Password Policy in Drupal 8

Drupal GovCon 2015

Adam Bergstein

ACQUIA®
THINK AHEAD

outline

- introduction
- password policy concepts
- getting started
- lessons learned
- d8 architecture review
- pro tips
- giving back
- my thanks

Acquia®

THINK AHEAD

intro

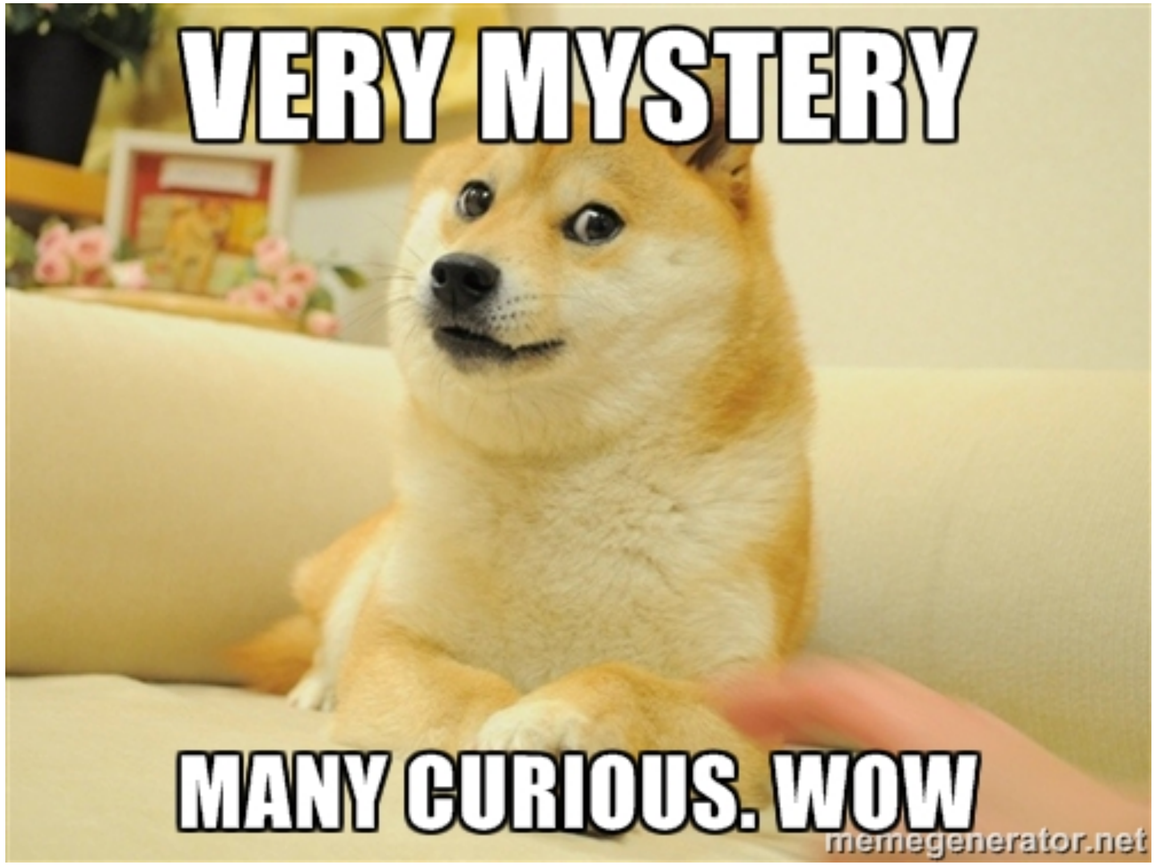
- technical architect at acquia
- using drupal since d6
- background in higher ed before coming to acquia
- interested in d8, devops, and continuous integration
- my true passion is teaching and enablement

