React and Redux

Hello!

I am Mateusz Choma

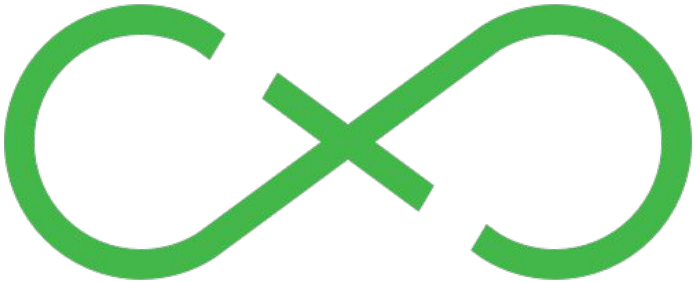
I am a scientific mind, passionate of technology, an engineer "squared" - a graduate of two universities in Lublin :)
As well, I am a JS developer, entrepreneur and owner of small software house - Amazing Design.

1.
Flux

Flux
About

“Flux is the application architecture that Facebook uses for building client-side web applications. It complements React's composable view components by utilizing a unidirectional data flow. It's more of a pattern rather than a formal framework.”

- *quote from Flux docs*



Flux
Key concepts

There are four key concepts in Flux:

- **Dispatcher**
- **Store**
- **Action**
- **View** - retrieve data from the stores and pass this data down to their children

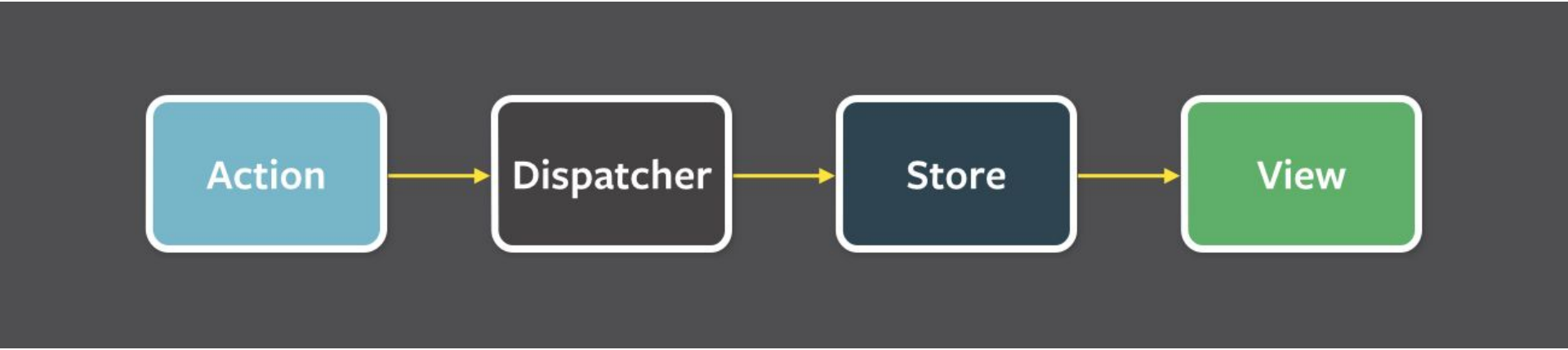
Flux

Unidirectional data flow

When a user interacts with a React view, the view propagates an action through a central dispatcher, to the various stores that hold the application's data and business logic, which updates all of the views that are affected.

Flux

Unidirectional data flow



Flux
Dispatcher

The **dispatcher** receives actions and dispatches them to stores that have registered with the dispatcher. Every store will receive every action. There should be only one singleton dispatcher in each application.

Example:

1. User types in title for a todo and hits enter.
2. The view captures this event (form submit) and **dispatches** an "add-todo" **action** containing the title of the todo.
3. Every **store** will then receive this action.

Flux
Store

A **store** is what holds the data of an application.

Stores works with the application's **dispatcher** so that they can receive **actions**.

The data in a store is mutated only when it is a response to an **action**.

There should not be any public way to set the value in a store directly only through actions, but store should be readable freerly.

Every time a store's data changes it must emit a "change" event.

Flux
Store

Example:

1. **Store** receives an "add-todo" **action**.
2. **Store** decides it is relevant and adds the todo to the list of things that need to be done today.
3. **Store** updates its data and then **emits a "change" event**.

Flux
Actions

Actions define the internal API of your application. They capture the ways in which anything might interact with your application.

They are simple objects that have a "type" property, and some data in others properties.

Actions should be semantic and descriptive of the action taking place. They should not describe implementation details of that action. For example: use "delete-user" action, rather than breaking it up into "delete-user-id", "clear-user-data", "refresh-credentials" (or however the process works).

