Flux

A simple architecture model to build **Client-side apps!**
Pedro Nauck
FRONTEND DEVELOPER

@pedronauck
pedronauck.com
why we choose MVC?
at all times...
Because we want to achieve
We want scalability, performance, adaptability, maintainability, and decoupling.
MVC

- Controller
- View
- Model

The diagram illustrates the Model-View-Controller (MVC) architecture, where the Controller acts as an intermediary between the View and the Model.
Tim B. Lee
1991
Hola! :) Fantasiosa_Putilla

<davidov82> hola wapa
<davidov82> te apetece k lo hagamos perdidos x la montaña
<Fantasiosa_Putilla> Venga, Ok
<davidov82> empieza wapa
<Fantasiosa_Putilla> Después de ir a la montaña, nos perdemos
<davidov82> estamos solos sin saber k hacer
<Fantasiosa_Putilla> Sin comida, ni leche fresca
<davidov82> sin nada de nada
<davidov82> solod tu y yo perdidos
<davidov82> escuchando ruidos en una noche muy oscura
<Fantasiosa_Putilla> Muriedome de sed
<Fantasiosa_Putilla> Con ganas de algo calentito en mi boquita para poder comer...
<davidov82> la boca hambrienta
<davidov82> hace calor

Ver más en www.IKEA.es
Backend Rulez!
write more, do less.
what is flux?
just a **simple** model

- Action
- Dispatcher
- Store
- View
Unidirectional Dataflow
focused on client-side APPS
Because we want to achieve
We want

scalability
performance
adaptability
maintainability
decoupling
what isn't flux?
Framework or Library
The dispatcher is the central hub that manages all data flow in a Flux application. It is essentially a registry of callbacks into the stores and has no real intelligence of its own — it is a simple mechanism for distributing the actions to the stores.

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

✓ Central hub of your app
✓ Only one dispatcher per application
✓ Manage data based on named actions
✓ Simple way to apply on your app
✓ Register callback to the stores
Callbacks
All actions are registered on the dispatcher, receives some payload, then are sent to a specific store.
var Dispatcher = require('flux').Dispatcher;

Dispatcher
just a simple library
The dispatcher exposes a method that allows us to trigger a dispatch to the stores, and to include a payload of data, which we call an action.

http://facebook.github.io/flux
var AppDispatcher = require('flux').Dispatcher;

var Action = {
    createItem: function(text) {
        AppDispatcher.dispatch({
            type: 'ADD_ITEM',
            data: {
                text: text
            }
        });
    }
};
var AppDispatcher = require('flux').Dispatcher;

var Action = {
    createItem: function(text) {
        AppDispatcher.dispatch({
            type: 'ADD_ITEM',
            data: {
                text: text
            }
        });
    }
};
var AppDispatcher = require('flux').Dispatcher;

var Action = {
  createItem: function(text) {
    AppDispatcher.dispatch({
      type: 'ADD_ITEM',
      data: {
        text: text
      }
    })
  }
};
Actions
creating your action

var AppDispatcher = require('flux').Dispatcher;

var Action = {
    createItem: function(text) {
        AppDispatcher.dispatch({
            type: 'ADD_ITEM',
            data: {
                text: text
            }
        })
    }
};
View triggers

When the user handle some action, the view triggers an action to the dispatcher.
Stores contain the **application state and logic**. Their role is somewhat similar to a model in a traditional MVC, but they manage the state of many objects — they do not represent a single record of data like ORM models do. Nor are they the same as Backbone's collections. More than simply managing a collection of ORM-style objects.

