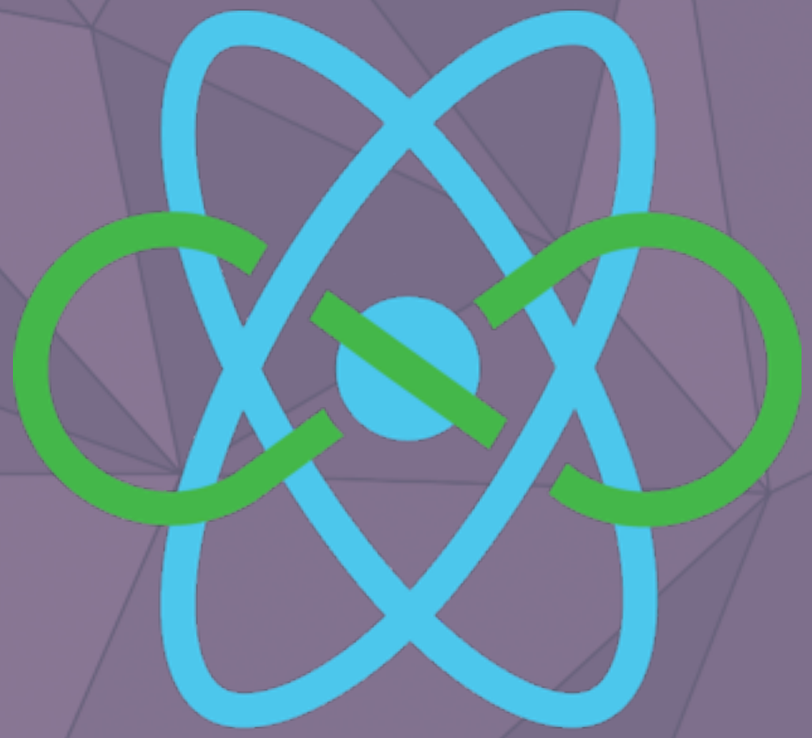


React and Flux

Building Applications with a Unidirectional Data Flow

React and Flux

Building Applications with a Unidirectional Data Flow

Bill Fisher
@fisherwebdev

React and Flux

https://speakerdeck.com/fisherwebdev/flux-applicative

http://facebook.github.io/flux/

http://facebook.github.io/react/

Bill Fisher
@fisherwebdev

#reactjs #fluxjs

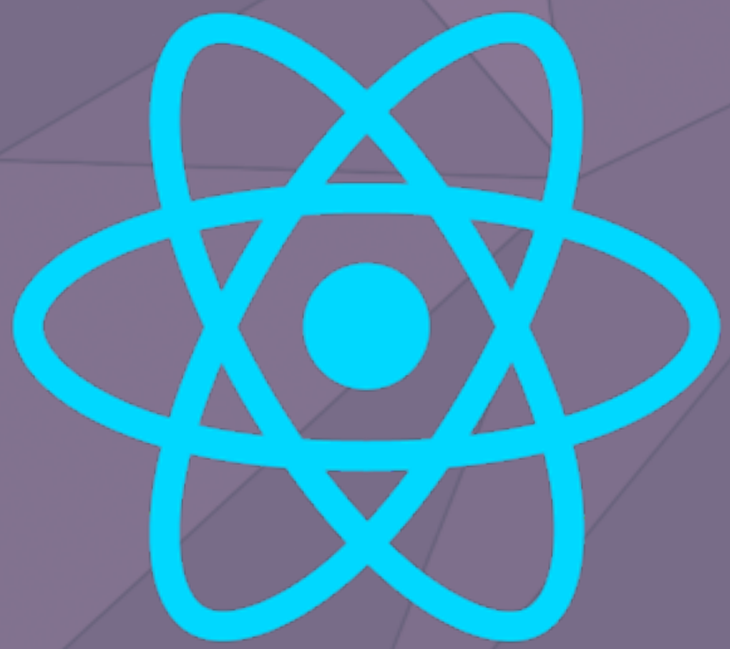
https://speakerdeck.com/fisherwebdev/flux-applicative

