From Flux to Redux

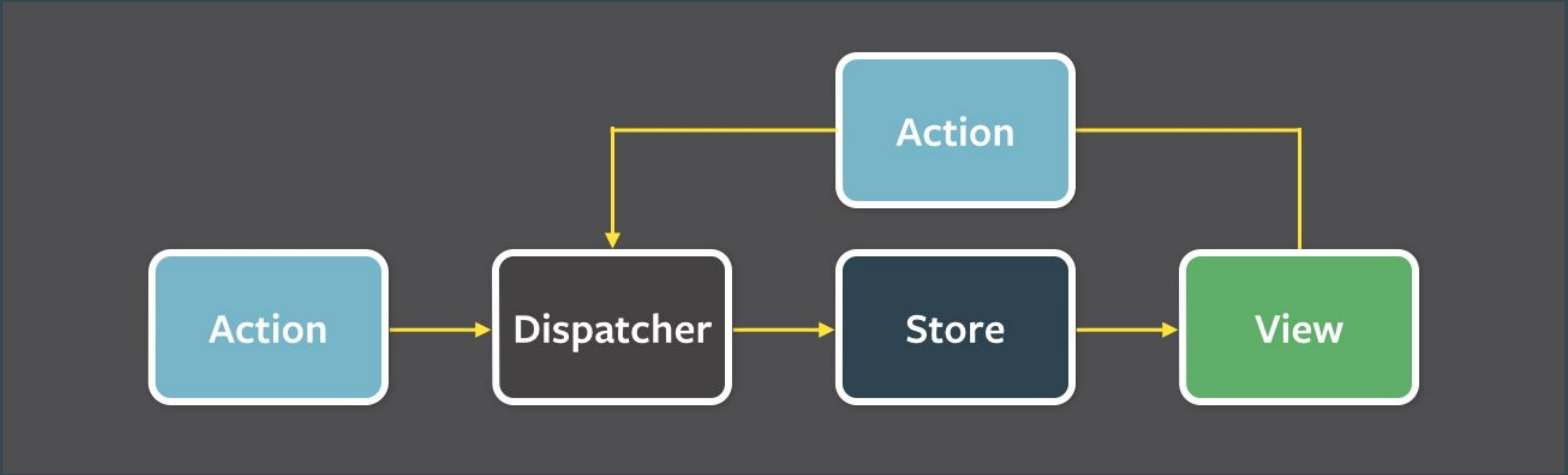
•••

How to get a great DX through a predictable state container

Handling application state is a complex task

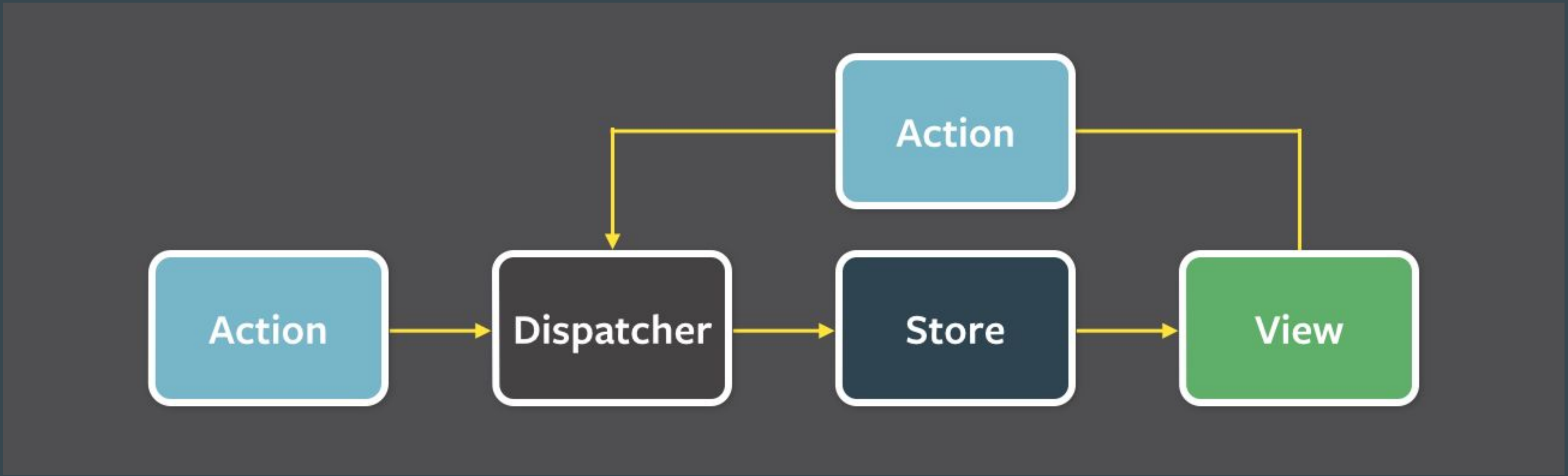
We want to use something that is predictable and testable

