Aide Documentation

Release 0.3.0

Jason Daly

Mar 29, 2017

Contents

Aide is a set of classes to assist Laravel and Silex development.

Contents:

CHAPTER 1

Authorization

Aide provides a lightweight, object-oriented authorization system based heavily on the popular Ruby gem, Pundit.

Policies

At the core of Aide's authorization is the notion of policy classes. Policy classes must extend `Deefour\Aide\Authorization\AbstractPolicy`. Each method should return a boolean. For example

```
use Deefour\Aide\Authorization\AbstractPolicy;

class ArticlePolicy extends AbstractPolicy {

  public function edit() {
    return $this->user->id === $this->record->author_id;
  }

}
```

When a policy class is instantiated through Aide, a *$record* to authorize is passed along with a *$user* to authorize against. Using the helper methods provided in `Deefour\Aide\Authorization\PolicyTrait` is optional; you could instantiate a policy and check authorization for the *$user* yourself

```
$user    = User::find(1);              // find some User with id = 1
$article = $user->articles()->first(); // get the first Article authored by the User

$policy = new ArticlePolicy($user, $article);

$policy->edit(); // true
```

Assumptions

When generating a policy class for an object via Aide's helpers, the following assumptions are made.

1. The policy class has the same name as the object being authorized, suffixed with *"Policy" (though this can be overridden)*
2. The first argument is the user to authorize for the action. When using Aide's helpers, this requires you create a *currentUser* method on the class using `Deefour\Aide\Authorization\PolicyTrait`.
3. The second argument is the object you wish to check the authorization against.

Scopes

Aide also provides support for policy scopes. A policy scope will typically generate an iterable collection of objects the current user is able to access

For example, the scope against the *Article* model below will return a collection of only **published** articles unless the current user is an administrator, in which case a collection of **all** articles will be returned.

```
use Deefour\Aide\Authorization\AbstractScope;

class ArticleScope extends AbstractScope {

  public function resolve() {
    if ($this->user->isAdmin()) {
      return $this->scope->all();
    } else {
      return $this->scope->where('published', true)->get();
    }
  }

}
```

When a policy scope is instantiated through Aide, the current *$user* and a *$scope* object are passed into the derived policy scope. By default, the policy scope is determined based on the name of the *$scope* object.

```
$user        = User::find(1);
$policyScope = new ArticleScope($user, Article::newQuery());

$articles    = $policyScope->resolve(); // collection of Articles
```

Assumptions

When generating a policy scope via Aide's helpers, the following assumptions are made.

1. The policy scope has the same name as the object being authorized, suffixed with *"Scope" (though this can be overridden)*
2. The first argument is the user to filter the scope for. When using Aide's helpers, this requires you create a *currentUser* method on the class using `Deefour\Aide\Authorization\PolicyTrait`.
3. The second argument is the scope object you wish to modify based on the state/details of the *$user*.

An Example in the Context of an Application

Using Laravel, the following could be added to the *BaseController*.

```
class BaseController extends Controller {

  use Deefour\Aide\Authorization\PolicyTrait;

  protected function currentUser() {
    return Auth::user() ?: new User;
  }

}
```

Now, for some *ArticleController*, to authorize the current user against the ability to edit a specific *Article*, the *edit()* method would like this

```
public function edit($id) {
  $article = Article::find($id);

  $this->authorize($article); // if NOT authorized, exception will be thrown

  return View::make('articles.edit'); // display the form
}
```

The *$this->authorize($article);* line will generate a fresh *ArticlePolicy* instance through Aide, passing the current user and the fetched *$article* into it. The *ArticlePolicy::edit()* method will be called, and if the user is authorized to edit the article, the view for the action will render as expected.

Usage Within Laravel

Aide provides a service provider and facade for the *Policy* class to make interacting with it very simple inside of a Laravel application.

Service Provider

In Laravel's *app/config/app.php* file, add the class:*Deefour\Aide\Authorization\PolicyServiceProvider* class to the list of providers

```
'providers' => array(

   // ...

   'Deefour\Aide\Support\Facades\PolicyServiceProvider',

),

// ...
```

The IoC container is responsible for instantiating a single, shared instance of the `Deefour\Aide\Authorization\Policy` class. This is done outside the scope of a controller method, meaning the IoC container has no access to or knowledge of the *currentUser* method that may exist within a base controller. Further, the API provided by the *Policy* facade does not expect a user to be passed. This means instead of calling

```
$app['policy']->policy(Auth::user(), new Article);
```

the following is actually correct

```
$app['policy']->policy(new Article);
```

To accomplish this, the service provider looks for configuration in an *app/config/policy.php* file. At a minimum, the following is required when using the policy service provider.

```
<?php

return array(

  'user' => function() {

    return Auth::user() ?: new User; // this logic can be replaced with anything you like.

  },

);
```