Stores

✓ Just a simple object-literal
✓ Contain the application state and logic
✓ Public interface: getters, no setter
✓ Setup: register with the dispatcher
✓ Emits a change event
✓ Receives data through a payload
Payloads

Store receives new data in the form of a payload, that is just an object literal registered on the dispatcher.
stores

receiving data

```javascript
var _data = []; 

var Store = {
    getAll: function() {
        return {
            data: _data
        };
    }
};
```
Stores

receiving data

var _data = [];

var Store = {
    getAll: function() {
        return {
            data: _data
        };
    }
};
Stores

var _data = [];

var Store = {
    getAll: function(){
        return {
            data: _data
        };
    }
};
var _data = [];

var Store = {
    getAll: function() {
        return {
            data: _data
        };
    }
};
dispatcherIndex: AppDispatcher.register(function(payload) {
    switch(payload.type) {
    case 'ADD_ITEM':
        _data.push(payload.data);
        break;
    case 'REMOVE_ITEM':
        // logic to remove data
        break;
    }
})
dispatcherIndex: AppDispatcher.register(function(payload) {
  switch(payload.type) {
    case 'ADD_ITEM':
      _data.push(payload.data);
      break;
    case 'REMOVE_ITEM':
      // logic to remove data
      break;
  }
})
dispatcherIndex: AppDispatcher.register(function(payload) {
    switch(payload.type) {
    case 'ADD_ITEM':
        _data.push(payload.data);
        break;
    case 'REMOVE_ITEM':
        // logic to remove data
        break;
    }
})
dispatcherIndex: AppDispatcher.register(function(payload) {
  switch(payload.type) {
    case 'ADD_ITEM':
      _data.push(payload.data);
      break;
    case 'REMOVE_ITEM':
      // logic to remove data
      break;
  }
})
dispatcherIndex: AppDispatcher.register(function(payload) {
    switch(payload.type) {
        case 'ADD_ITEM':
            _data.push(payload.data);
            break;
        case 'REMOVE_ITEM':
            // logic to remove data
            break;
    }
})
Updating Views

A view is updated when the store emit some change event.
Stores based on events

```javascript
addChangeListener: function (callback) {
    this.on('change', callback);
},

removeChangeListener: function (callback) {
    this.removeListener('change', callback);
},

emitChange: function () {
    this.emit('change');
}
```
Stores
based on events

addChangeListener: function(callback) {
    this.on('change', callback);
},

removeChangeListener: function(callback) {
    this.removeListener('change', callback);
},

emitChange: function() {
    this.emit('change');
}
Stores
based on events

addChangeListener: function(callback) {
    this.on('change', callback);
},

removeChangeListener: function(callback) {
    this.removeListener('change', callback);
},

emitChange: function() {
    this.emit('change');
}
Stores based on events

addChangeListener: function(callback) {
    this.on('change', callback);
},

removeChangeListener: function(callback) {
    this.removeListener('change', callback);
},

emitChange: function() {
    this.emit('change');
}
Stores
registering on the dispatcher

dispatcherIndex: AppDispatcher.register(function(payload) {
    switch(payload.type) {
        case 'ADD_ITEM':
            _data.push(payload.data);
            this.emitChange();
            break;
        case 'REMOVE_ITEM':
            // logic to remove data
            break;
    }
})
dispatcherIndex: AppDispatcher.register(function(payload) {
    switch(payload.type) {
        case 'ADD_ITEM':
            _data.push(payload.data);
            this.emitChange();
            break;
        case 'REMOVE_ITEM':
            // logic to remove data
            break;
    }
})

Stores
registering on the dispatcher

emit change
to the view
Fetching Asynchronous
How fetch data from remote source
Action

Dispatcher

Store

View
var ItemsAPI = require('./ItemsApi');
var AppDispatcher = require('flux').Dispatcher;

var Action = {
    searchItems: function(query) {
        AppDispatcher.dispatch({ type: 'SEARCH_ITEMS' });

        ItemsApi.search(query).done(function(data) {
            AppDispatcher.dispatch({
                type: 'SEARCH_ITEMS_SUCCESS',
                data: data
            });
        });
    }
};
var ItemsAPI = require('./ItemsApi');
var AppDispatcher = require('flux').Dispatcher;

var Action = {
  searchItems: function(query) {
    AppDispatcher.dispatch({ type: 'SEARCH_ITEMS' });

