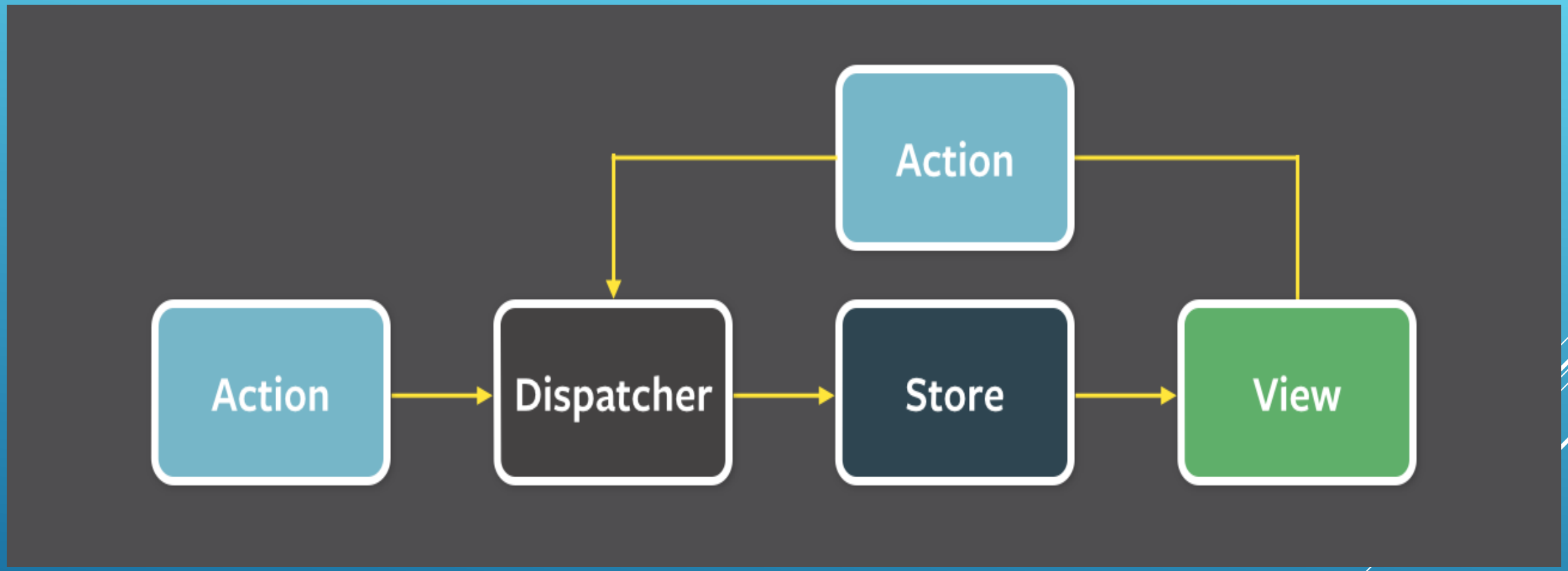
Given the same parameters, will always return the same output.

Its execution doesn’t depend on the state of the system.

A dead giveaway that a function is impure is if it makes sense to call it without using its return value.

Pure functions always return a value.

FLUX PATTERN



Action Creators: Helper functions that pass a payload data to the Dispatcher on user interaction or server response.

Dispatcher: Receives the Action and broadcasts its payload to all registered stores.

Stores: Act as containers for application state & logic. A store registers itself with the dispatcher and provides it with a callback. This callback receives the action as a parameter. The store will then emit an event for the React components that care.

Views: React Components get the state from the stores and update the UI. Some of them (Controller Views) also pass it down to the child components.

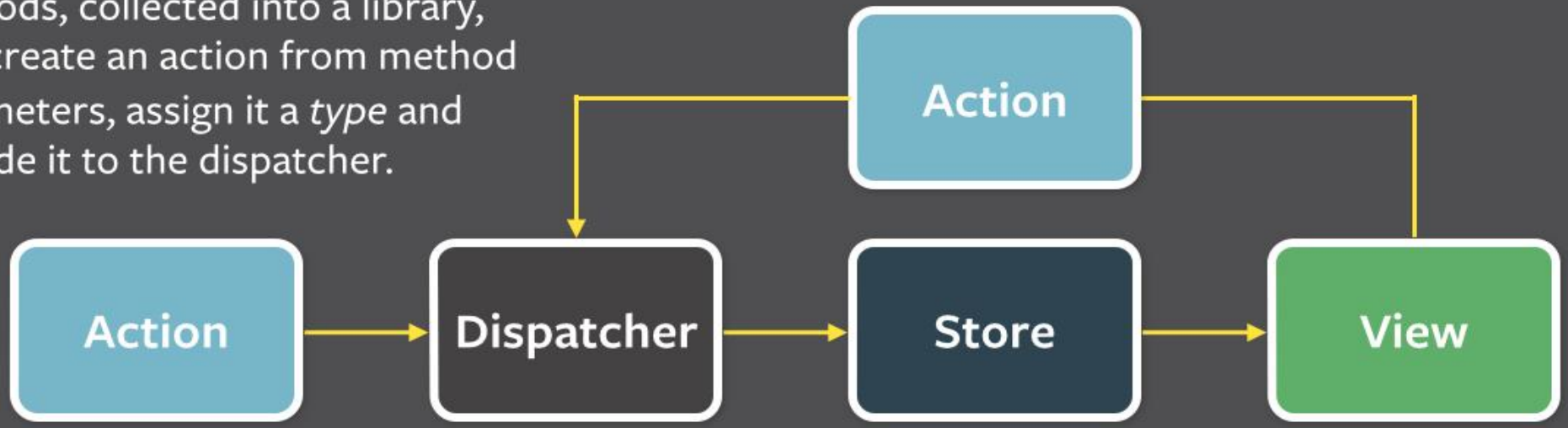
SAMPLE ACTIONS

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

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

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

Action creators are helper methods, collected into a library, that create an action from method parameters, assign it a *type* and provide it to the dispatcher.



Every action is sent to all stores via the *callbacks* the stores register with the dispatcher.

After stores update themselves in response to an action, they emit a *change* event.

Special views called *controller-views*, listen for *change* events, retrieve the new data from the stores and provide the new data to the entire tree of their child views.

REACT-FLUX FLOW

React (View): Hey Action, someone clicked this “Save Course” button.

Action: Thanks React! I triggered an action creator with the dispatcher, so the dispatcher should take care of notifying all the stores that care.

Dispatcher: Let me see who cares about a course being saved. Ah! Looks like a Store has registered a callback with me, so I’ll let her know.

Store: Hi dispatcher! Thanks for the update! I’ll update my data with the payload you sent. Then I’ll emit an event for the React components that care.

React (View): Ooo! Shiny new data from the store! I’ll update the UI to reflect this!

MVC VS FLUX

- Many Models vs Few Stores
- Bi-directional vs unidirectional

END OF CHAPTER

APPENDIX