Acquia®

THINK AHEAD

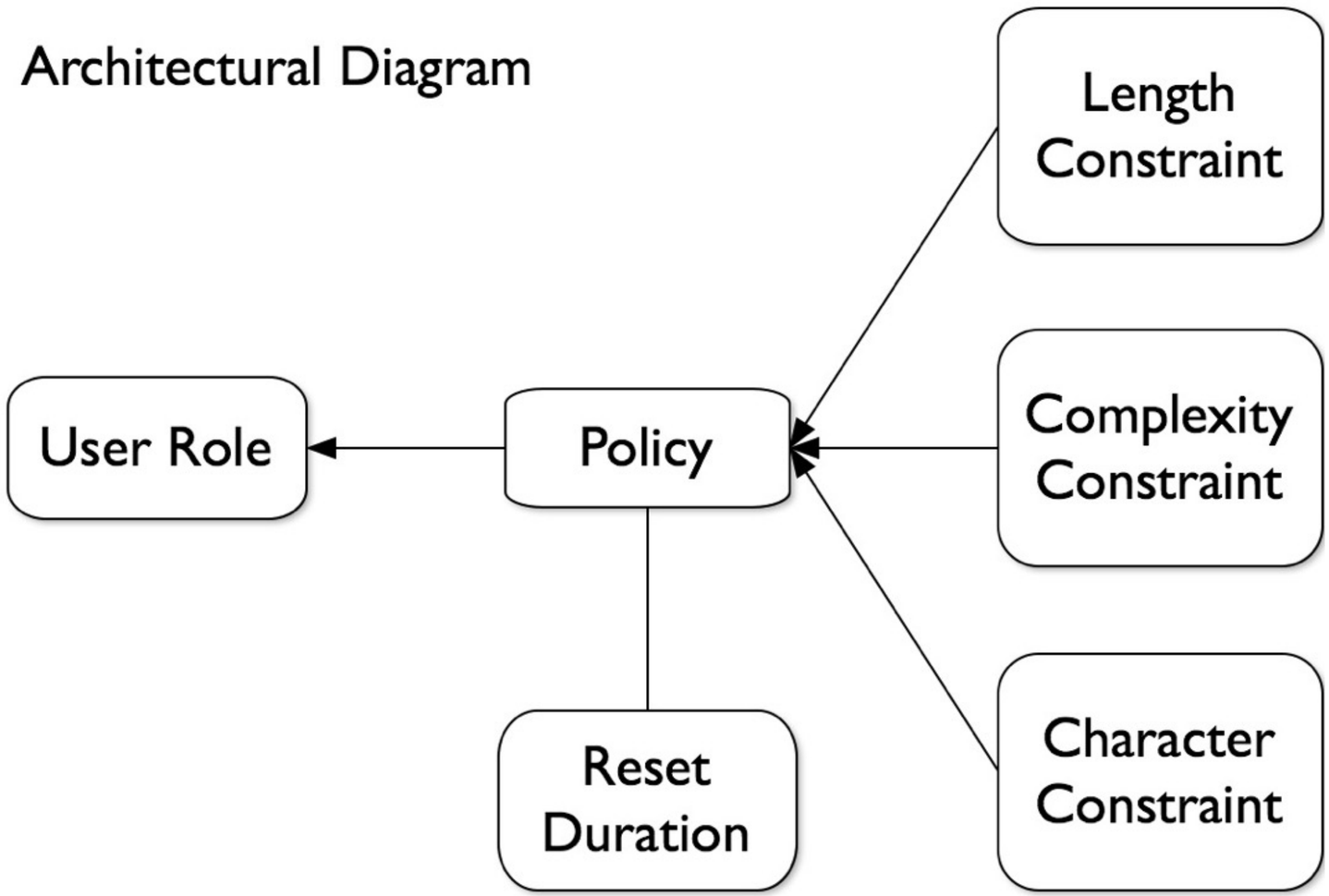
password policy module

so, what's it do?



password policy goals

- main goal is to constrain user passwords through policies and constraints
- **a policy applies to creating or updating passwords**
- a policy has one or more constraints to restrict attributes of a password
- one or more policies apply to one or more user roles
- example: a *length* policy has a *minimum character length constraint* and a *maximum character length* constraint


Architectural Diagram
Length Constraint
Complexity Constraint
Character Constraint
User Role
Policy
Reset Duration

constraint types

- a *constraint type* provides the ability to have consistency for similar constraints across policies
- character length constraint type example:

  "*X {minimum/maximum} characters*"

- X is a parameter for a specific constraint
- {minimum/maximum} can be selected per constraint

password reset

- part of a policy is to define when a user needs to reset his/her password
- this feature existed in the d7 module
- this is a *time-based constraint*
- does not apply to the password itself, maintains *user-by-user duration between password changes*