http://facebook.github.io/flux/

http://facebook.github.io/react/

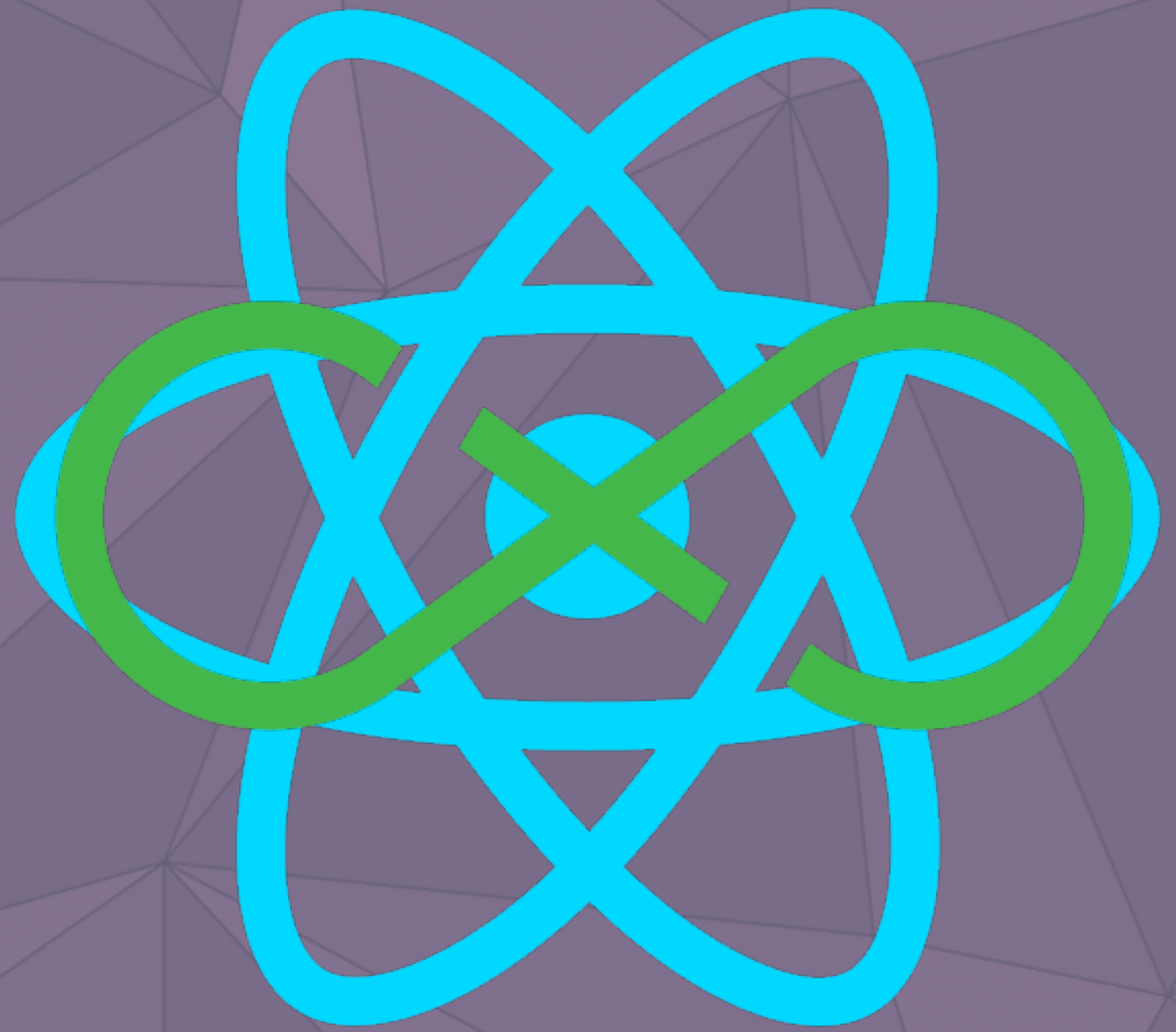
Bill Fisher
@fisherwebdev

https://speakerdeck.com/fisherwebdev/flux-applicative

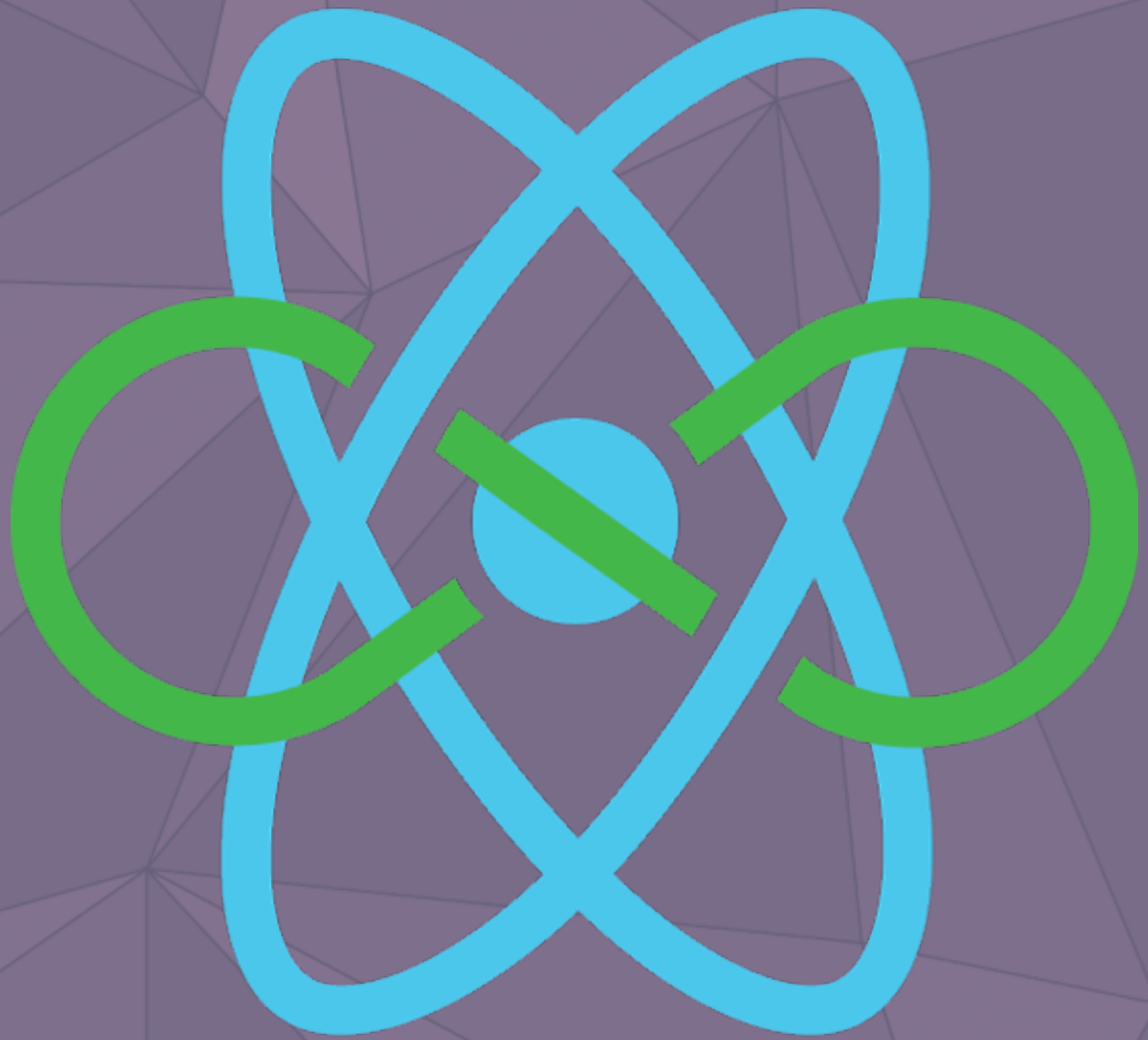


React and Flux

https://speakerdeck.com/fisherwebdev/flux-applicative



https://speakerdeck.com/fisherwebdev/flux-applicative



https://speakerdeck.com/fisherwebdev/flux-applicative

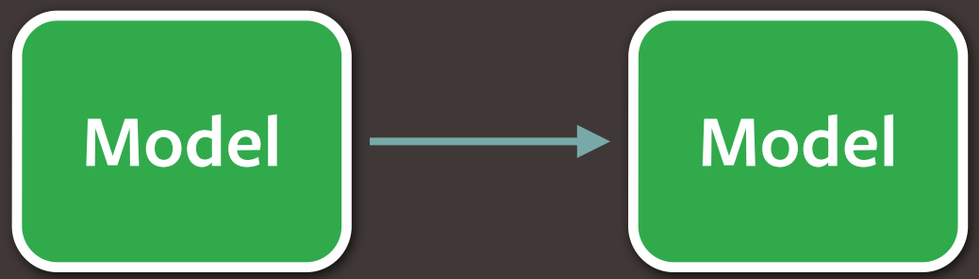
https://speakerdeck.com/fisherwebdev/flux-applicative