To keep things DRY, the *currentUser* method in the base controller could be modified to take advantage of this same Closure.

```
public function currentUser() {
  return call_user_func(Config::get('policy.user'));
}
```

Policy Facade

With the service provider in place, the instantiated policy class can be accessed via the main application container.

```
$articlePolicy = $app['policy']->policy(new Article);
```

Remember, you can also instantiate the policy provider yourself.

```
use Deefour\Aide\Authorization\Policy;

$policyProvider = new Policy(Auth::user());
$articlePolicy  = $policyProvider->policy(new Article);
```

This can be simplified through the use of a *Policy* facade. Add the following to *app/config/app.php*

```
'aliases' => array(

   // ...

   'Policy' => 'Deefour\Aide\Support\Facades\Policy',

),

// ...
```

The same functionality above is now as simple as this

```
$articlePolicy = Policy::policy(new Article);
```

Policies in Views

The facade makes working with policies in views simple too. For example, to conditionally show an 'Edit' link for a specific *$article* based on the current user's ability to edit that article, the following could be used in a blade template

```
@if (Policy::policy($article)->edit())
  <a href="{{ URL::route('articles.edit', [ 'id' => $article->id ]) }}">Edit</a>
@endif
```

Handling Unauthorized Exceptions

If *false* is returned by the *authorize()* call, a `Deefour\Aide\Authorization\NotAuthorizedException` will be thrown. This exception can be caught by Laravel with the following in *app/start/global.php*.

```
use Deefour\Aide\Authorization\NotAuthorizedException;

App::error(function(NotAuthorizedException $exception) {
  // Handle the exception...
});
```

Note: There is nothing Laravel-specific about Aide's authorization component. The `Deefour\Aide\Authorization\PolicyTrait` trait can be used in any class.

Ensuring Policies Are Used

Again using Laravel as an example, an after filter can be configured to prevent accidentally unauthorized actions from being wide open by default. A filter in the constructor of the *ArticleController* could look like this

```
public function __construct() {
  $this->afterFilter(function() {
    $this->verifyAuthorized();
  }, [ 'except' => 'index' ]);
}
```

There is a similar method to ensure a scope is used, which is particularly useful for *index* actions where a collection of objects is rendered and is dependent on each user.

```
public function __construct() {
  $this->afterFilter(function() {
    $this->requirePolicyScoped();
  }, [ 'only' => 'index' ]);
}
```

Policy/Scope Instantiation Without Trait Methods

Policies and scopes can easily be retrieved using static or instance methods on the `Deefour\Aide\Authorization\Policy` class.

Static Instantiation

The following methods are statically exposed:

- *Policy::policy()*
- *Policy::policyOrFail()*
- *Policy::scope()*
- *Policy::scopeOrFail()*

For example:

```
use Deefour\Aide\Authorization\Policy;

$user    = User::find(1);
$article = $user->articles()->first();

Policy::policy($user, $article);
Policy::policyOrFail($user, $article);

Policy::scope($user, new Article);
Policy::scopeOrFail($user, new Article);
```

The *...OrFail* version of each method will throw a `Deefour\Aide\Authorization\NotDefinedException` exception if the policy class Aide tries to instantiate doesn't exist.

Instance Instantiation

A limited version of the above API is available when creating an instance of the *Policy* class.

- *Policy::policy()*
- *Policy::scope()*
- *Policy::authorize()*

```
use Deefour\Aide\Authorization\Policy;

$user    = User::find(1);
$article = $user->articles()->first();
$policy  = new Policy($user);

$policy->policy($article);

$policy->scope($article);

$policy->authorize($article, 'edit');
```

Note: The authorize method in this case **requires** an action/method be passed as the second argument.

The *policy()* and *scope()* methods are pass-through's to the *...OrFail()* methods on the *PolicyTrait*; exceptions will be thrown if a policy or scope cannot be found.

