# Immutable State Management

Using Redux with Angular to Untangle Angular Application State

Travis Stokes

# Ready for some fun...ctional state management?

## Here's What We'll Cover

### The Basics

**1** State Management

What is it? Why is it so hard? Don't patterns like MVC separate our concerns enough and resolve this issue?

**2** ng-rx/store? Redux? What the Flux?

What is ng-rx/store, and how does it relate to Redux, Flux, and all the other stuff it gets confused for?

**3** Stores, Actions, and Reducers

The basic elements of @ng-rx/store. What they do, what they look like, and how they fit into the big picture

### Extras

**4** A Little More: Side Effects and Routing

Going beyond data management and view binding

**5** Tooling

Tools? We don't need no stinkin'...actually, yea, tools are great.

### Wrapping Up

**6** Q&A

I'm here to help, so don't be afraid to ask - even before we get here!

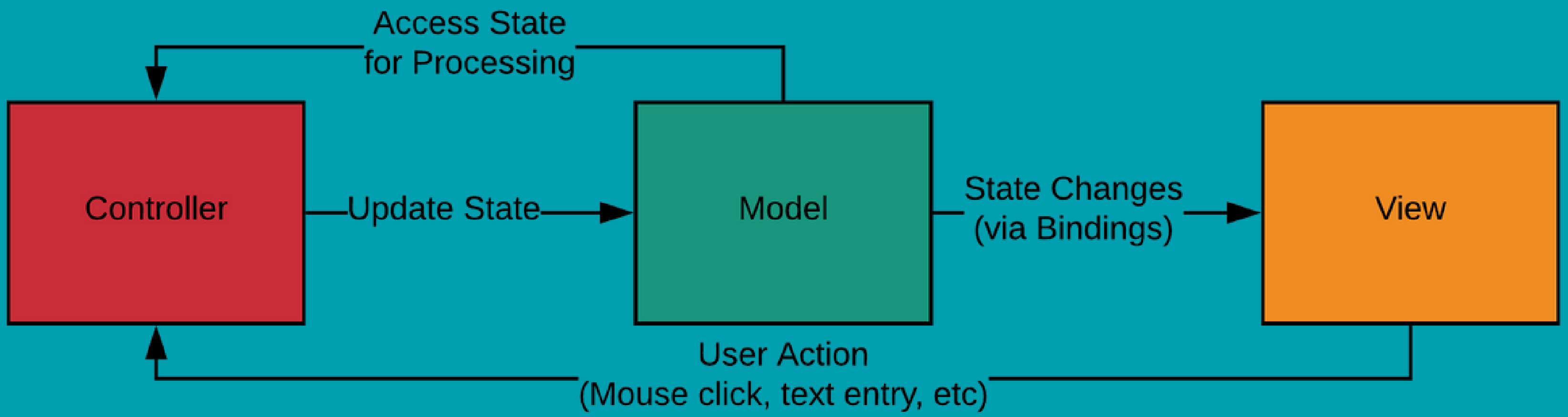
## What Is State Management?

Obligatory Boring Wikipedia Quote

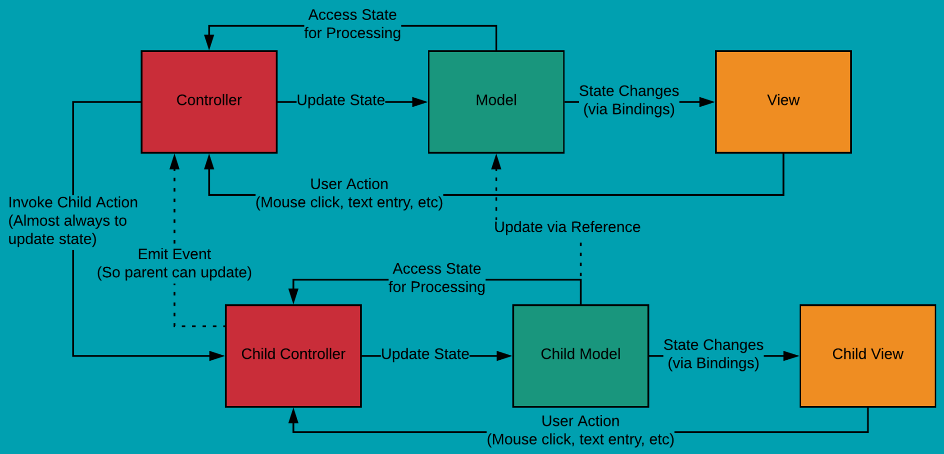
State management refers to the management of the state of [...] user interface controls [...] in a graphical user interface.

The panda is here because it's more interesting!