Model

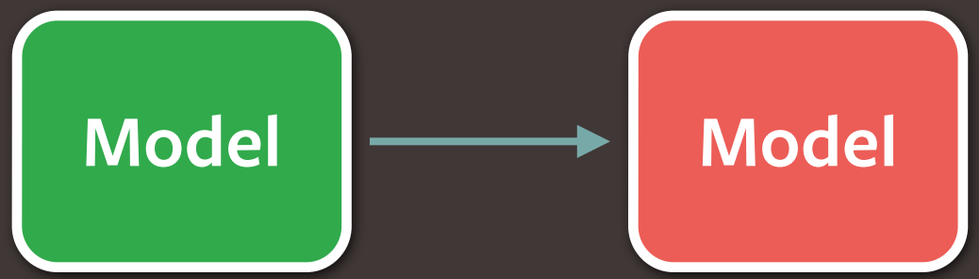
Model

Model

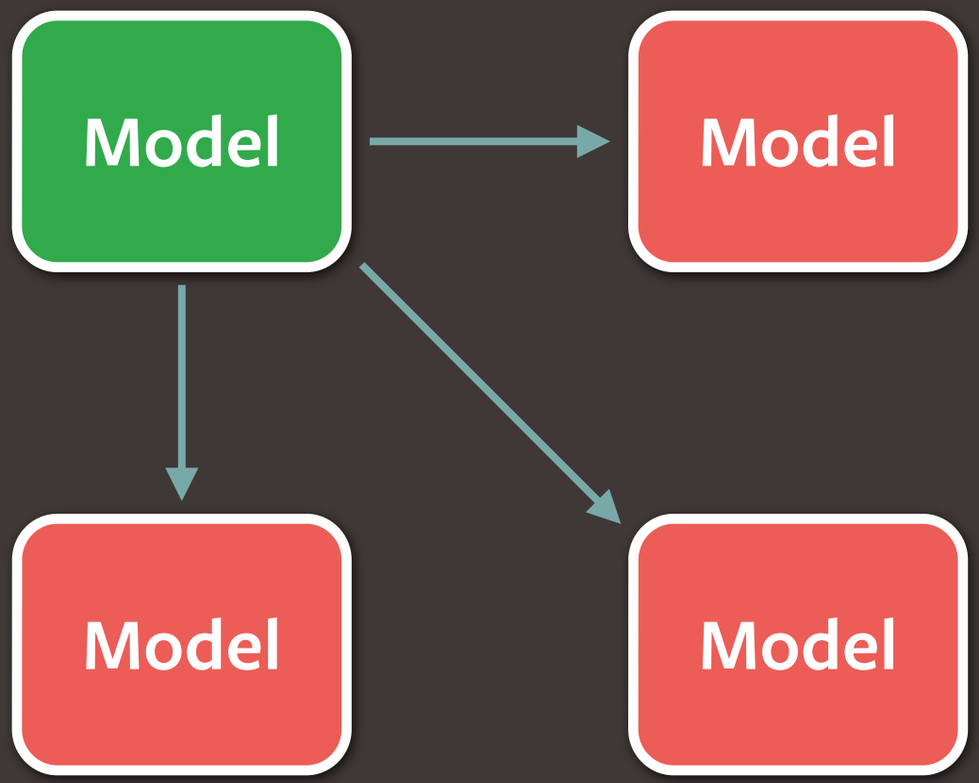
Model

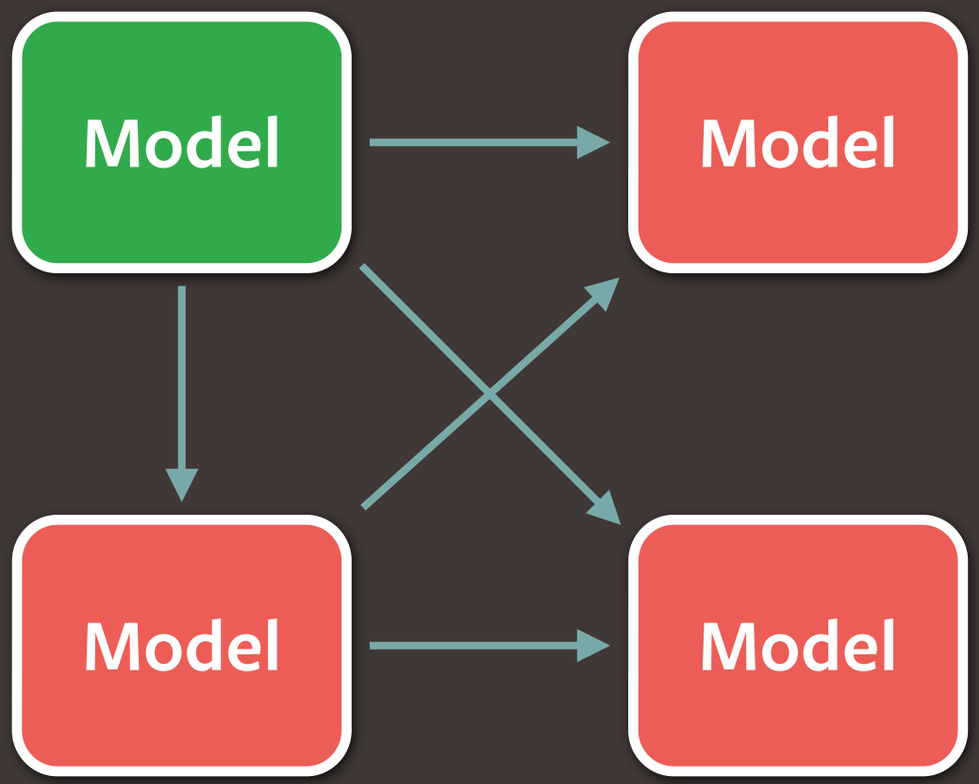

Model
Model

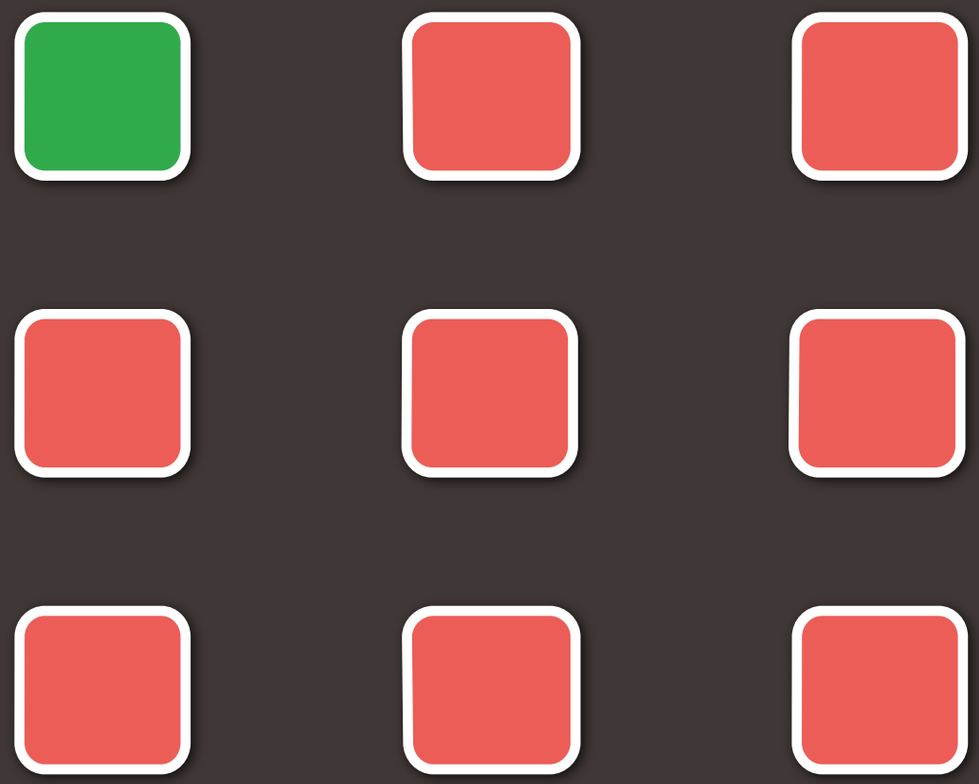

Model
Model

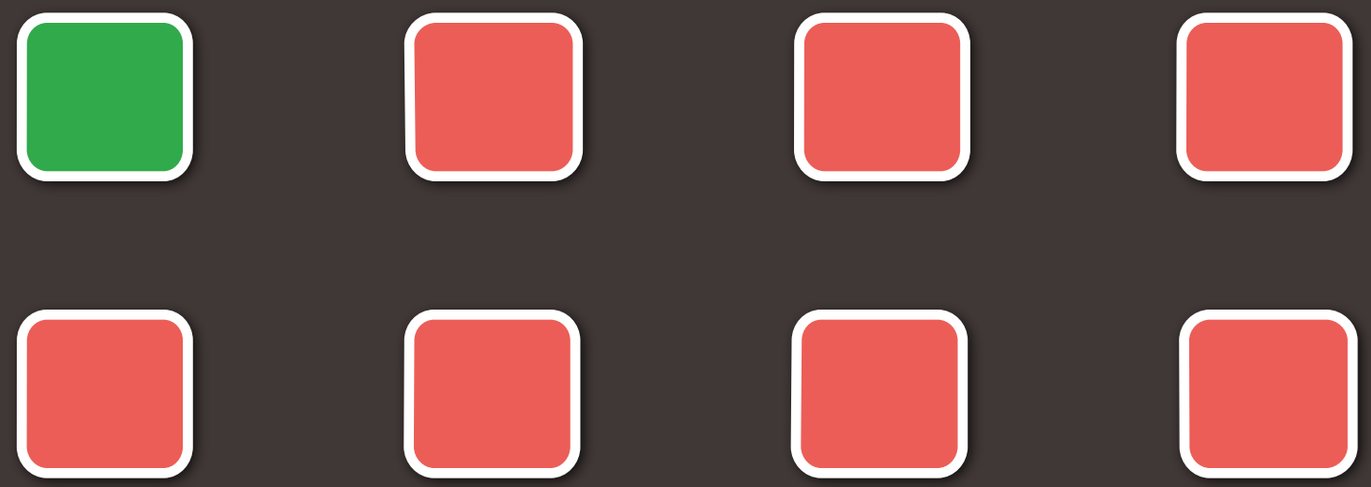

Model
Model

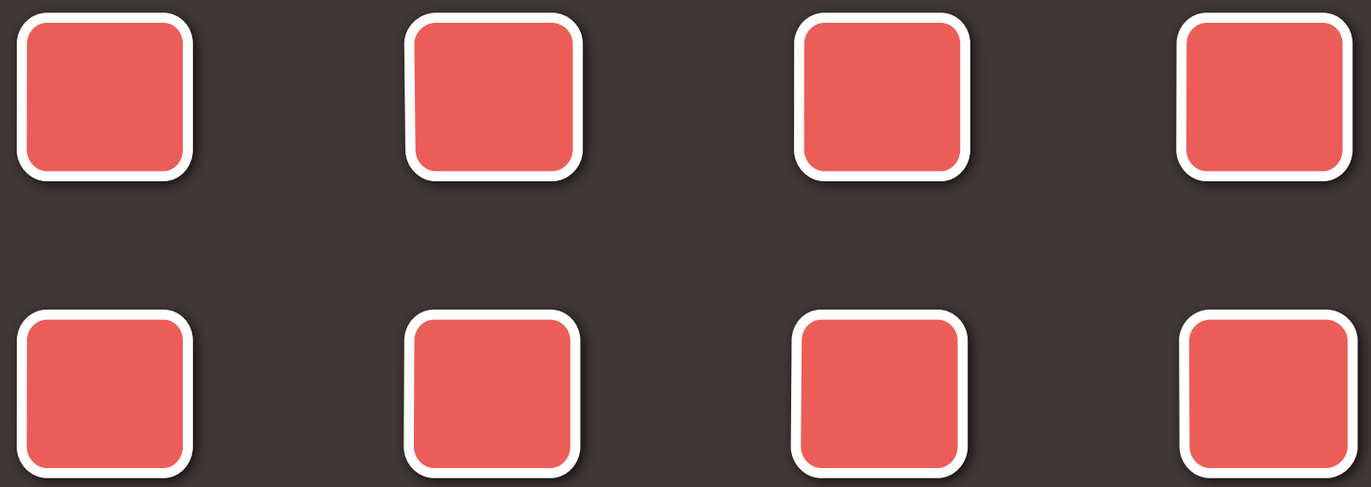

Model
Model


Model
Model
Model
Model
Model

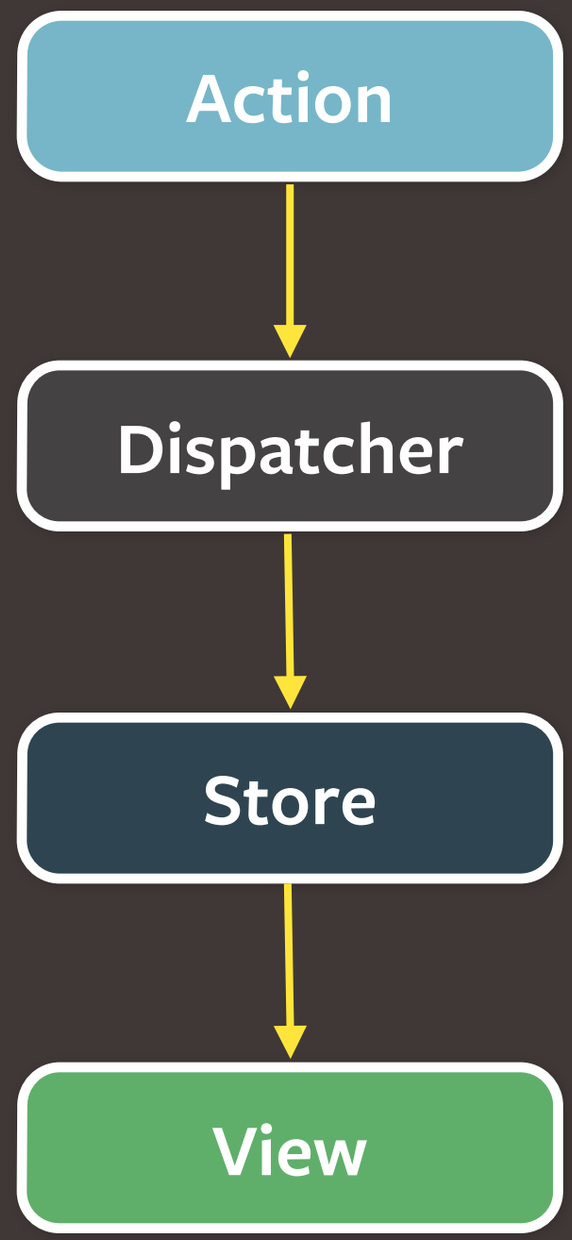

Model
Model
Model
Model







"Tangled wires, Freegeek, Portland, Oregon, USA" by Cory Doctorow, used under CC BY 2.0\