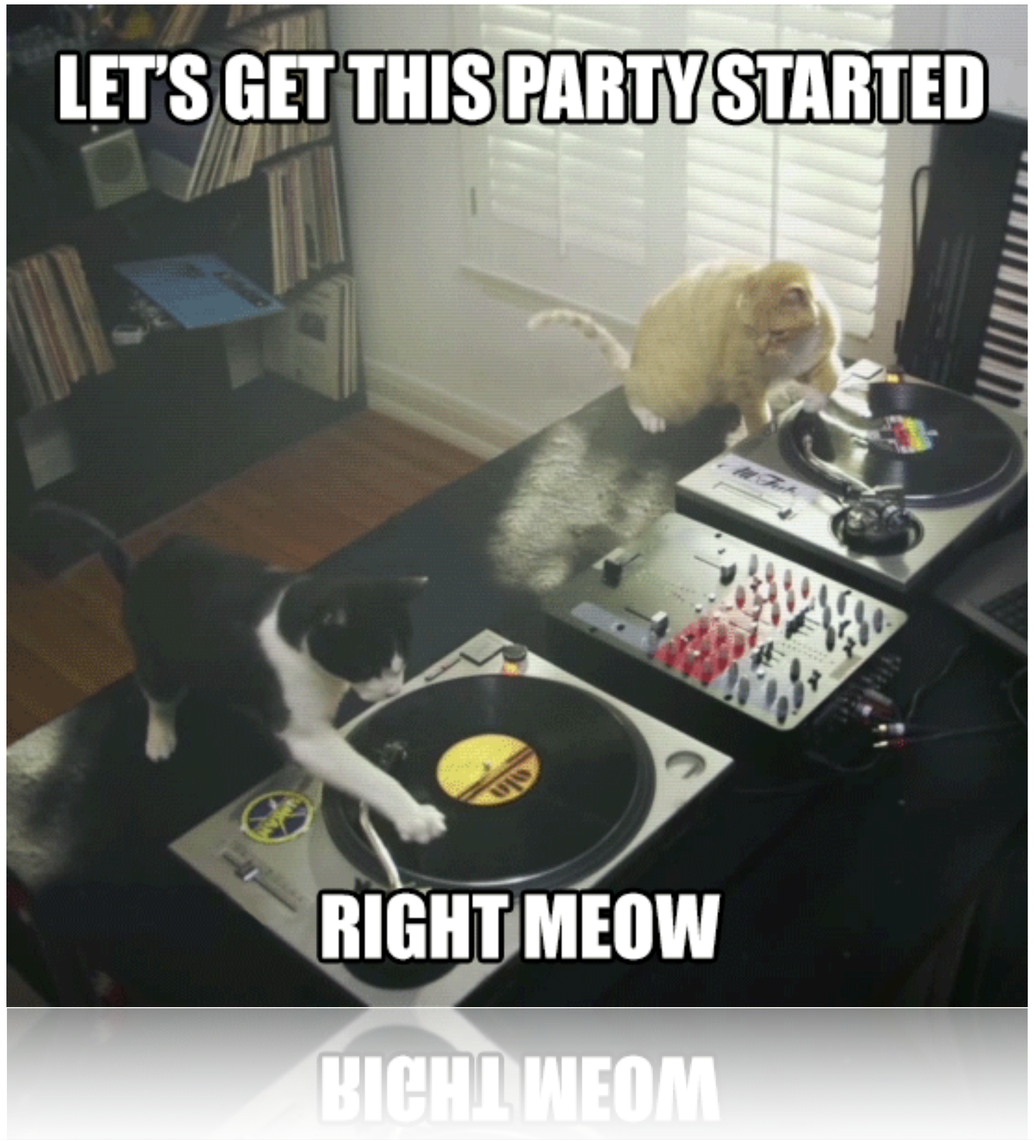
Acquia®

THINK AHEAD

getting started

starting from a d7 mindset

exploring the world of d8



module design

- **persistently store** password policies, constraints, and password reset settings
  - we will dive deeper into best practices here
- use of hooks (*form_alter*) to enforce policy constraints when passwords are created / updated
  - same as D7
- use of cron events to enforce password reset
- use of *Symphony* events to force password reset for expired user passwords

