## Overview

Data flows through an application

Not a framework

It’s just a pattern

There are several different implementations

Deals with actions and data changes.

Prioritizes explicit, verbose code that’s easy to reason about.

There’s a good amount of boilerplate needed.

## Code organization / best-practices

Create a separate folder for each component of flux

Dispatcher

Actions

Stores

Constants

Create a separate file for each group of related actions

Create a separate file for each group of related constants

Store action types in a constants file as a form of documentation

Enables people to view all the actions in the app simply by viewing the constant files.

Create a separate store for each major domain / concept in the application

Start private variable names with an underscore

Store initialization actions in a separate actions file

Initialization actions bootstrap the app

## Binding

Flux is a reaction to traditional MVC

In traditional MVC, a controller ends up working with multiple models ends up working with multiple views, and vice-versa

In other words, there are a lot of dependencies, in both directions.

Updates to a model may cascade to multiple views, which may cascade to multiple models, etc.

The unidirectional flow breaks the cascade into an easy to think about flow.

Two-way binding

Angular, ember, knockout

When you enter data in a text box, the corresponding data behind the scenes is updated immediately.

In contrast, typing data into a text box triggers an “action”.

This action is sent through the dispatcher

The dispatcher then notifies any stores which have registered interest with the dispatcher that an action has occurred.

When a store changes, the react component is updated.

Differes from unidirectional flows mainly because unidirectional flows don’t directly update state.

Requires less typing than unidirectional binding and is simpler to understand when implementing

View models may send cascading updates to other views and view-models

Unidirectional flows are more deterministic

Requires more code

Benefits from clarity, testability and predictability

## Actions

User interactions which occur in react components

Click button

Delete user

Add item

May be triggered by

* user interaction
* server action

Actions send a payload to the dispatcher.

Actions often invoke AJAX web-API’s to get / send data.

Server actions may include

Page load

Errors occur during calls to server

Action payload

Always contains

Type

Tells store what event just occurred

Data

Varies depending on the action

Encapsulate events that occur in an application

A payload of data representing an event triggered either by a user interaction with the app or a server event.

Action creators are dispatcher helper methods.

Describe all the events that are possible in an application

Typically grouped together by similar use case

Create types that are stored in a constants file

The type is used by the dispatcher to properly handle the action and pass the action on to related stores

There may be dependencies between stores

One store may need to be updated before another can.

Stores can declaratively wait on other stores to update before being updated themselves.

The dispatcher may notify stores in a specific order.

Views call actions in response to user interaction.

Actions can be used to initialize a store

Trigger an initialization action from the app entry-point (main.js, for example)

## Dispatcher

Singleton registry

Centralized list of callbacks

Invokes stores

Only one dispatcher per app

A central hub through which all data must flow.

Exposes methods which allow dispatches to be triggered to the stores containing a payload of data which is called an action.

Dispatches actions to stores

Different from traditional pub/sub systems in that every store is notified of actions

Stores register with a dispatcher so they can be notified when data changes

Dispatcher holds a list of callbacks

Dispatcher exists to get actions from views to stores.

Decouples views from state

Constants

Constants file

Keep things organized by defining constants used throughout the system in a centralized place.

Provides a good documentation of all the things an app actually does.

All updates to app state occur via a centralized dispatcher

Dispatches data to app’s datastores

Data flows in one direction, as opposed to Angular

Flux is Facebook’s name for a unidirectional dataflow with a centralized dispatcher

Very simple.

Basically just holds a list of callbacks

Flow implementations

Facebook

Alt

Reflux

Flumix

Marty

Fluxxor

Delorean

Redux

NuclearJS

Fluxible

Facebook’s implementation \_does\_ have a lot of boilerplate necessary, but once you get used to it, it’s easy to understand and is scalable.

Some implementations are less verbose at the expense of clarity or flexibility.

## Store

Hold an application’s data

Hold app state, logic, and data retrieval methods

A store is not a model.

Instead, it contains models

An app can contain one, or many stores.

As an app gets larger, it is often convenient and helpful to have multiple stores

Keeps related data grouped together.

Registers a callback with the dispatcher

Only stores can register callbacks with the dispatcher

Never register a component callback with the dispatcher.

Dispatcher.register(function(action) {});

Extended with Node’s EventEmitter

Enable stores to both listen to and broadcast events

Allows react components to update based on those events

React components listen to stores

Stores emit changes using Node’s EventEmitter

This is how components know when an app’s state has changed

Components are redrawn when state changes

Central place where an app’s state and logic is stored.