Action
Dispatcher
Store
View

"Couch" by Brian Teutsch, used under CC BY 2.0 / Modified from original

THE PLAN

Anatomy of a React + Flux Application

Actions & Action Creators

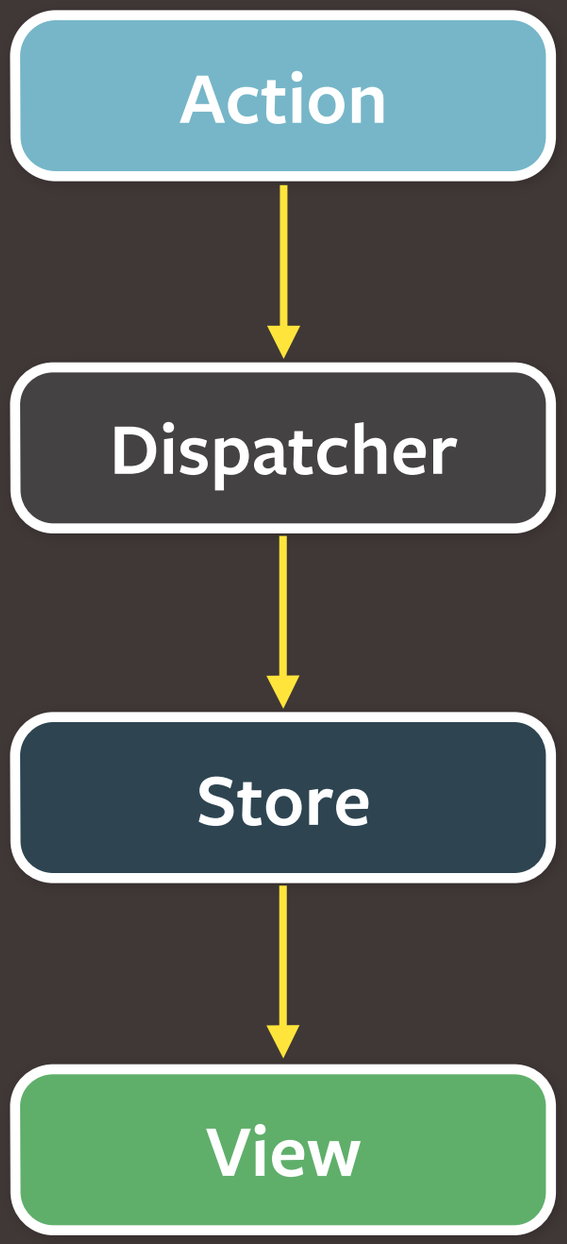
The Dispatcher

Stores + Testing

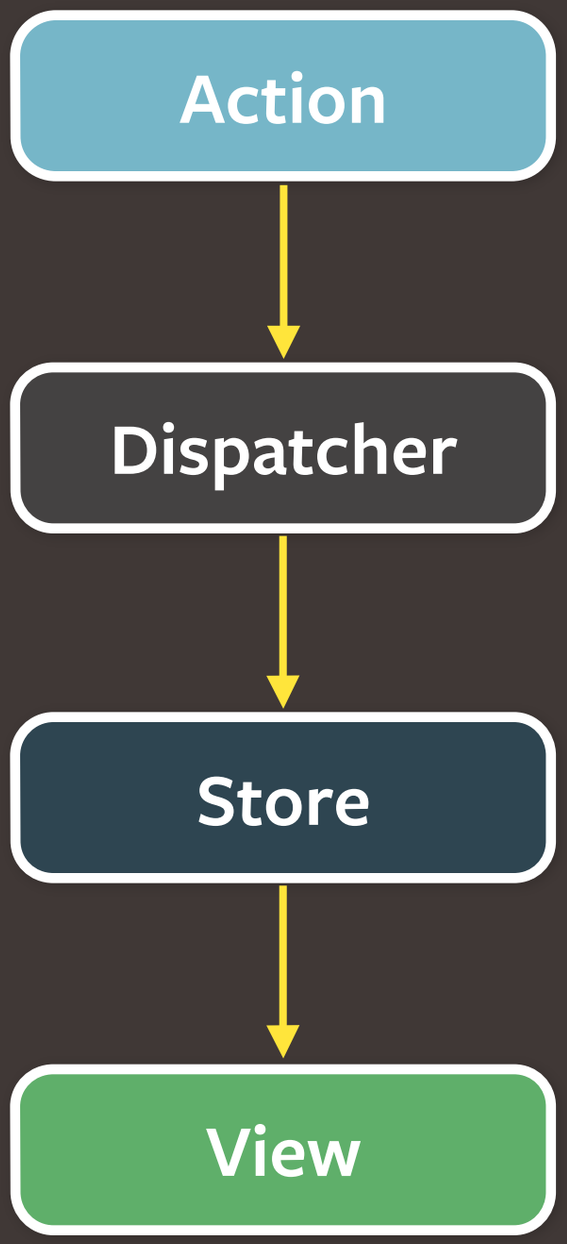
Views, Controller-views and React

Initialization, Interacting with a Web API, Immutable Data, Patterns & Anti-Patterns

```
js
├── actions
│   └── MessageActionCreators.js
├── AppBootstrap.js
├── AppConstants.js
├── AppDispatcher.js
├── stores
│   ├── MessageStore.js
│   └── __tests__
│       └── MessageStore-test.js
├── utils
│   ├── AppWebAPIUtils.js
│   ├── FooUtils.js
│   └── __tests__
│       ├── AppWebAPIUtils-test.js
│       └── FooUtils-test.js
└── views
    ├── MessageControllerView.react.js
    └── MessageListItem.react.js
```




```
js
├── actions
│   └── MessageActionCreators.js
├── AppBootstrap.js
├── AppConstants.js
├── AppDispatcher.js
├── stores
│   ├── MessageStore.js
│   └── __tests__
│       └── MessageStore-test.js
├── utils
│   ├── AppWebAPIUtils.js
│   ├── FooUtils.js
│   └── __tests__
│       ├── AppWebAPIUtils-test.js
│       └── FooUtils-test.js
└── views
    ├── MessageControllerView.react.js
    └── MessageListItem.react.js
```




```
js
├── actions
│   └── MessageActionCreators.js
├── AppBootstrap.js
├── AppConstants.js
├── AppDispatcher.js
├── stores
│   ├── MessageStore.js
│   └── __tests__
│       └── MessageStore-test.js
├── utils
│   ├── AppWebAPIUtils.js
│   ├── FooUtils.js
│   └── __tests__
│       ├── AppWebAPIUtils-test.js
│       └── FooUtils-test.js
└── views
    ├── MessageControllerView.react.js
    └── MessageListItem.react.js
```

ACTIONS & ACTION CREATORS


```
actions
└── MessageActionCreators.js
```

ACTIONS & ACTION CREATORS

Actions:

an object with a type property and new data

Action creators:

semantic methods that create actions

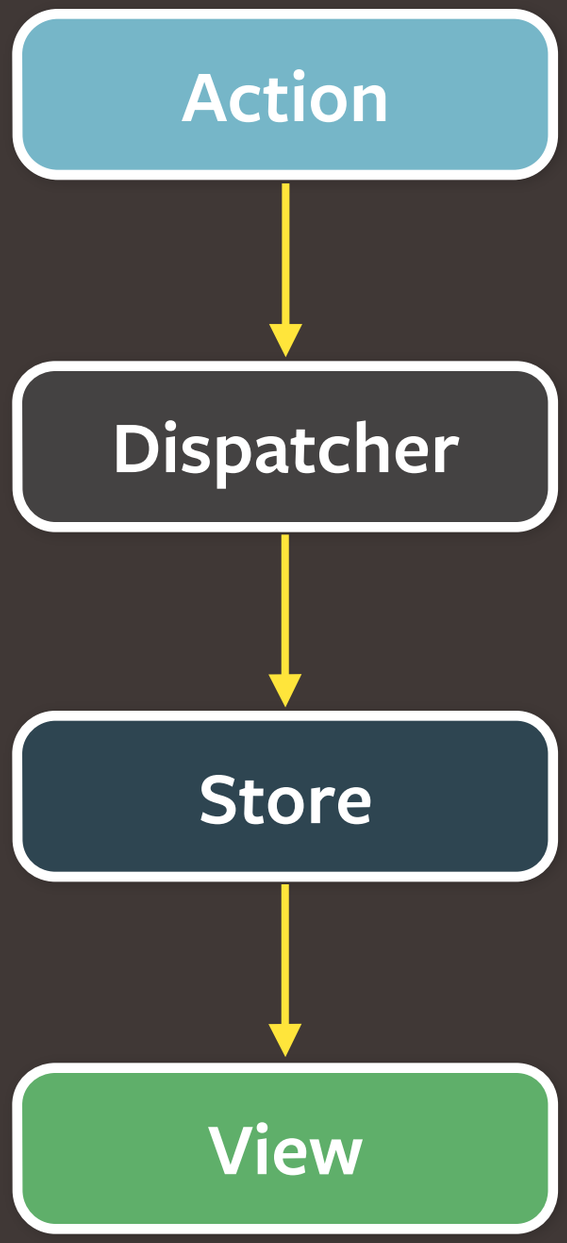
collected together in a module to become an API

```
// MessageActionCreators.js

var AppDispatcher = require('../AppDispatcher');
var AppConstants = require('../AppConstants');

var ActionTypes = Constants.ActionTypes;

module.exports = {

  createMessage: text => {
    AppDispatcher.dispatch({
      type: ActionTypes.MESSAGE_CREATE,
      text
    });
  }

};
```

