flask-allows Documentation

Release 0.7.0

Alec Nikolas Reiter

Sep 30, 2018

Contents

Version 0.7.0 (*Change log*)

Flask-Allows gives you the ability to impose identity requirements on routes in your Flask application:

```
from flask import Flask
from flask_allows import Allows, Requirement
from flask_login import current_user

app = Flask(__name__)
allows = Allows(app=app, identity_loader=lambda: current_user)

class IsAdmin(Requirement):
    def requires(self, user):
        return user.permissions.get('admin')

@app.route('/admin')
@allows.requires(IsAdmin())
def admin_index():
    return "Welcome to the super secret club"
```

CHAPTER 1

Installation

Flask-Allows is available on pypi and installable with:

```
pip install flask-allows
```

Flask Allows supports 2.7, and 3.4+. Support for 3.3 was ended in the version 0.3 release.

Note: If you are installing `flask-allows` outside of a virtual environment, consider installing it with `pip install --user flask-allows` rather than using sudo or adminstrator privileges to avoid installing it into your system Python.

CHAPTER 2

Content

2.1 Quickstart

This guide will walk you through the basics of creating and using requirements with Flask-Allows.

2.1.1 Setting Up Allows

Before we can set up requirements and guard routes, we must first setup the extension object:

```
from flask_allows import Allows
from flask import Flask

app = Flask(__name__)

allows = Allows(app)
```

This is all that is needed to register the extension object against a Flask application, however we must also register a way to load a user object to check against requirements. Managing user sessions is outside the scope of Flask-Allows, however for the sake of this tutorial we'll assume that you are using `flask_login`:

```
from flask_login import current_user

allows.identity_loader(lambda: current_user)
```

An identity_loader can also being provided at object instantiation:

```
Allows(identity_loader=lambda: current_user)
```

2.1.2 Creating Requirements

Now that we have a configured *Allows* instance registered against the application, let's create some custom requirements. Requirements can either be a child of *Requirement*:

```
class HasPermission(Requirement):
    def __init__(self, permission):
        self.permission = permission

    def fulfill(self, user):
        return self.permission in user.permissions
```

or a function that accepts the current identity:

```
def is_admin(user):
    return 'admin' in user.groups
```

Note: Until version 0.5.0 requirements, both class and function based, needed to accept both an identity and the current request. This has been deprecated in favor of accepting only the identity and will be removed in version 1.0.0.

These classes and functions can be as simple or as complicated as needed to enforce a particular requirement to guard a route or action inside your application.

2.1.3 Guarding Routes

In order to guard route handlers, two decorators are provided:

- The *requires()* method on the configured Allows instance
- The standalone *requires()*

Both accept the same arguments the only difference is where each is imported from. We'll use the standalone decorator in this tutorial.

Applying a requirement is done by passing the desired requirements into the decorator:

```
from flask_allows import requires

@app.route('/admin')
@requires(is_admin)
def admin_section():
    return render_template('admin_section.html')
```

In order to pass a class based requirement into the decorators, you must pass an instantiated object rather than the class itself:

```
@app.route('/admin')
@requires(HasPermission('view_admin_panel')
def admin_section():
    return render_template('admin_section.html')
```

Danger: If you're using `requires` to guard route handlers, the `route` decorator must be applied at the top of the decorator stack (visually first, logically last):

```
@app.route('/')
@requires(SomeRequirement())
def index():
    pass
```

If the `requires` decorator comes after the `route` decorator, then the unguarded function is registered into the application:

```
@requires(SomeRequirement())
@app.route('/')
def index():
    pass
```

This invocation registers the actual `index` function into the routing map and then decorates the index function.

To apply either of these decorators to class based views, there are two options:

1. Supply it in the `decorators` class attribute of the view. In the case of `MethodView` this will guard every action handler:

```
class SomeView(View):
    decorators = [requires(is_admin)]
```

2. Apply it directly to an action handler, such as with `MethodView`. In the following example, only the `post` method will be guarded:

```
class SomeView(MethodView):
    def get(self):
        return render_template('some_template.html')

    @requires(is_admin)
    def post(self):
        return render_template('some_tempalte.html')
```

2.1.4 Guarding entire application or blueprint

If you find yourself applying the same set of requirements to every route in an applicationr or blueprint, you can instead guard all routes with that set of requirements instead using *`guard_entire()`*:

```
from flask import Blueprint
from flask_allows import guard_entire

from myapp.requirements import is_admin


admin_area = Blueprint(__name__, "admin_area")
admin_area.before_request(guard_entire([is_admin]))
```