What is Flux?



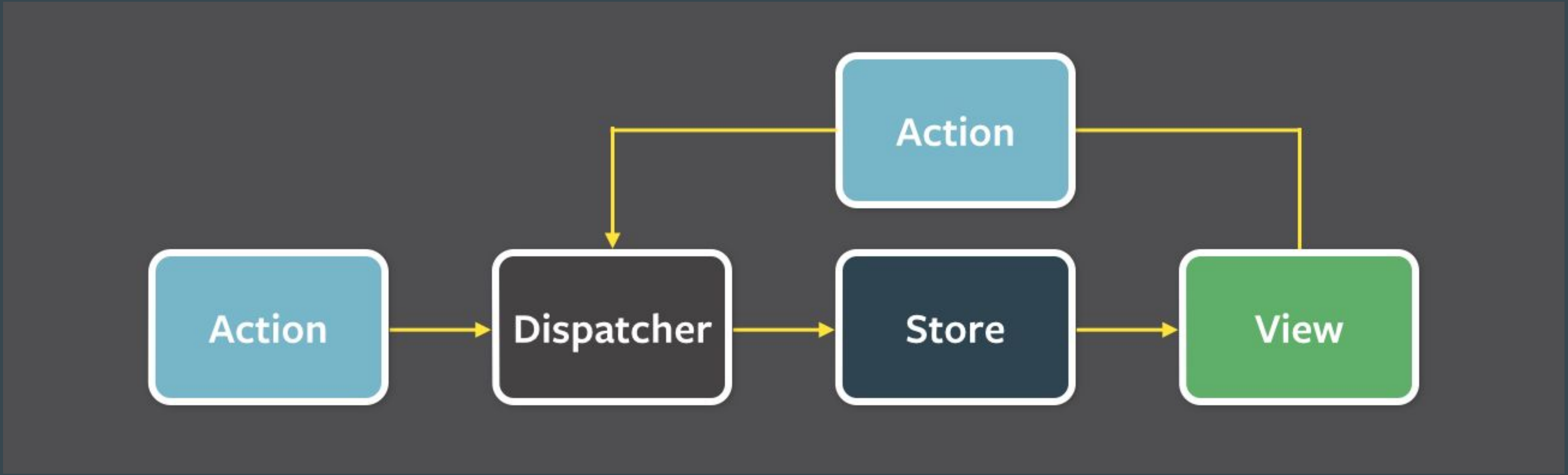
Unidirectional data flow

What is Flux?