    ItemsAPI.search(query).done(function(data) {
      AppDispatcher.dispatch({
        type: 'SEARCH_ITEMS_SUCCESS',
        data: data
      });
    });
  }
};
var ItemsAPI = require('./ItemsApi');
var AppDispatcher = require('flux').Dispatcher;

var Action = {
  searchItems: function(query) {
    AppDispatcher.dispatch({ type: 'SEARCH_ITEMS' });
    ItemsApi.search(query).done(function(data) {
      AppDispatcher.dispatch(
        { type: 'SEARCH_ITEMS_SUCCESS',
          data: data
        });
    });
  }
};

Actions

fetching data from remote
var ItemsAPI = require('./ItemsApi');
var AppDispatcher = require('flux').Dispatcher;

var Action = {
    searchItems: function(query) {
        AppDispatcher.dispatch({ type: 'SEARCH_ITEMS' });
        ItemsApi.search(query).done(function(data) {
            AppDispatcher.dispatch({
                type: 'SEARCH_ITEMS_SUCCESS',
                data: data
            });
        });
    }
};

Actions
fetching data from remote
var ItemsAPI = require('./ItemsApi');
var AppDispatcher = require('flux').Dispatcher;

var Action = {
  searchItems: function(query) {
    AppDispatcher.dispatch({ type: 'SEARCH_ITEMS' });

    ItemsApi.search(query).done(function(data) {
      AppDispatcher.dispatch({
          type: 'SEARCH_ITEMS_SUCCESS',
          data: data
        });
    });
  }
};
var ItemsAPI = require('./ItemsApi');
var AppDispatcher = require('flux').Dispatcher;

var Action = {
    searchItems: function(query) {
        AppDispatcher.dispatch({ type: 'SEARCH_ITEMS' });
        ItemsApi.search(query).done(function(data) {
            AppDispatcher.dispatch({
                type: 'SEARCH_ITEMS_SUCCESS',
                data: data
            });
        });
    }
};
Demo

https://github.com/facebook/flux/tree/master/examples/flux-todomvc/
That’s a lot of code, no?
Flummox

Idiomatic, modular, testable, isomorphic Flux. No singletons required.

$ npm install --save flummox

Join the #flummox channel of the Reactiflux Slack community.

- API documentation
- Quick start guide
- React integration guide
Fluxxor is a set of tools to facilitate building JavaScript data layers using the Flux architecture by reifying many of the core Flux concepts. It works particularly well in conjunction with React as the view layer, and contains a few helpers to make integration with React applications easier.

The Flux architecture...

- makes it easier to reason about changes to your application's data
- eschews complex MVC hierarchies in favor of a one-way data flow
- helps improve data consistency
- prevents hard-to-debug cascading updates
- works great with React and complements its reactive data flow

Want to learn more? Start by checking out What is Flux.

To get started, `npm install fluxxor` or see other ways to install.
A completely agnostic JavaScript framework to apply Flux concepts into your interfaces easily.
McFly
Flux Architecture Made Easy

npm install mcfly

Download  Github
Learn more about

https://reactjsnews.com/the-state-of-flux/
http://blog.andrewray.me/flux-for-stupid-people/
http://facebook.github.io/flux/docs/overview.html#content
React: Flux Architecture

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, and you can start using Flux immediately without a lot of new code.

In this series we will explore the Flux Architecture for Facebook's React framework. From the basics through building a working example application, you'll learn the ins and outs of practical, real-world React development.

Be sure to check out the React Fundamentals series for an introduction to React.

Browse the React: Flux Architecture series.

showing All 9 lessons...
Facebook engineers

React and Flux: Building Applications with a Unidirectional Data Flow

https://www.youtube.com/watch?v=i__96gnoyAM
I hope you enjoyed
Pedro Nauck
FRONTEND DEVELOPER

@pedronauck
pedronauck.com
Questions?