You may also specify what happens when the requirements aren't met by providing either of `throws` or `on_fail`. If `on_fails` returns a non-None value, that will be treated as the result of the routing request:

```
from flask import flash, redirect

def flash_and_redirect(message, level, endpoint):
    def flasher(*a, **k):
        flash(message, level)
        return redirect(endpoint)
    return _

admin_area.before_request(
```

(continues on next page)

(continued from previous page)

```
    guard_entire(
        [is_admin],
        on_fail=flash_and_redirect(
            message="Must be admin",
            level="danger",
            endpoint="index"
        )
    )
)
```

`guard_entire` will pass `flask.request.view_args` as keyword arguments to the `on_fail` handler registered with it, this is useful is the blueprint is registered with a dynamic component such as a username:

```
def flash_formatted_message(message, level):
    def flasher(*a, **k):
        flash(message.format(**k), level)
    return flasher

user_area = Blueprint(__name__, "users")
user_area.before_request(
    guard_entire(
        [MustBeLoggedIn()],
        on_fail=flash_formatted_message(
            "Must be logged in to view {}'s profile",
            level="warning"
        )
    )
)
```

If you need to exempt a route handler inside the blueprint from these permissions, that is possible as well by using *`exempt_from_requirements()`*:

```
@admin_area.route('/unpermissioned')
@exempt_from_requirements
def index():
    ...
```

Danger: `exempt_from_requirements` only prevents ambient runners like `guard_entire` from acting on the route handler. However, if `requires`, `allows.requires` or another runner acts on the route then those requirements will be run.

Warning: `guard_entire` will only run if there is not a routing exception, such as a 404. This is to prevent unhandled exceptions from attempting to matching against `request.endpoint` when it is not populated.

2.2 Requirements

Requirements are how routes and code paths are guarded with Flask-Allows, they are also entirely defined by the user. Requirements come in two flavors:

- functions

- class based

2.2.1 Function Requirements

Function requirements need to accept the current identity and return a boolean representing if the requirement has been fulfilled or not:

```
def user_is_admin(user):
    return user.level == 'admin'
```

Note: Optionally, the function may accept the request argument, though this behavior is deprecated as version 0.5.0 and will be removed in 1.0.0:

```
def user_is_admin(user, request):
    return user.level == 'admin'
```

This function can be provided to any of the requirement runners, if you wanted to guard a route with it:

```
@app.route('/admin')
@requires(user_is_admin)
def admin():
    return render_template('admin.html')
```

Or guard a particular code path with *Permission*:

```
p = Permission(user_is_admin)
if p:
    print("Welcome!")
```

2.2.2 Class Based Requirements

Class based requirements are good if you have something to represent that is too complicated for a function. While it is possible to implement class based requirements by adding a `__call__()`, there is the *Requirement* that provides the `fulfill` hook to implement. It also provides future proofing if new hooks are also implemented. For example:

```
class Has(Requirement):
    def __init__(self, permission):
        self.permission = permission

    def fulfill(self, user):
        return self.permission in user.permissions
```

Note: Optionally, the fulfill method may accept the request argument, though this behavior is deprecated as version 0.5.0 and will be removed in 1.0.0:

```
class Has(Requirement):
    def __init__(self, permission):
        self.permission = permission

    def fulfill(self, user, request):
        return self.permission in user.permissions
```

To apply this to a route:

```
@app.allows('/admin')
@requires(Has('admin'))
def admin():
    return render_template('admin.html')
```

Danger: If you use class based requirements, you are responsible for instantiating them and providing any necessary arguments to them. A common mistake is providing just the class itself to the requirement runner:

```
@app.allows('/admin')
@requires(Has)
def admin():
    return render_template('admin.html')
```

This will result in an exception being raised at verification because the identity and request objects are passed into a constructor that only expected one argument. If the constructor expected two arguments, there is a chance that the requirement would incorrectly pass as object default to True when expressed as booleans.

While using `Requirement` isn't strictly necessary, it is provided for people that prefer an object oriented approach instead.

2.2.3 Combining Requirements

In addition to the *`Requirement`* base class, Flask-Allows also provides a way to combine requirements.

All requirement runners provided by Flask-Allows accept multiple requirements and all must pass for the verification to pass:

```
@app.route('/admin')
@requires(user_is_logged_in, user_is_admin)
def admin():
    return render_template('admin.html')
```

If either requirement returns False, then the user will not be allowed to access that route. However, if you have a more complicated requirement, such as a user must be logged in AND a user must be an admin OR a user must have the `'view_admin_panel'` permission, these can be difficult to express in an all-or-nothing evaluation strategy.