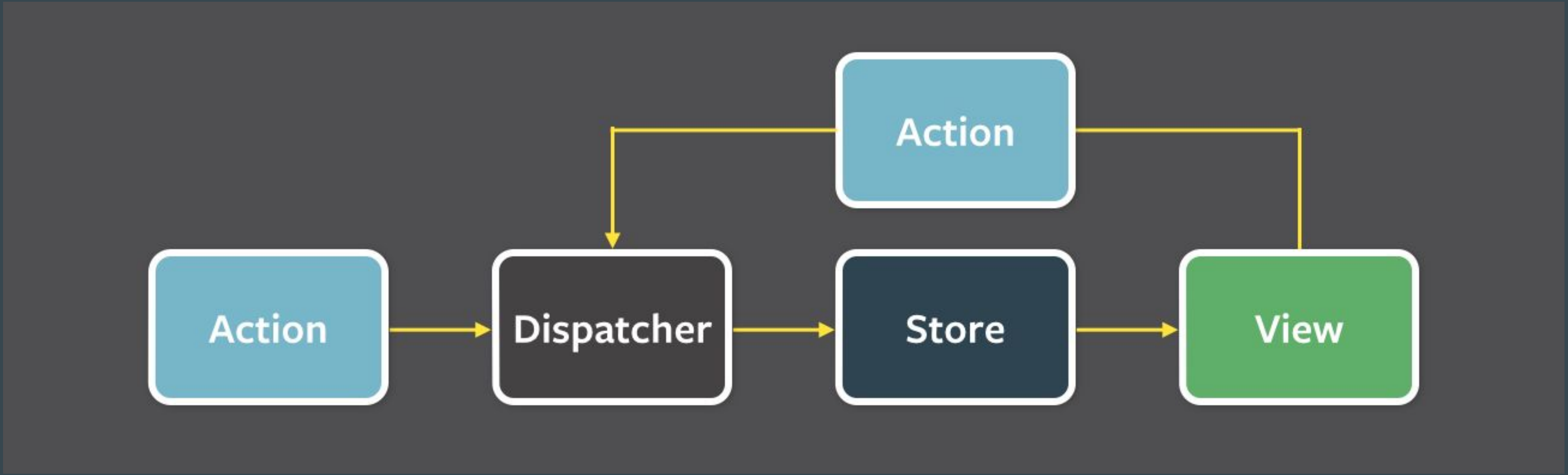
An Action describes an event and its payload

What is Flux?



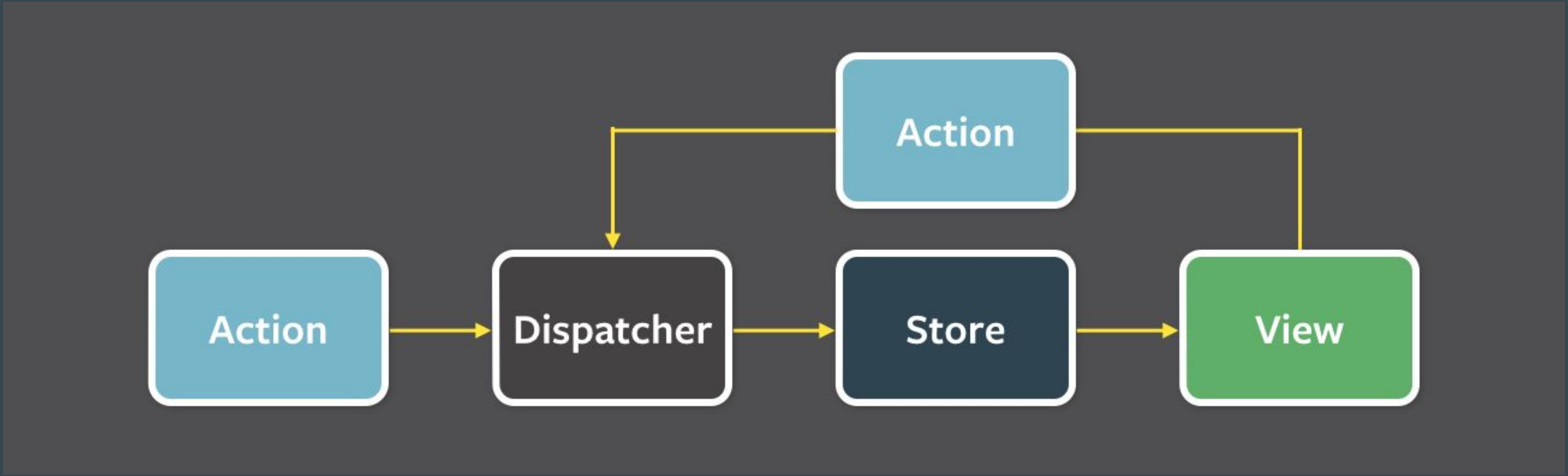
Many Singleton Stores contain application state and logic

What is Flux?



A Single Dispatcher routes actions to Stores

What is Flux?