Flux
Actions

Examples:

1. When a user clicks "delete" on a completed todo a single "delete-todo" action is dispatched:

```
{
    type: 'delete-todo',
    todoID: '1234',
}
```

Flux
Views

Data from **stores** is displayed in **views**.

Views can use whatever framework you want (of course we will be using React :)).

When a **view** uses data from a **store** it must also subscribe to **change events** from that **store**. Then when the **store** emits a **change event** the **view** can get the new data and re-render.

Actions are typically **dispatched** from **views** when the user interacts with parts of the application's interface (e.g. button click).

Flux
Views

Example:

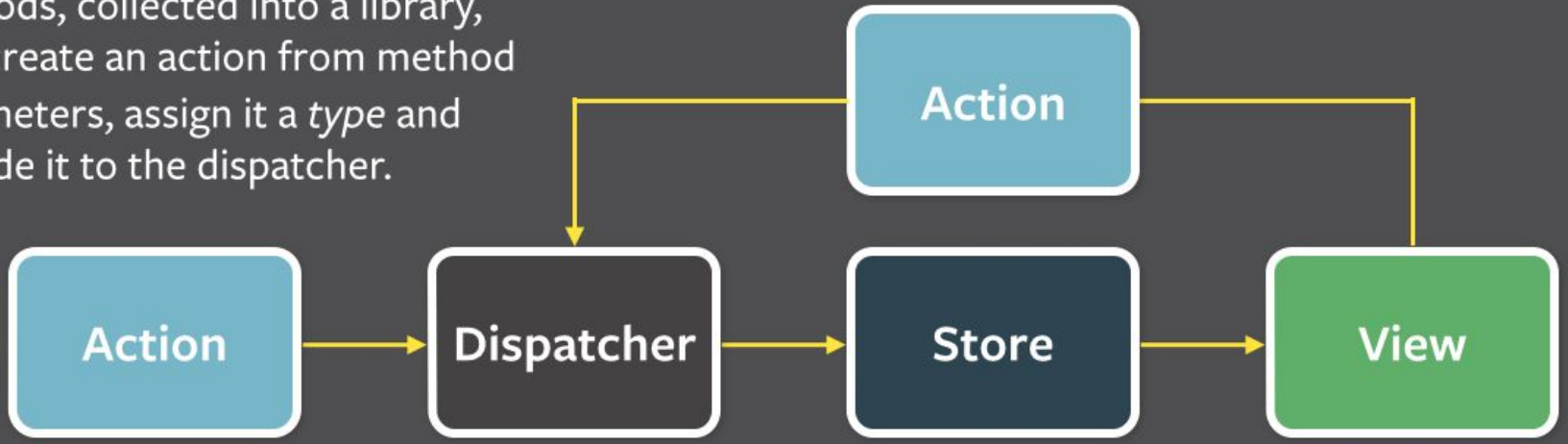
1. The main **view** subscribes to the **TodoStore**.
2. It accesses a list of the Todos and renders them in a readable format for the user to
3. interact with.
4. When a user types in the title of a new Todo and hits enter the view tells the **dispatcher** to **dispatch** an **action**.
5. All **stores** receive the **dispatched action**.
6. The **TodoStore** handles the action and adds another Todo to **its internal data** structure, then **emits a "change" event**.
7. The main **view** is listening for the "**change" event**. It gets the event, gets new data from the **TodoStore**, and then re-renders the list of Todos in the user interface.

Flux
Summary

We can piece the parts of Flux above into a diagram describing how data flows through the system.

1. **Views** sends **actions** to the **dispatcher**.
2. The **dispatcher** sends **actions** to every **store**.
3. **Stores** send data to subscribed **views**.

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.

2.
Redux

Redux
About

“**Redux** is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger.”

- *quote from Redux docs*

Redux
Redux vs Flux

Redux is an implementation of flux unidirectional data flow. There is no big difference, but there are some small:

- only single store
- reducer pure functions - that decides what to do when action is dispatched
- no separate dispatcher - store can dispatch actions