password policy plugins

- a policy is just a collection of constraints (a wrapper)
- **constraint types should be logically extensible for your specific policies**
- d8's custom *Plugin API* supports this concept perfectly
- allows for modules to **extend password policy behaviors by creating their own constraint types**
- example: *Password Policy Zxcvbn*
  - https://github.com/nerdstein/password_policy_zxcvbn
  - creates a **scoring-based constraint type** based on *Zxcvbn* library
  - specific policies select the score for the constraint

my first attempt

- too many d7 concepts
  - database driven, not entity driven
  - standard *Form API* and form alters
- incorporated a few d8 concepts
  - basic *YAML*: permissions, menu / routes, installation settings
  - many core-provided classes/interfaces (forms, plugin, tests)
  - created automated tests
  - custom plugin system and service integration for the plugins
  - leveraging *Symphony* events for user login enforcement

Acquia®

THINK AHEAD

lessons learned

take some time to reflect on how you didn’t really leverage d8 concepts

flaws with this attempt

- **its possible to use d7 concepts, but this makes for a crappy d8 module**
- d8 is object oriented, not meant to be procedural like d7
- database content is not exportable
- password reset did not properly leverage *CMI* (extending fields of *User* entity)
- the code was overly complex in architecture
- enabling a module enables *all* plugins, what if you don't want all of them?
- **most d8 ports are very lean and leverage core-provided interfaces and classes**
  - my initial port was hefty and bloated

a more appropriate d8 port

- heavy use of objects, lean use of hooks
- *Config Entities* instead of database
  - use of *Drupal Console* to generate entities
  - config schemas
  - config forms
- use of *controllers* and *services*
- plugins implemented in submodules or other modules altogether

details

- services are globally available
  - registered in module's *services.yml* file
  - *\Drupal::service('encryption')->encrypt($string);*
- ***PSR-0/4*** and **namespace concept** used within *src* directory to automagically understand the purpose of your classes
  - *src/Form/*… to maintain all of your module's form classes
  - *src/EventSubscriber/*… to associate classes with low-level Symfony events
  - *src/Tests/*… to register automated tests
- **object-oriented all the things!**
  - leverage and extend core-provided classes
  - interfaces, base classes, controllers, entities, **EVERYTHING**!
  - custom callbacks into controllers, not in module file or includes

entities are extensible

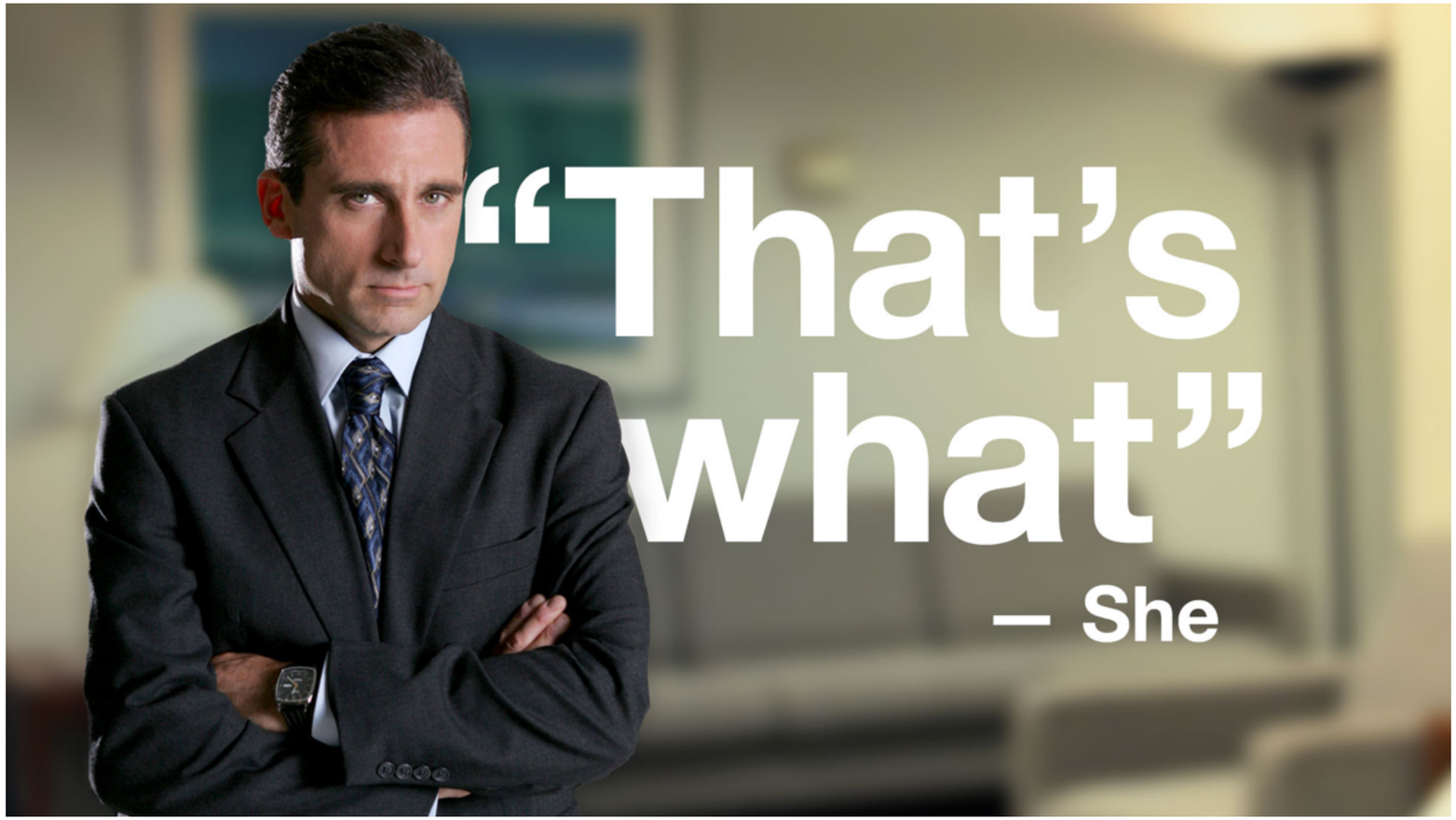
- you can create your own fields for existing entities
- **user reset attributes should not be a database table, they belong to the user entity**
- create the fields using Drupal interface
- use *CMI* to export fields as YML
- place YML into *config/install* directory of your module