## In theory, MVC is great!



# It Gets Painful, Fast


Access State for Processing
Controller
Update State
Model
State Changes (via Bindings)
View
User Action (Mouse click, text entry, etc)
Invoke Child Action (Almost always to update state)
Update via Reference
Emit Event (So parent can update)
Access State for Processing
Child Controller
Update State
Child Model
State Changes (via Bindings)
Child View
User Action (Mouse click, text entry, etc)

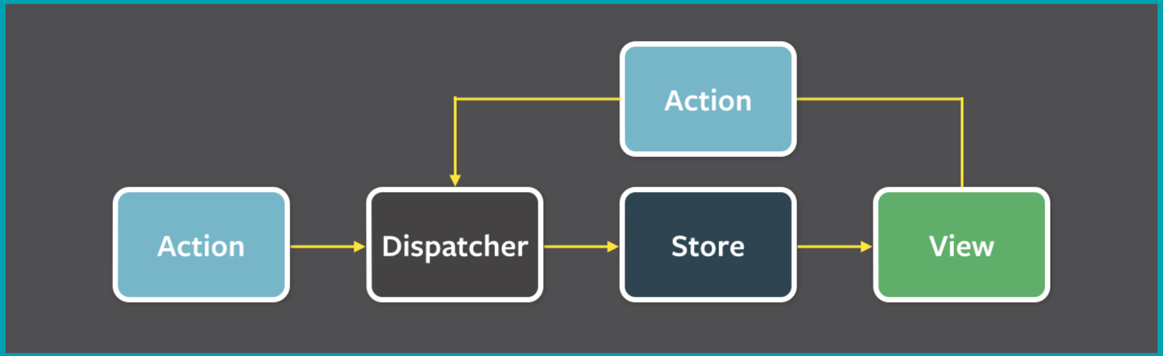
Adding components that need to share state is like feeding this guy after midnight

| Flux | Redux | @ng-rx/store |
|---|---|---|
| The basic pattern of using stores and actions to manage state | Roughly, a flux implementation built as a re-usable, standardized JS state management framework | Inspired by Redux, but built specifically for Angular utilizing RxJS to enable state flow via Observables |

## What is Flux?

- Pattern / Architectural Approach
- Designed to de-couple state management from view management
- Underlying principle is that components (views/controllers) shouldn't have to fight to keep state updated and views reactive
- “We originally set out to deal correctly with derived data: for example, [...] show an unread count for message threads while another view showed a list of threads, with the unread ones highlighted. [...] marking a single thread as read would update the thread model, and then also need to update the unread count model. These dependencies and cascading updates [... lead] to a tangled weave of data flow and unpredictable results.”*

*https://facebook.github.io/flux/docs/in-depth-overview.html#content



# Redux and ng-rx

## Redux

- Is a codified interpretation of Flux
- Originally designed with react in mind, but not bound to a specific view/rendering library
- Disregards the single dispatcher / multiple store concept for a composable store model in which the store IS the dispatcher

## ng-rx

- Builds off the redux approach
- Does not DEPEND on redux
- Utilizes RxJS to support notification of changes
- Provides bindings and implementations that enable seamless integration with Angular

# Principle Entities for Redux/ng-rx

## Actions

- Define what can be done to state
- Define the payload needed to apply the action
- Act as messages between components in your application and the store
- Typically very light weight classes to allow for Type Safe messaging

## Reducers

- Pure functions
- Take an existing state and apply actions to it
- Act as maps, defining the shape of the state managed and what changes occur for a given action
- Are the only components capable of causing “changes” to state
- Always return a NEW state object, making store data immutable and forcing pattern adherence

## Store

- Brings together reducers
- Dispatches messages (actions) sent by your application to handlers (reducers) registered with the store
- Manages notifying consumers of new data

## Side Effects

- Act as a logic/service layer
- Receive all actions
- Delegate any resulting state changes to reducers via new actions

# Tools!

## Schematics

- Scaffolding Library
- Allows easy creation of various ng-rx components

## Store Dev Tools

- Allows review of the action/reducer history
- Allows time travel
- Shows chart of reducers and current state values
-

# Helpful Links / References

- Flux
  - https://github.com/facebook/flux/tree/master/examples/flux-concepts
  - https://facebook.github.io/flux/docs/in-depth-overview.html#content
  - Flux is the pattern on which all of this is based. I encourage anybody using ng-rx Store to dig into the patterns it is based on, and these links represent the best places to start.
- Redux https://redux.js.org/introduction
  - https://redux.js.org/introduction
  - https://github.com/angular-redux/example-app
  - Redux's conceptual overviews/tutorials are a bit better than those for ng-rx, and can be used as a basis for understanding ng-rx
- ng-rx
  - https://github.com/ngrx/platform (schematics and dev tools links can be found in the readme)
  - https://goo.gl/T9XSot - StackBlitz instance for the ng-rx sample app
- This Presentation

  https://www.beautiful.ai/deck/-LH34-y5XrSsrVlj5_i2/ngrx-Store

# Travis Stokes

Owner / Oceanview Consulting
Senior Consultant / SingleStone

travis@ovitconsulting.com

http://ovitconsulting.com

@sysgineer

https://www.linkedin.com/in/travis-stokes/