Manually Specifying Policy Classes

The policy class Aide tries to instantiate for an object can be overridden. Given the following scenario

```
use Deefour\Aide\Authorization\Policy;

class ArticlePolicy {}

class Article { }
class NewsArticle extends Article { }

Policy::policyOrFail(new Article);     // returns fresh ArticlePolicy instance
Policy::policyOrFail(new NewsArticle); // throws␣
↪Deefour\Aide\Authorization\NotDefinedException
```

Aide can be instructed to instantiate an `ArticlePolicy` class for the `NewsArticle` through a *policyClass()* method on `Article` *(since :class:'NewsArticle' extends it)*.

```
use Deefour\Aide\Authorization\Policy;

class ArticlePolicy {}

class Article {

  public function policyClass() {
    return 'ArticlePolicy';
  }

}
class NewsArticle extends Article { }

Policy::policyOrFail(new Article);     // returns fresh ArticlePolicy instance
Policy::policyOrFail(new NewsArticle); // returns fresh ArticlePolicy instance
```

Similarly, if a *name()* method is provided on the object, the string returned will be used as the class prefix for the policy class Aide tries to instantiate.

```
use Deefour\Aide\Authorization\Policy;

class PostPolicy {}

class Article {

  public function name() {
    return 'Post';
  }

}

Policy::policyOrFail(new Article); // returns fresh PostPolicy instance
```

Closed System

Many apps only allow authenticated users to perform most actions. Instead of verifying on every policy action that the current user is not *null*, unpersisted in the database, or similarly not a legitimately authenticated user, this can be done

through a special `ApplicationPolicy` that your other policy classes extend.

```
use Deefour\Aide\Authorization\AbstractPolicy;
use Deefour\Aide\Authorization\NotAuthorizedException;

class ApplicationPolicy extends AbstractPolicy {

  public function __construct($user, $record) {
    if (is_null($user) or ! $user->exists) {
      throw new NotAuthorizedException;
    }

    parent::__construct($user, $record);
  }

}

class ArticlePolicy extends ApplicationPolicy { }
```

Mass Assignment Protection

A special *permittedAttributes* method can be created on a policy to conditionally provide a whitelist of attributes for a given request by a user to create or modify a record.

```
use Deefour\Aide\Authorization\AbstractPolicy;

class ArticlePolicy extends AbstractPolicy {

  public function permittedAttributes() {
    $attributes = [ 'title', 'body', ];

    // prevent the author and slug from being modified after the article
    // has been persisted to the database.
    if ( ! $this->record->exists) {
      return array_merge($attributes, [ 'user_id', 'slug', ]);
    }

    return $attributes;
  }

}
```

This policy method can be used within a controller to filter unauthorized attributes from being set on a model via mass assignment. Again, in a Laravel controller action *('Repository' below comes from a facade provided by Aide for Laravel)*

```
$article    = Article::find(1);
$repository = Repository::make($article);
$policy     = Policy::make($article);

$repository->update(
  $policy->permittedAttributes(Input::get('article'))
);
```

CHAPTER 2

Validation

Aide provides an abstraction for validating entities using 3rd party validation libraries. Out of the box Aide supports Laravel's validator from the illuminate/validation package. The example below will use the `Deefour\Aide\Validation\IlluminateValidator`.

Instantiation

To validate an entity, a new instance of the validator must be created. This typically will be done within a service provider.

```
use Illuminate\Validation\Factory;
use Symfony\Component\Translation\Translator;
use Deefour\Aide\Validation\IlluminateValidator as Validator;
use Symfony\Component\Translation\MessageSelector;

$translator = new Translator('en', new MessageSelector);
$validator  = new Factory($translator);

$validator  = new Validator($factory);
```

Usage

With this new `$validator` instance, any class that extends `Deefour\Aide\Validation\ValidatableInterface` can be validated easily. For example, given the following `User` entity

```
// AbstractEntity implements the Deefour\Aide\Validation\ValidatableInterface
use Deefour\Aide\Persistence\Entity\AbstractEntity;

class Article extends AbstractEntity {

  // attributes
```

```
  public $title;
  public $body;

  // validation rules
  public function validations() {
    return [
      'title' => 'required',
      'body'  => 'required',
    ];
  }

}
```

validation on a `Article` instance could be done as follows

```
$entity = new Article([ 'title' => 'A Great Title', 'body' => 'Lots of text...' ]);

$validator->setEntity($entity);

$validator->isValid(); // boolean whether the entity passes validation rules or not

$validator->errors(); // array, keyed by attribute names, with array values␣
↪containing list of errors for each attribute
```

The raw validation instance behind the abstraction Aide provides can also be accessed.

```
$validator->getValidator(); // returns the \Illuminate\Validation\Factory instance
```

Validation Rules

Part of the `Deefour\Aide\Validation\ValidatableInterface` contract is the following

```
/**
 * List of rules to use in the validation abstraction layer to ensure all required
 * information has been provided in the expected format.
 *
 * @param  array  $context  [optional]
 * @return array
 */
public function validations(array $context = []);
```

Note: This is a strict requirement. The `Deefour\Aide\Persistence\Entity\AbstractEntity` class that all entity classes are to extend defines an implementation of this `validations()` method that will throw a `\BadMethodCallException` in an attempt to prevent the developer from forgetting to set up proper validation rules.