Acquia®

THINK AHEAD

d8 architecture review

lets review the d8-ness



review of d8 concepts

- *ConfigEntity, ConfigForm, config/schema/[module].schema.yml*
  - CMI FTW! seriously, **consider this for persistent storage**
  - limits use of database api
  - **extend existing system entities with module-defined fields**
  - learn how to use schema-based storage - it's worth it!
- *Services* and *Plugins*
  - plugin manager, interface, base class concepts
  - Use of global *Drupal* class, containers and service invocation
- core-provided interfaces / classes perform a lot of the leg work
  - delivers a ton of out-of-the-box functionality

how do you figure out the tools in d8?

- learning the tools core provides is **really tricky**
  - d.o documentation exists for core interfaces / classes
  - some of core is still being developed and actively worked on
  - it's not always clear the purpose or the use of these tools
- learning this takes time, but **should look to core for it's use of design patterns**
  - using *PHPStorm*? hit shift key twice to perform a wildcard search for text through the project, e.g. "AJAX" will show you all files that have the text
  - after finding the class, look up on d.o for other parts of core invoking the class
  - helpful for determining potential examples or core-provided functionality

Acquia®

THINK AHEAD

pro tips

seriously…

there are ways to do this correctly

porting tips

- start fresh, don't try working directly off of d7 code
  - the best d8 modules take advantage of the new architecture
  - d8 features do not directly map to d7 features, so this requires developer intervention
- know your d8 architecture
  - plan out how you intend to use the new d8 concepts, have someone knowledgeable review it

leverage core

- try to identify design patterns set by core functionality
  - use parts of d8 system to help you find behaviors you wish to poach!
  - **review the automated tests of the module** to see how the code is used
- audit the core code that provides that behavior
  - **do a code review to identify potentially similar functionality you need**
  - familiarize yourself with core interfaces and classes you can use within your module
  - **review other classes within the same namespace** for other similar classes that may offer other additional behaviors to consider

drupal console

- **no, this is not *Drush***
  - *Drush* handles operations on a live Drupal system
  - *Console* helps you generate code for common D8 components found in modules
- major time-saver for creating D8 components used in your module
  - generate config entities
  - generate forms
  - generate services
- commands for config management (*CMI*)
  - edit, debug, override
- http://drupalconsole.com/

phpstorm

- inherent object oriented code formatting / syntax highlighting
- identifies missing or undefined class references
  - *press command click on the missing reference* to search for the class name and add the dependency
- *press shift key twice* lets you do text-based search through all of the code base
  - very helpful when trying to identify core classes and interfaces
- *press command click on a class to load the associated class contents*
- full integration of *PHPCS / Drupal Coder*
- integration with *Git* and various code versioning options

Acquia®

THINK AHEAD

giving back

we've made password policy extensible for your needs

share the love, others may benefit

how do i sign up?