Redux

Redux data flow

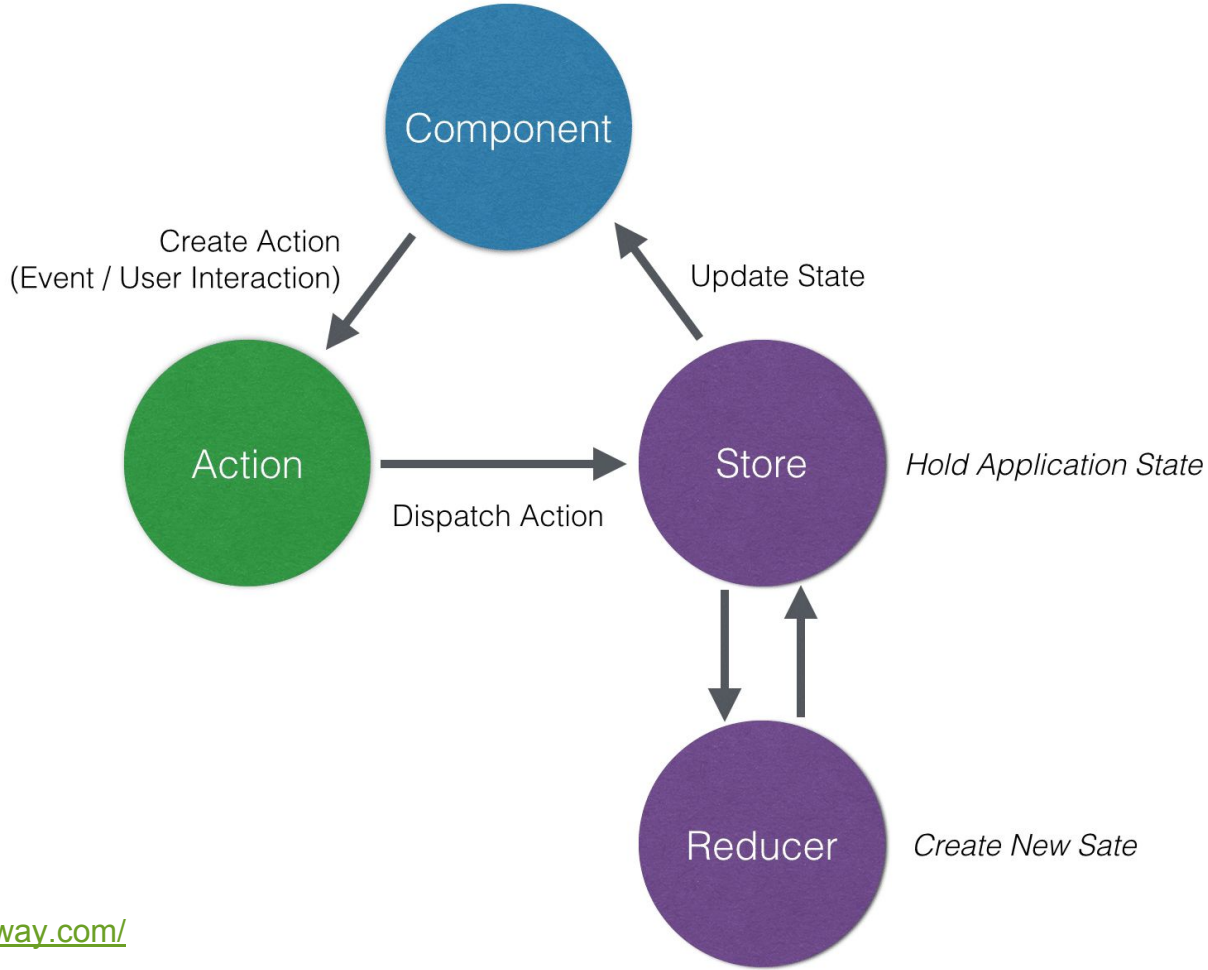


Image source - https://codingthesmartway.com/

Redux
Three principles

When using Redux we must follow, three simple principles:

- Single source of truth
- State is read-only
- Changes are made with pure functions

More -> https://redux.js.org/docs/introduction/ThreePrinciples.html

Redux
Single source of truth

When using Redux we must follow, three simple principles:

- **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 change the state is to emit an action, an object describing what happened.
- **Changes are made with pure functions**

More -> https://redux.js.org/docs/introduction/ThreePrinciples.html

Redux
Pure function

A pure function is a function which:

- Given the same input, will always return the same output.
- Produces no side effects.

More - >
https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-pure-function-d1c076bec976

Redux

Pure function example

```
const add = (a, b) => a + b
```

Redux

Impure function example

```
const returnDate = () => Date.now()
```

Redux

Impure function example

```
const addToArray = (arr, x) => arr.push(x)
```

Redux
Core concepts

Redux is very simple if we can understand core concepts and think in Redux-way and restricting three principles!

Redux
Core concepts

First thing to do is to try to imagine our app as a pure object. For example:

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

Redux
Core concepts

Go to
https://redux.js.org/docs/introduction/CoreConcepts.html
for further examples to understand core concepts.