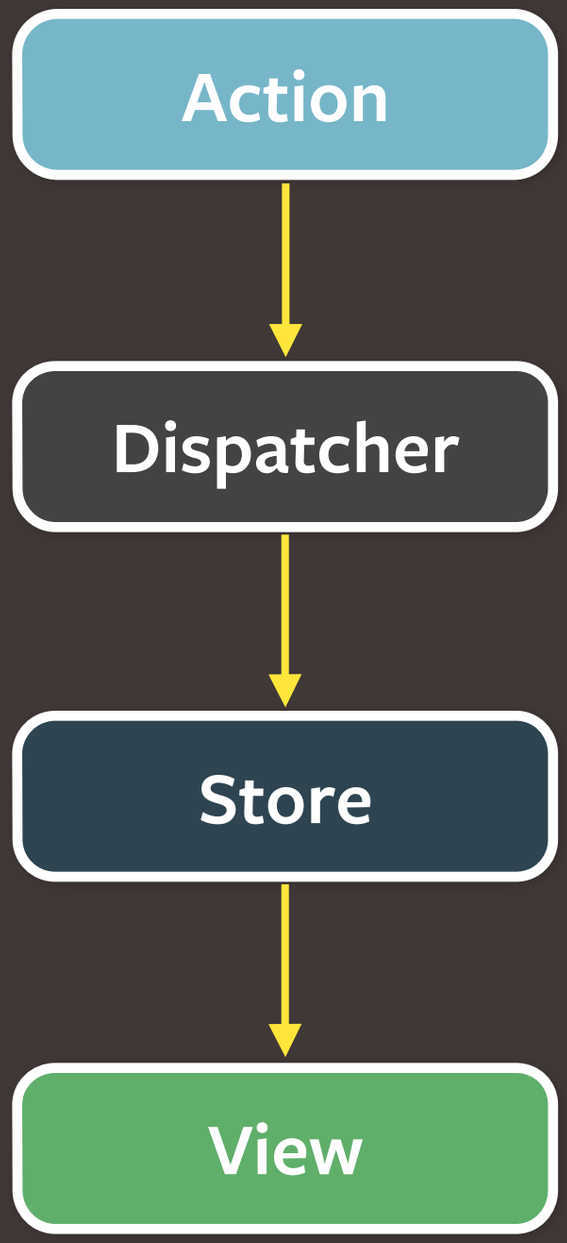
```
createMessage: text => {
  AppDispatcher.dispatch({
    type: ActionTypes.MESSAGE_CREATE,
    text
  });
}
```

```
createMessage: text => {
  AppDispatcher.dispatch({
    type: ActionTypes.MESSAGE_CREATE,
    text
  });
}
```

```
  createMessage: text => {
    AppDispatcher.dispatch({
      type: ActionTypes.MESSAGE_CREATE,
      text
    });
  }
```




```
js
├── actions
│   └── MessageActionCreators.js
├── AppBootstrap.js
├── AppConstants.js
├── AppDispatcher.js
├── stores
│   ├── MessageStore.js
│   └── __tests__
│       └── MessageStore-test.js
├── utils
│   ├── AppWebAPIUtils.js
│   ├── FooUtils.js
│   └── __tests__
│       ├── AppWebAPIUtils-test.js
│       └── FooUtils-test.js
└── views
    ├── MessageControllerView.react.js
    └── MessageListItem.react.js
```




```
js
├── actions
│   └── MessageActionCreators.js
├── AppBootstrap.js
├── AppConstants.js
├── AppDispatcher.js
├── stores
│   ├── MessageStore.js
│   └── __tests__
│       └── MessageStore-test.js
├── utils
│   ├── AppWebAPIUtils.js
│   ├── FooUtils.js
│   └── __tests__
│       ├── AppWebAPIUtils-test.js
│       └── FooUtils-test.js
└── views
    ├── MessageControllerView.react.js
    └── MessageListItem.react.js
```

THE DISPATCHER

Dispatcher

AppDispatcher.js

THE DISPATCHER

Essentially a registry of callbacks

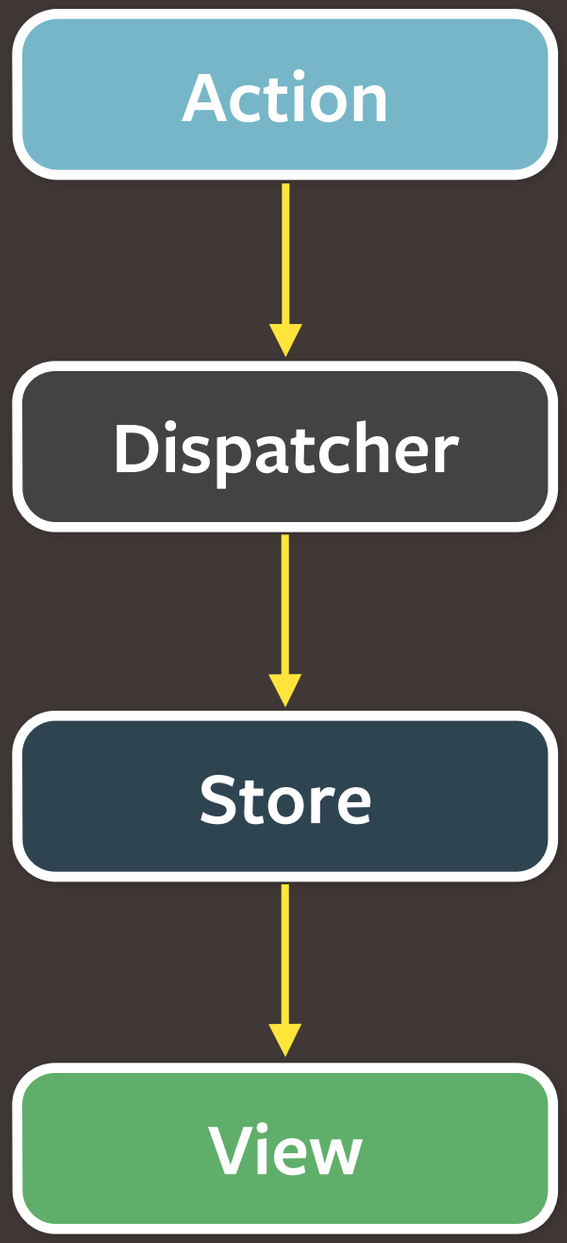
To dispatch, it invokes all the callbacks with a payload

Flux dispatcher is a singleton; payload is an action

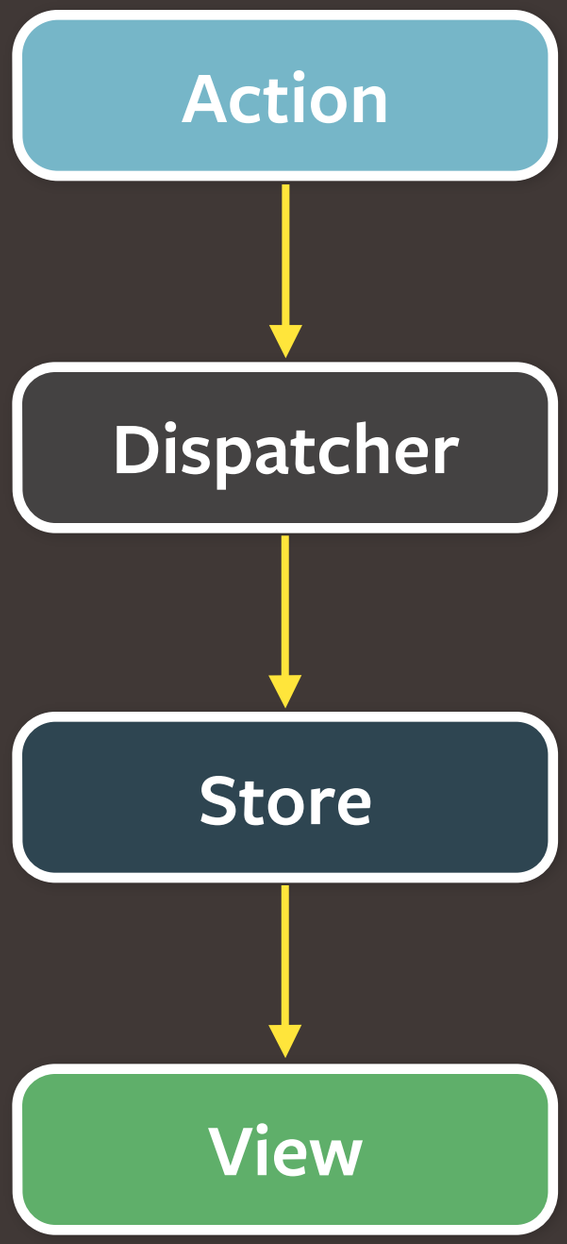
Primary API: `dispatch()`, `register()`, `waitFor()`

Base class is available through npm or Bower.

```
// AppDispatcher.js

var Dispatcher = require('Flux.Dispatcher');

// export singleton
module.exports = new Dispatcher();
```




```
js
├── actions
│   └── MessageActionCreators.js
├── AppBootstrap.js
├── AppConstants.js
├── AppDispatcher.js
├── stores
│   ├── MessageStore.js
│   └── __tests__
│       └── MessageStore-test.js
├── utils
│   ├── AppWebAPIUtils.js
│   ├── FooUtils.js
│   └── __tests__
│       ├── AppWebAPIUtils-test.js
│       └── FooUtils-test.js
└── views
    ├── MessageControllerView.react.js
    └── MessageListItem.react.js
```




```
js
├── actions
│   └── MessageActionCreators.js
├── AppBootstrap.js
├── AppConstants.js
├── AppDispatcher.js
├── stores
│   ├── MessageStore.js
│   └── __tests__
│       └── MessageStore-test.js
├── utils
│   ├── AppWebAPIUtils.js
│   ├── FooUtils.js
│   └── __tests__
│       ├── AppWebAPIUtils-test.js
│       └── FooUtils-test.js
└── views
    ├── MessageControllerView.react.js
    └── MessageListItem.react.js
```

STORES

Store

```
stores
├── MessageStore.js
└── __tests__
    └── MessageStore-test.js
```