The `User` entity Aide provides contains a simple set of default rules.

```
public function validations(array $context = []) {
  return [
    'first_name' => [ 'required', 'between:3,30' ],
    'last_name'  => [ 'required', 'between:3,30' ],
    'email'      => [ 'required', 'email' ],
```

```
  ];
}
```

The keys match attribute names. The values are arrays of strings matching the format Laravel's validator expects. See the basic usage for Laravel's Validator to learn more about the above syntax.

Context

When there is a need to validate against external data, configuration, etc..., a special context can be built up on the validator. The context is passed into every `validations()` method, and as the 2nd argument to all Closure validation rules.

With the context being passed into the `validations()` method, rules can be conditionally set.

First, set the entity and context on the validator

```
$user = new User([ 'first_name' => 'Jason', 'email' => 'jason@deefour.me' ]);

$validator->setEntity($user)
          ->setContext([ 'last_name_max' => 20 ]);
```

Then refer to the context and make the validation rule dependent on it's value.

```
public function validations(array $context = []) {
  $lastNameMax = array_key_exists('last_name_max', $context) ? $context['last_name_max
↪'] : 30;

  return [
    'first_name' => [ 'required', 'between:3,30' ],
    'last_name'  => [ 'required', 'between:3,' . $lastNameMax ],
    'email'      => [ 'required', 'email' ],
  ];
}
```

Rule Callbacks

There are times where more complex validation is required for a rule. PHP Closures can be appended to the rules. The same context is passed to each Closure rule too.

> **Note:** Both within the `validations()` method itself and the Closure rules, `$this` can be used to access attributes or other methods on the entity instance.

For example, to do a dns lookup against the domain used for the email address on the `User` entity above, the example could be expanded as follows

```
public function validations(array $context = []) {
  $rules = [
    'first_name'  => [ 'required', 'between:3,30' ],
    'last_name'   => [ 'required', 'between:3,30' ],
    'email'       => [ 'required', 'email' ],
  ];

  $rules['dns-lookup'] = function() {
    $email  = $this->email;
    $domain = substr($email, mb_strpos($email, '@'));
```

```
    if (dns_get_record($domain) === false) {
      return 'invalid-hostname';
    }
  };

  return $rules;
}
```

The validation Closure will be considered failing if a string is returned. The returned string should match a key for a message template. The Closure rules are not keyed in the validation rules do not have ot be keyed by a specific attribute on the entity. It is important the developer be aware of this, Because the string `'dns-lookup'` does not match any attributes on the entity

Message Templates

The base `Deefour\Aide\Validation\AbstractValidator` instance has a currently-very-limited-but-growing set of error message templates.

```
protected $messageTemplates = array(
  'required'       => '%s is required',
  'email'          => '%s must be a valid email address',
  'date'           => '%s is not a valid date',
  'digits_between' => '%s is out of bounds',
);
```

The collection of error messages returned when calling `$validator->errors()` is composed of message templates like those above after having their `sprintf` tokens replaced by data from the validator. This token replacement currently does not leverage translation/localization or other sophisticated message replacement strategies. The single `%s` is replaced with the attribute name related to each error message. An attributes name like `first_name` will be transformed into `first name` by removing the snake case.

Entity Message Templates

Any entity can define it's own additional message templates. Since there is no default `'invalid-hostname'` message template defined, it can be defined directly on the `User` entity.

```
protected $messageTemplates = array(
  'invalid-hostname' => '%s contains an invalid/unknown domain',
);
```

An Example

Let's look at a full example within the context of a Laravel controller action.

```
public function update($id) {
  $user      = User::find($id)->toEntity(); // toEntity() is an Aide method
  $input     = Input::get('user');
  $validator = $this->validator;

  $errors = $validator->setEntity($user)->errors();

  if ( ! empty($errors)) {
```

```
    // error: invalid data
    return View::make('user.edit', compact('user', 'input', 'errors'));
  }

  // success
  return Redirect::to('home');
}
```

CHAPTER 3

License