Views dispatch Actions and Subscribe to Stores changes

Still too complex

•••

we want something simpler

Let’s reduce complexity

using reducers

Reducers

- Describe state mutations using pure functions

(state, action) => state

```
<array>.reduce( (previousValue, current) => nextValue, initialValue);
```

```
const sum = [1, 2, 3, 4, 5].reduce( (previous, current) => previous + current, 0);
```

- Compose reducers to build complex application state

```
const rootReducer = combineReducers({slice1: reducer1, … sliceN: reducerN});
```

Let’s build a stateless store

•••

using reducers

Store

- First commandment: Thou shalt have no other store before me

```
const store = createStore(rootReducer, initialState);
```

The Redux Store is the single source of truth for the application state.

- Second commandment: Thou shalt not mutate the state (that is a graven image)

```
store.dispatch({
    type: ACTION_TYPE,
    payload: payload
});
```

The application state is read-only, the only way to mutate it is to dispatch an action.

Store

- Third commandment: Thou shalt not use side-effects in vain

```
const sum = [1, 2, 3, 4, 5].reduce( (previous, current) => previous + current, 0);
```

Reducers, that describe state mutations, must be pure functions.

The three commandments enable predictability of the state in any moment of time:

- you can always look at the whole application state in a single place
- No one can change the current state, each mutation creates a new state (history is always preserved)
- given an initial state and an action, the next state is always predictable

Demo time

We want a predictable UI too

•••

ReactJS to the rescue

ReactJS

- Components and JSX

```
const MyComponent = React.createClass({
   render: () => <div>Hello {this.props.name}</div>
});
```

- Composition

```
const MyContainer = React.createClass({
   render() {
       return <div><MyComponent name=”Mauro”/></div>;
   }
});
ReactDOM.render(MyContainer, document.getElementById(‘container’));
```

ReactJS

- Stateless Components (ReactJS 0.14)

```
const MyComponent = (props) => <div>Hello {this.props.name}</div>;
```

- UI = f(props)

ReactJS enables a declarative approach to build views that are a pure function of a given set of properties.

Let’s connect UI to state

using selectors

Connect

- Dumb Components

```
const Dumb = (props) => <div>Hello {this.props.name}</div>;
```

Dumb components are not connected to the state, they only receive properties from their parents.

- Smart Components

```
const Smart = connect(state => state.person.name)(Dumb);
```

Smart components are connected to part of the state through a selector.

State properties are mapped to Components props.

UI = f(state)

Provider

A Provider component provides state access to a hierarchy of Components.

```
<Provider store={store}>
   <App/>
</Provider>
```

This is enabled by the React context (undocumented until 0.14) feature.

Dispatching actions

```
const Dumb = (props) => <div onClick={this.props.action1}>Hello</div>;
const Smart = connect(selector,
     (dispatch) => bindActionCreators({action1, ...actionN}), dispatch) (Dumb);
```

Demo time

When the going gets tough …

the tough get going

ImmutableJS

- Big applications need a big state tree
- Creating a new state for each mutation can slow down such a big application
- ImmutableJS implements a set of immutable but efficient data structures
- List, Stack, Map, OrderedMap, Set, OrderedSet, Record and Seq

```
const map1 = Immutable.Map({a:1, b:2, c:3});
const map2 = map1.set('b', 50);
map1.get('b'); // 2
map2.get('b'); // 50
```

Reselect

- The application state should be as small as possible
- Computed properties need to be calculated for each state mutation in selectors
- Reselect enables recalculating properties only if needed (only if the property they depend on are changed)
- This is enabled by memoized selectors

```
const totalSelector = (state) => state.items ? state.items.size : 0;


const computedSelector = createSelector([totalSelector], (total) => {
    return total * 2;
});
```

Asynchronous actions

- Enabled by the thunk middleware
- Middlewares can be configured to enable additional functionality in a transparent way (e.g. logging)

```
const asyncAction = (payload) => {
    return (dispatch) => {
        axios.get(“/service”).then((response) => {
            dispatch(responseReceived(response.body));
        });
    };
}
```

Demo time

Bells and whistles

everybody is here for them

Devtools

- Store enhancer
- Time travelling debugger
- Customizable monitor

Hot reload

- WOW
- Code is reloaded but state is mantained

Undo - redo

- Easily implement history functionality
- Easily persist / reload state

That’s all folks!

•••

https://github.com/mbarto/ReduxApericoder