STORES

Each store is a singleton

The locus of control within the application

Manages application state for a logical domain

Private variables hold the application data

Numerous collections or values can be held in a store

STORES

Register a callback with the dispatcher

Callback is the only way data gets into the store

No setters, only getters: a universe unto itself

Emits an event when state has changed

```
// MessageStore.js

var _dispatchToken;
var _messages = {};

class MessageStore extends EventEmitter {

  constructor() {
    super();
    _dispatchToken = AppDispatcher.register(action => {

      switch(action.type) {

        case ActionTypes.MESSAGE_CREATE:
          var message = {
            id: Date.now(),
            text: action.text
          }
          _messages[message.id] = message;
          this.emit('change');
          break;

        case ActionTypes.MESSAGE_DELETE:
          delete _messages[action.messageID];
          this.emit('change');
          break;

        default:
          // no op
      }

    });
  }

  getDispatchToken() {
    return _dispatchToken;
  }

  getMessages() {
    return _messages;
  }

}

module.exports = new MessageStore();
```

```
_dispatchToken = AppDispatcher.register(action => {

  switch(action.type) {

    case ActionTypes.MESSAGE_CREATE:
      var message = {
        id: Date.now(),
        text: action.text
      }
      _messages[message.id] = message;
      this.emit('change');
      break;

    case ActionTypes.MESSAGE_DELETE:
      delete _messages[action.messageID];
      this.emit('change');
      break;

    default:
      // no op
  }

});
```

TESTING STORES WITH JEST

Stores have no setters — how to test is not obvious

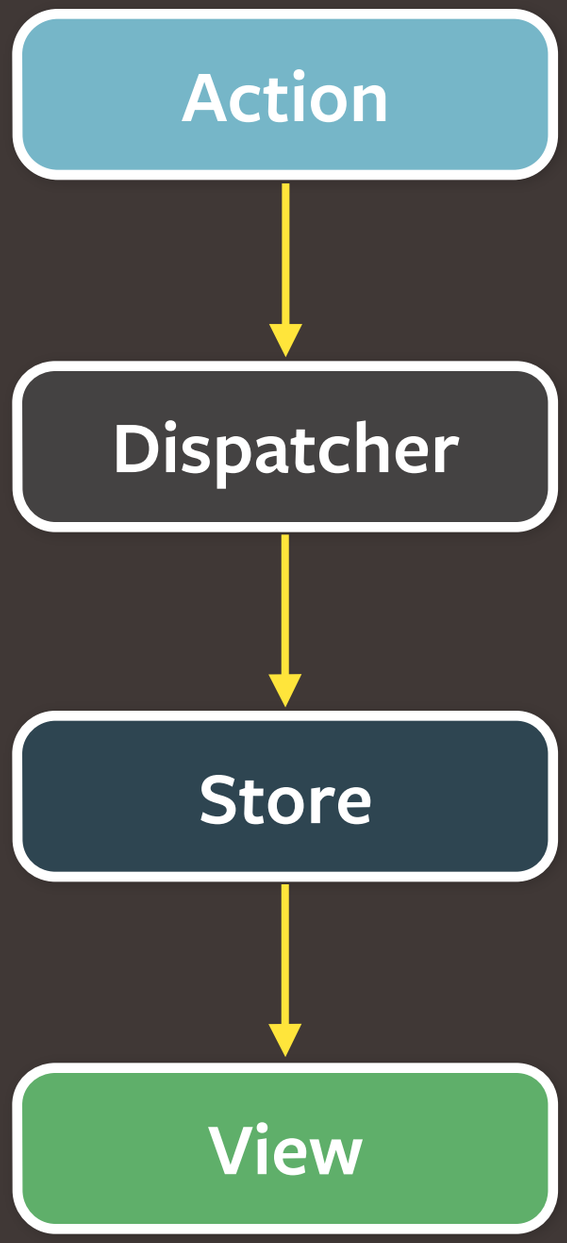
Jest is Facebook's auto-mocking test framework

Built on top of Jasmine

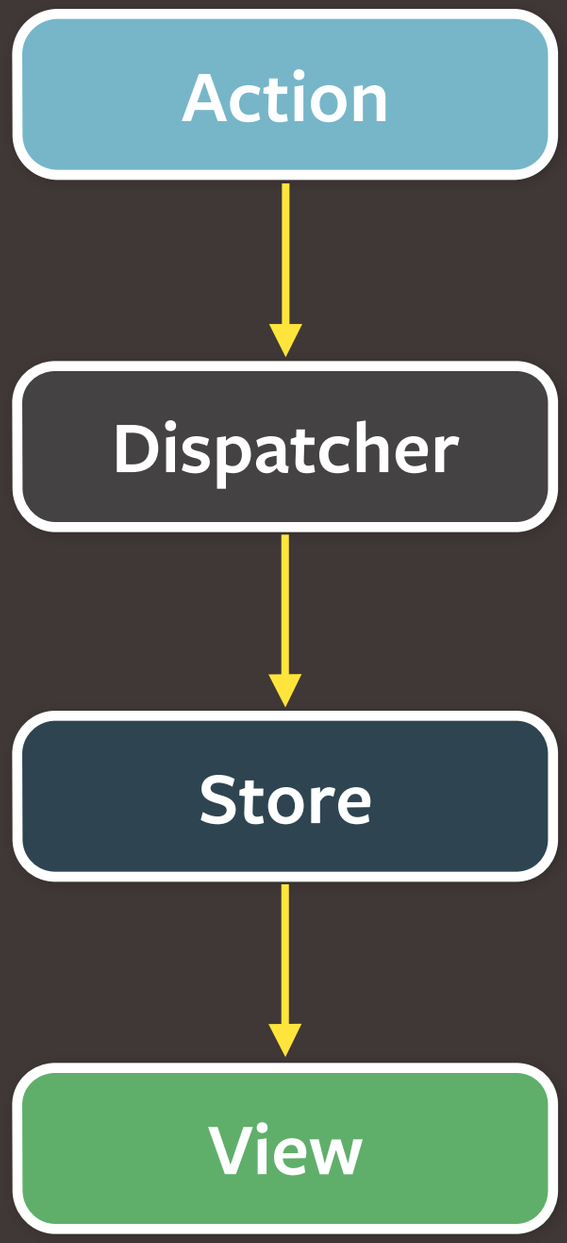
http://facebook.github.io/jest/

```
callback = AppDispatcher.register.mock.calls[0][0];
```

```
jest.dontMock('MessageStore');

var AppConstants = require('AppConstants');

var ActionTypes = AppConstants.ActionTypes;

describe('MessageStore', function() {

  var callback;

  beforeEach(function() {
    AppDispatcher = require('AppDispatcher');
    MessageStore = require('MessageStore');
    callback = AppDispatcher.register.mock.calls[0][0];
  });

  it('can create messages', function() {
    callback({
      type: ActionTypes.MESSAGE_CREATE,
      text: 'test'
    });
    var messages = MessageStore.getMessages();
    var firstKey = Objects.keys(messages)[0];
    expect(MessageStore.getMessages()[firstKey].text).toBe('test');
  });

});
```




```
js
├── actions
│   └── MessageActionCreators.js
├── AppBootstrap.js
├── AppConstants.js
├── AppDispatcher.js
├── stores
│   ├── MessageStore.js
│   └── __tests__
│       └── MessageStore-test.js
├── utils
│   ├── AppWebAPIUtils.js
│   ├── FooUtils.js
│   └── __tests__
│       ├── AppWebAPIUtils-test.js
│       └── FooUtils-test.js
└── views
    ├── MessageControllerView.react.js
    └── MessageListItem.react.js
```




```
js
├── actions
│   └── MessageActionCreators.js
├── AppBootstrap.js
├── AppConstants.js
├── AppDispatcher.js
├── stores
│   ├── MessageStore.js
│   └── __tests__
│       └── MessageStore-test.js
├── utils
│   ├── AppWebAPIUtils.js
│   ├── FooUtils.js
│   └── __tests__
│       ├── AppWebAPIUtils-test.js
│       └── FooUtils-test.js
└── views
    ├── MessageControllerView.react.js
    └── MessageListItem.react.js
```

VIEWS & CONTROLLER VIEWS

View

```
views
├── MessageControllerView.react.js
└── MessageListItem.react.js
```