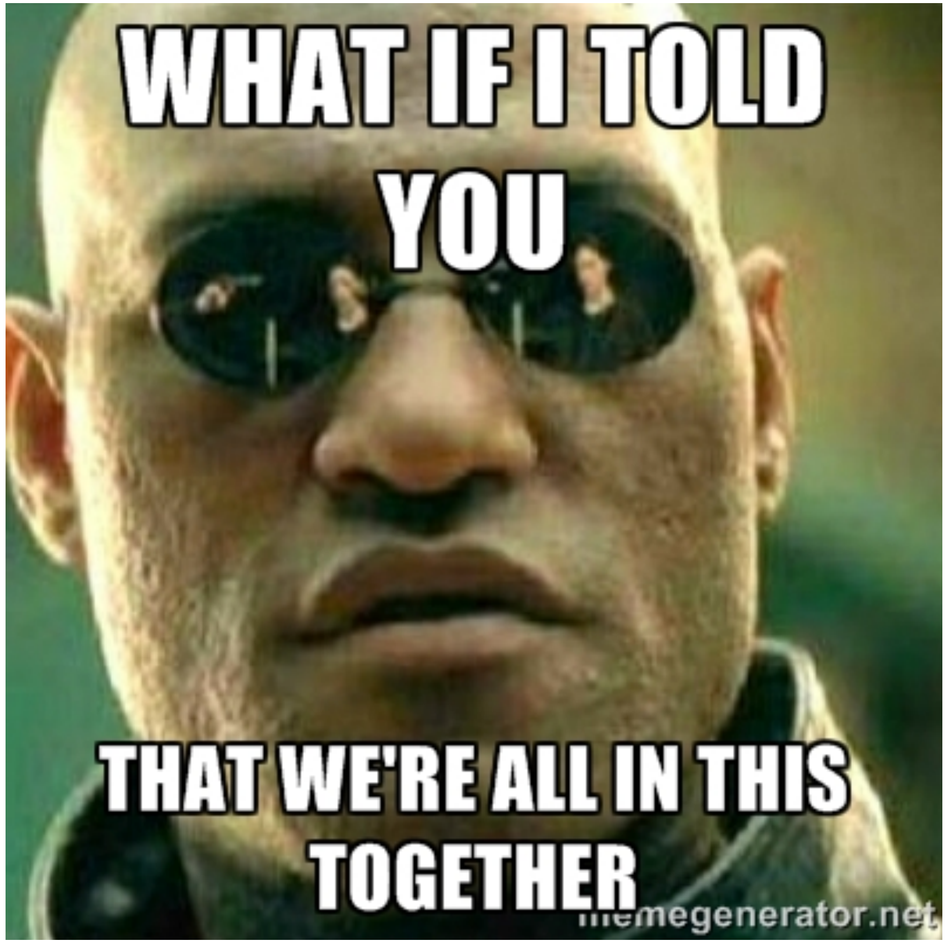
- use the module and submit issues/feedback
  - https://www.drupal.org/project/password_policy
  - http://github.com/nerdstein/password_policy
  - *8.x-3.x-dev* branch
- **contribute constraint types as new contrib modules**
  - make your own d8 module
  - leverage the password policy plugin system to build your own constraint types
  - contribute it back to the community

Acquia®

THINK AHEAD

credits

i certainly didn't figure this out alone



credits

- many folks helped me get up to speed
- Kris Vanderwater (@eclipsegc)
- Tim Plunkett (@timplunkett)
- Meagen Williams
- Aurelien Navarre
- Alex Knoll
- core developers / documentation

Acquia®

THINK AHEAD

thanks!

questions?

i'll be here through friday and attending the reception tonight

Acquia®

THINK AHEAD