Contain no direct setter methods

All interaction with a store must occur via callback registered with the dispatcher

A store is the only thing in the application that knows how to update data (IMPORTANT!!!!)

Most important concept in flux

A dispatcher may dispatch actions to multiple stores

Each store gets the same action

Stores need a way for react components to interact with them

Boilerplate for all stores

var AuthorStore = assign({}, EventEmitter.prototype, {

addChangeListener: function(callback) {

this.on(CHANGE\_EVENT, callback);

},

removeChangeListener: function(callback) {

this.removeListener(CHANGE\_EVENT, callback);

},

emitChange: function() {

this.emit(CHANGE\_EVENT);

}

});

// register the store to be notified of actions, and only

// act on those of interest. Remember, this gets invoked for every

// action received by the Dispatcher

Dispatcher.register(function(action) {

});

// We export AuthorStore since it's the public API. Anything else remains hidden to clients,

// like the Dispatcher.register stuff.

"src/stores/authorStore.js" 33L, 1068C written

Alternative Flux implementations build on the same boilerplate, just hiding it from clients

Stores maintain state in private storage (variables)

A common practice is for a component to register itself as a store listener on mounting the component and remove itself as a listener when the component unmounts.

Components only need to listen to store events if component isn’t unmounted / remounted as part of a transition or otherwise.

When component is unmounted / mounted, component sets initial state.

## Wait-for API

Manages the order in which stores are updated

## Controller Views

Top-level component that ends up composing other components.

Manages data flow down to all child components

They’re the only components which should interact with stores.

When a store is updated, the controller view should receive the event and pass the data down to it’s child views

Hold data as state and pass data to children via props

Passing a single object down to children makes for less maintenance as more data needs to be passed down to children

You CAN nest controller views, but it’s not recommended.

May cause multiple render calls, hurting performance

Best practice is to have a single controller view for each page, or at least for each major section of a page

## Misc

keyMirror library

Automatically copies the key in a js object to the value

keyMirror({

Foo: null ⇐ copies Foo to value here...

} )

Object-assign

Basically a way to glue two objects together

Merges properties from two objects into a single object containing the properties from the original objects

Can be used to create a new object containing the behavior / properties of multiple other objects.

Used to add eventEmitter capabilities to stores

A ponyfill that allows use of objectAssign (an ES6 construct) in ES5 browsers.

EventEmitter

Node library enabling an object to emit events

Lodash

Helper library

## FluxAPI

### Flux

register(function callback) - Store registers callbacks with the dispatcher

unregister(string id) - id is a dispatch token

waitFor(array<string> ids) - Lets you tell Flux to complete execution of the given callbacks before invoking the current callback

dispatch(object payload) - invoked on the dispatcher by an action. Tells dispatcher to tell registered stores about this action

isDispatching() - true if dispatcher is currently busy dispatching

### Dispatcher

### Actions

### Stores

Each store has the same basic interface

Extends EventEmitter so store can emit events to tell components that data has changed

addChangeListener - register component to be notified when a store changes

removeChangeListener

emitChange

## PubSub?

Not quite. Similar but different in a few ways.

Callbacks aren’t registered for particular events

Every payload is sent to every registered callback

Callbacks can be deferred until other callbacks have been invoked / other datastores have been updated.

## Async Calls

When performing async calls, it might be desired to display status immediately after a request is made and then subsequent to the response being received.

Dispatch one event when the request is submitted, another when the response is received

## Debugging

Adding debugger to code will cause browser to break automatically when it process this line

## Terms

Polyfill

Replicate an API using Javascript (or whatever) if the platform (e.g. a browser) doesn’t support it natively

Poly meaning can be solved using any number of techniques

Fill meaning it would fill the API hold in the platform

Also named after a Polyfilla spackle product meant to patch holes in walls

Fills a hole in the platform so that clients requiring feature detection will work

An example is filling a hole in String.prototype

Ponyfill

SImilar to a polyfill but not quite

Provides an API as a standalone module that can perform the missing functionality, but doesn’t actually fill the hole in the platform.

Doesn’t resolve platform detection holes.

Useful when using a polyfill would enable assumptions about the platform which really aren’t valid.

Adding a polyfill might make the client think the platform is more modern than it really is, for example.

A polyfill that doesn’t overwrite a native method

Shim

A piece of code that can fix some functionality, but has an API of its own

Shim.gif used to be used to allow expanding empty td’s

It wasn’t automatic.

It required you to manually add it to the markup.

MVC

Consists of three concepts that interact with one-another