VIEWS & CONTROLLER VIEWS

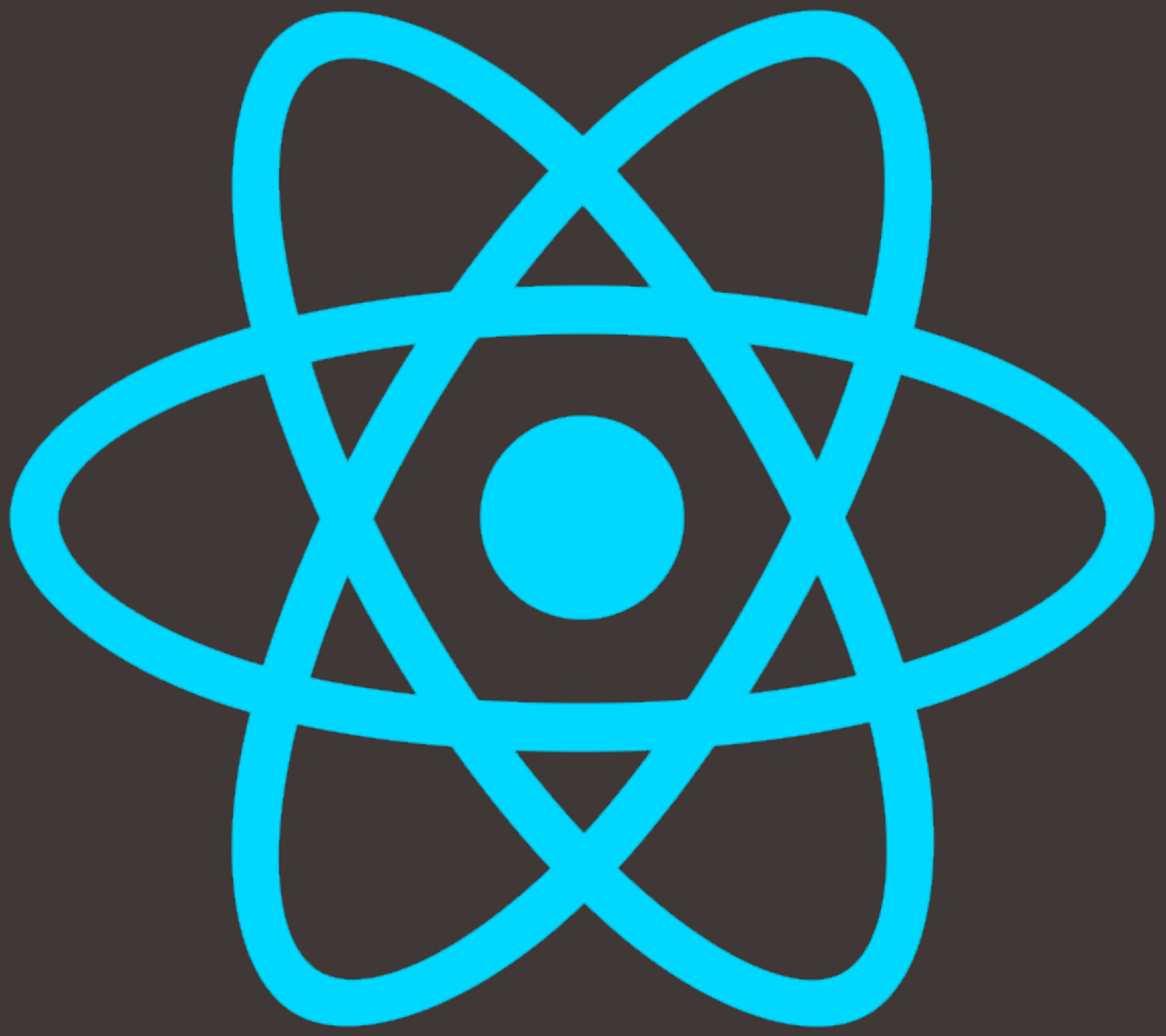
Tree of React components

Controller views are near the root, listen for change events

On change, controller views query stores for new data

With new data, they re-render themselves & children

REACT



REACT AND THE DOM

Component-based system for managing DOM updates

Uses a “Virtual DOM”: data structure and algorithm

Updates the DOM as efficiently as possible

Huge performance boost

Bonus: we can stop thinking about managing the DOM

REACT'S PARADIGM

Based on Functional-Reactive principles

Unidirectional data flow

Composability

Predictable, Reliable, Testable

Declarative: what the UI should look like, given props

USING REACT

Data is provided through props

Rendering is a function of `this.props` and `this.state`

“Re-render” (or not) on every state change

Keep components as stateless as possible

Component lifecycle and update cycle methods

Lifecycle:
Mounting and Unmounting

getDefaultProps()

getInitialState()

componentWillMount()

render()

---------- DOM Mutations Complete ----------

componentDidMount()

---

componentWillUnmount()

Update:
New Props or State

componentWillReceiveProps()*

shouldComponentUpdate()

componentWillUpdate()

render()

componentDidUpdate()

**Called only with new props*

Lifecycle: Mounting and Unmounting	Update: New Props or State
getDefaultProps()	componentWillReceiveProps()*
getInitialState()	shouldComponentUpdate()
componentWillMount()	componentWillUpdate()
render()	render()
---------- DOM Mutations Complete ----------	
componentDidMount()	componentDidUpdate()
----------	
componentWillUnmount()	

**Called only with new props*

```
// MessagesControllerView.react.js

var React = require('react');

var MessagesControllerView = React.createClass({

  render: function() {
    // TODO
  }

});

module.exports = MessagesControllerView;
```

```
// MessagesControllerView.react.js

var MessagesControllerView = React.createClass({

  render: function() {
    // TODO
  }

});
```

```
// MessagesControllerView.react.js

var MessagesControllerView = React.createClass({

  componentDidMount: function() {
    MessageStore.on('change', this._onChange);
  },

  componentWillUnmount: function() {
    MessageStore.removeListener('change', this._onChange);
  },

  render: function() {
    // TODO
  }

});
```

```
// MessagesControllerView.react.js

var MessagesControllerView = React.createClass({

  getInitialState: function() {
    return { messages: MessageStore.getMessages() };
  },

  componentDidMount: function() {
    MessageStore.on('change', this._onChange);
  },

  componentWillUnmount: function() {
    MessageStore.removeListener('change', this._onChange);
  },

  render: function() {
    // TODO
  },

  _onChange: function() {
    this.setState({ messages: MessageStore.getMessages() });
  }

});
```

```
// MessagesControllerView.react.js

var MessagesControllerView = React.createClass({

  getInitialState: function() {
    return { messages: MessageStore.getMessages() };
  },

  componentDidMount: function() {
    MessageStore.on('change', this._onChange);
  },

  componentWillUnmount: function() {
    MessageStore.removeListener('change', this._onChange);
  },

  render: function() {
    // TODO
  },

  _onChange: function() {
    this.setState({ messages: MessageStore.getMessages() });
  }

});
```

```
render: function() {
  // TODO
},
```

```
render: function() {
  return (
    <ul>
    </ul>
  );
},
```

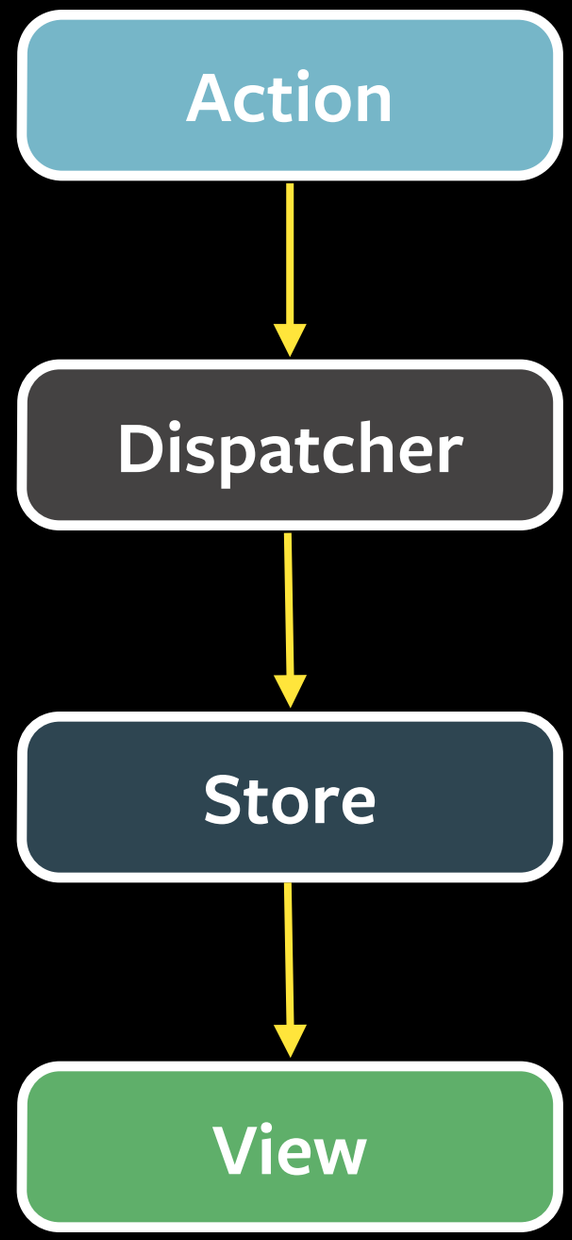
```
render: function() {
  var messageListItems = [];
  return (
    <ul>
      {messageListItems}
    </ul>
  );
},
```