Redux
Actions and action types

Action types are strings that denotes certain action.

```
const ADD_TODO = 'ADD_TODO'
```

It is a good practice to put them into consts and use const in **actions**:

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

Actions and JS object with type and optional data inside.

Redux
Actions creators

Action creators are helper functions that we can use to create actions:

```
const addTodo = text => ( {
    type: ADD_TODO,
    text
}
```

Redux
Reducers

Reducers are pure functions that responds to dispatched action and returns new state.

```
const todosReducer =(state = [], action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat(
          [{ text: action.text, completed: false }]
       )
    default:
      return state
  }
}
```

Redux
Reducers

1. **We don't mutate the state.** We create a copy with Object.assign, or spread operator.

2. **We return the previous in the default case.** It's important to return the previous state for any unknown action.

Redux
Reducers

Redux provides a utility function called **combineReducers()** that calls all reducer function at once and returns a new state, so we can write separate reducers for every part of our app logic, and Redux will treat them as one reducer.

```
import { combineReducers } from 'redux'

const todoApp = combineReducers({
    visibilityFilter,
    todos
})
```

Redux
Store

The **store** is the object that brings all together. The store has the following responsibilities:

1. Holds application state;
2. Allows access to state via getState() methods
3. Allows state to be updated via dispatch(action)
4. Registers listeners via subscribe(listener)
5. Handles unregistering of listeners via the function returned by subscribe(listener).

Redux
Store

We can create store by redux function createStore(). That function accepts one argument - reducer. That can be single reducer or reducer combined from multiple reducers by combineReducers() function.

```
import { createStore } from 'redux'

let store = createStore(todoApp)
```

3.
react-redux

react-redux
About

react-redux is a library that provides components and functions to connect Redux to React components.

react-redux
Provider component

Provider component is a component that should wrap whole application. It provides store to the components that are its children.

In most cases it will be added directly in render method in index.js or in App component (main component of the app).

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

react-redux
connect

connect is a function that gets two methods as arguments and returns a function that gets a component as an argument and returns the component wrapped in Connect component that can provide action dispatchers and parts of store as our component props, and refresh them on state change.

react-redux
connect

```
import React from ‘react’
import { connect } from ‘react-redux'

class TodoList extends React.Component{
    ...
}

export default connect(
    mapStateToProps,
    mapDispatchToProps
)(TodoList)
```

react-redux
mapStateToProps

mapStateToProps is a function that will be called with state as an argument and should return object, with props names that we want to connect to parts of state as properties, and parts of state as they values.

```
const mapStateToProps = state => ({
    todos: state.todos
})
```

react-redux
mapDispatchToProps

mapDispatchToProps is a function that will be called with store dispatch method as an argument and should return object, with props names that we want to connect to dispatch functions as properties, and dispatch functions as values.

```
const mapDispatchToProps = dispatch => ({
    onTodoClick: id => dispatch(toggleTodo(id))
})
```

4.
Redux Dev Tools

Redux Dev Tools
Chrome extension

Redux Dev Tools is a Chrome extension that we can connect to Redux in our application and have real-time insight on store, and dispatched actions.

We can use it as a time-machine in our application!

https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en

Redux Dev Tools
Chrome extension

We can connect **Redux Dev Tools** with our app simply adding global variable provided by this extension to **createStore** function:

```
const store = createStore(
    reducer,
    window.__REDUX_DEVTOOLS_EXTENSION__ &&
    window.__REDUX_DEVTOOLS_EXTENSION__()
)
```

4.
Creating simple To Do App with Redux

5.
redux-thunk

redux-thunk
About

Redux Thunk teaches Redux to recognize special kinds of actions that are functions not an objects with type!

When a function not an object is dispatched, that function will get executed by the Redux Thunk middleware.

This function doesn't need to be pure! It is thus allowed to have side effects, including executing asynchronous API calls! The function can also dispatch actions!

redux-thunk
About

The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met!

If Redux Thunk middleware is enabled, any time you attempt to dispatch a function instead of an action object, the middleware will call that function with dispatch method itself as the first argument.

And then since we “taught” Redux to recognize such “special” action creators (we call them thunk action creators), we can now use them in any place where we would use regular action creators.

redux-thunk

Enabling middleware with Redux Dev Tools

```
import { createStore, applyMiddleware, compose } from
'redux';

const composeEnhancers =
window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose

const store = createStore(
    reducer,
    composeEnhancers(
        applyMiddleware(...middleware)
    )
)
```

6.
Sum up

https://medium.com/@gyeon/redux-vs-flux-a31a02faccf3

7.
Creating simple To Do App with Redux and async database connections