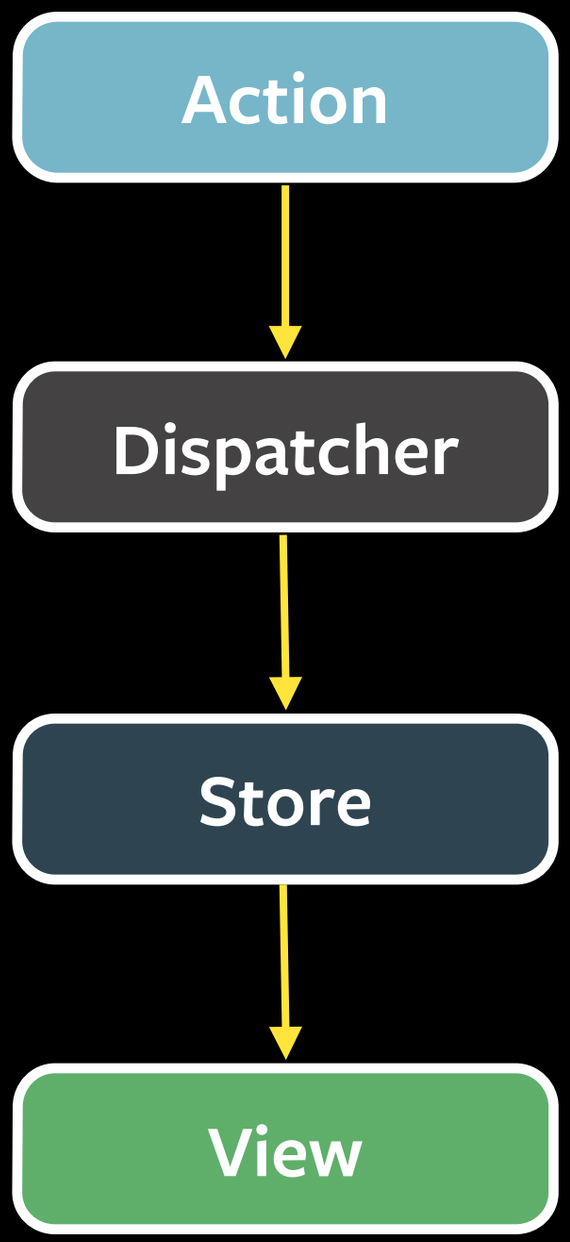
```
render: function() {
  var messageListItems = this.state.messages.map(message => {
    return (
      <MessageListItem
        key={message.id}
        messageID={message.id}
        text={message.text}
      />
    );
  });
  return (
    <ul>
      {messageListItems}
    </ul>
  );
},
```

```
// MessageListItem.react.js

var React = require('react');

var MessageListItem = React.createClass({

  propTypes = {
    messageID: React.PropTypes.number.isRequired,
    text: React.PropTypes.string.isRequired
  },

  render: function() {
    return (
      <li>
        {this.props.text}
      </li>
    );
  }

});

module.exports = MessageListItem;
```

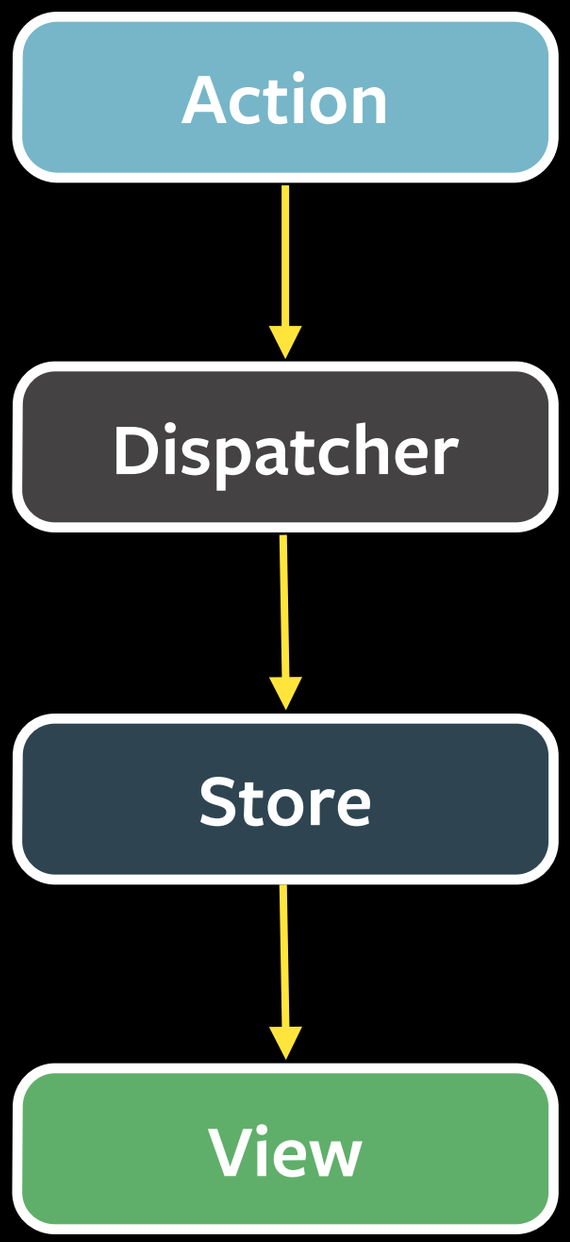
```
// MessageListItem.react.js

var MessageActionCreators = require('MessageActionCreators');
var React = require('react');

var MessageListItem = React.createClass({

  propTypes = {
    messageID: React.PropTypes.number.isRequired,
    text: React.PropTypes.string.isRequired
  },

  render: function() {
    return (
      <li onClick={this._onClick}>
        {this.props.text}
      </li>
    );
  },

  _onClick: function() {
    MessageActionCreators.deleteMessage(this.props.messageID);
  }

});

module.exports = MessageListItem;
```

```
// MessageListItem.react.js

var MessageActionCreators = require('MessageActionCreators');
var React = require('react');

var MessageListItem = React.createClass({

  propTypes = {
    messageID: React.PropTypes.number.isRequired,
    text: React.PropTypes.string.isRequired
  },

  render: function() {
    return (
      <li onClick={this._onClick}>
        {this.props.text}
      </li>
    );
  },

  _onClick: function() {
    MessageActionCreators.deleteMessage(this.props.messageID);
  }

});

module.exports = MessageListItem;
```


Action
Dispatcher
Store
View




```
deleteMessage: messageID => {
  AppDispatcher.dispatch({
    type: ActionTypes.MESSAGE_DELETE,
    messageID
  });
}
```




```
deleteMessage: messageID => {
  AppDispatcher.dispatch({
    type: ActionTypes.MESSAGE_DELETE,
    messageID
  });
}
```

```
case ActionTypes.MESSAGE_DELETE:
  delete _messages[action.messageID];
  this.emit('change');
  break;
```

INITIALIZATION OF THE APP

Usually done in a bootstrap module

Initializes stores with an action

Renders the topmost React component

```
// AppBootstrap.js

var AppConstants = require('AppConstants');
var AppDispatcher = require('AppDispatcher');
var AppRoot = require('AppRoot.react');
var React = require('React');

require('FriendStore');
require('LoggingStore');
require('MessageStore');

module.exports = (initialData, elem) => {
  AppDispatcher.dispatch({
    type: AppConstants.ActionTypes.INITIALIZE,
    initialData
  });
  React.render(<AppRoot />, elem);
};
```

CALLING A WEB API

Use a WebAPIUtils module to encapsulate XHR work.

Start requests directly in the Action Creators, or in the stores.

Important: create a new action on success/error.

Data must *enter* the system through an action.

IMMUTABLE DATA

Boost performance in React's `shouldComponentUpdate()`

React.addons.PureRenderMixin

immutable-js: http://facebook.github.io/immutable-js/

Lifecycle:
Mounting and Unmounting

getDefaultProps()

getInitialState()

componentWillMount()

render()

---------- DOM Mutations Complete ----------

componentDidMount()

---

componentWillUnmount()

Update:
New Props or State

componentWillReceiveProps()*

shouldComponentUpdate()

componentWillUpdate()

render()

componentDidUpdate()

**Called only with new props*

MORE FLUX PATTERNS

LoggingStore

Error handling with client ID / dirty bit

Error handling with actions queue

Resolving dependencies in the Controller-view

ANTI-PATTERNS FOR REACT+FLUX

Getters in `render()`

Public setters in stores & the setter mindset trap

Application state/logic in React components

Props in `getInitialState()`

Dataflow Programming

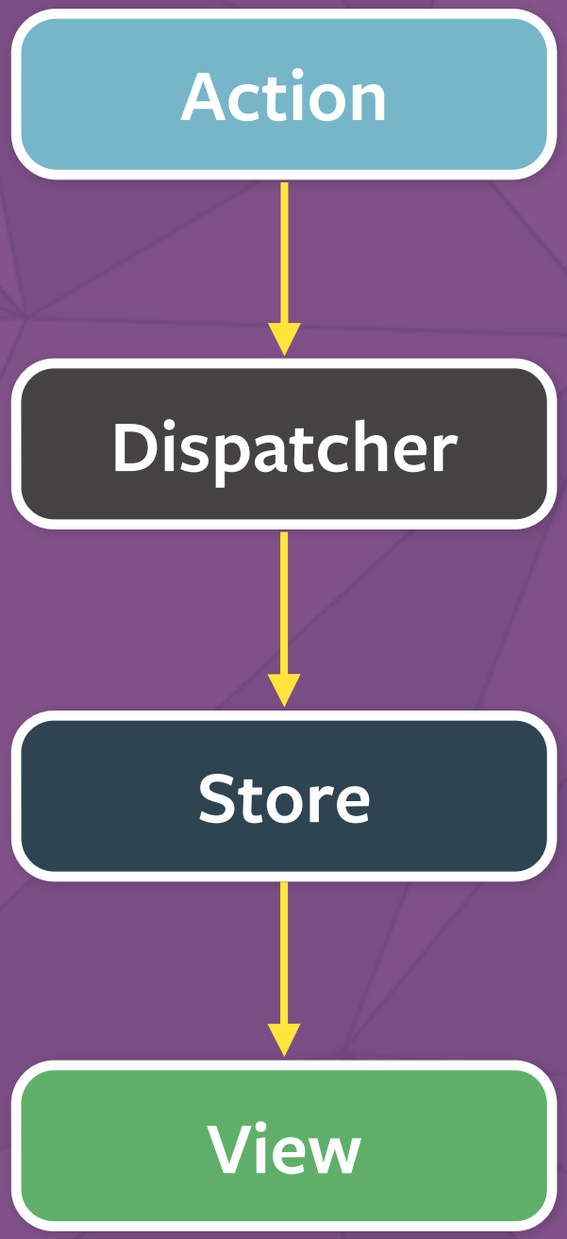
CQRS

Reactive

FLUX

MVC

Functional and FRP



FLUX TOTALLY WORKS

THANK YOU!