To handle these situations, Flask-Allows exposes several helper requirements:

```
from flask_allows import And, Or


@app.route('/admin')
@requires(And(user_is_logged_in, Or(user_is_admin, Has('view_admin_panel'))))
def admin():
    return render_template('admin.html')
```

Strictly speaking, the outer `And` isn't necessary as the requirements will already be combined in an `and` fashion but is presented for example sake. The `And` help is most useful when nested inside of an `Or` such as:

```
Or(user_is_admin, And(Has('view_admin_panel'), user_is_moderator))
```

Flask-Allows also exposes a helper to invert the result of a requirement:

```
@app.route('/login')
@requires(Not(user_is_logged_in))
def login():
    return render_template('login.html')
```

Finally, Flask-Allows also exposes a generalized version of these helpers called *ConditionalRequirement* (also importable as `C` to avoid typing out the name every time).

By using `ConditionalRequirement` you can build your own requirements combinator. In addition to the requirements themselves, `ConditionalRequirement` will also accept:

- `op` a binary operator to reduce results with
- `negated` if the opposite of the result should be returned (e.g. False turns into True)
- `until` a boolean value to short circuit on and end evaluation

For example, if you needed your requirements combined with xor, that is possible:

```
from operator import xor

C(perm_1, perm_2, op=xor)
```

Finally, `ConditionalRequirement` also provides the magic methods for:

- `&` short cut to applying `And` between two instances of `ConditionalRequirement`
- `|` short cut to applying `Or` between two instances of `ConditionalRequirement`
- `~` (invert) short to negating a single instance of `ConditionalRequirement`

Using these operators, our earlier combined and negated requirements would look like:

```
C(user_is_logged_in) & (C(user_is_admin) | C(Has('view_admin_panel')))
~C(user_is_logged_in)
```

However, using the named helper methods are often clearer and more efficient.

2.2.4 Transition to User Only Requirements

As of version 0.5, passing the request object directly into a requirement is deprecated and will be removed in version 1.0. Considering the following requirement:

```
from flask_allows import Requirement

class AllowedToViewPost(Requirement):
    def fulfill(self, user, request):
        post_id = request.view_args.get('post_id')
        if post_id is None:
            abort(404)

        if post.hidden:
            return 'view_hidden_post' in user.permissions

        return True
```

In order to make the transition to a user only requirement, the only change to make is:

```
from flask import request
from flask_allows import Requirement

class AllowedToViewPost(Requirement):
    def fulfill(self, user):
        post_id = request.view_args.get('post_id')
        if post_id is None:
            abort(404)

        if post.hidden:
            return 'view_hidden_post' in user.permissions

        return True
```

To be clear, `request` is now being imported directly from the `flask` package. This is the same request object that Allows would pass into the requirement itself. And the other change is removing the request parameter from the `fulfill` definition.

Behind the scenes, Allows handles both definitions and will dispatch between them as needed.

Danger: If you have a requirement defined with an optional request, such as:

```
def allowed_to_view_post(user, request=None):
    ...
```

Allows will incorrectly determine that you have provided a user only requirement.

If your requirement does not need the request object, the only change to make is to remove the parameter. If your requirement does need the request object you may either remove the default value and Allows will determine that you have provided a user-request requirement, or you may remove the parameter altogether and import `request` directly from Flask.

Additionally, there is *wants_request()* which marks the requirement as user only but passes the current request behind the scenes. This decorator is intended only to assist during a transitionary phase and will be remove in flask-allows 1.0

2.3 Helpers

In addition to the *Allows*, there are several helper classes and functions available.

2.3.1 Permission

Permission enables checking permissions as a boolean or controlling access to sections of code with a context manager. To construct a Permission, provide it with a collection of requirements to enforce and optionally any combination of:

- `on_fail` callback
- An exception type or instance with `throws`
- A specific identity to check against with `identity`

Note: Using `Permission` as a boolean or as a context manager requires an active application context.

Once configured, the Permission object can be used as if it were a boolean:

```
p = Permission(SomeRequirement())

if p:
    print("Passed!")
else:
    print("Failed!")
```

When using Permission as a boolean, only the requirement checks are run but no failure handling is run as not entering the conditional block is considered handling the failure. Not running the failure handling on a failed conditional check also helps cut down on unexpected side effects.

If you'd like the failure handlers to be run, Permission can also be used as a context manager:

```
p = Permission(SomeRequirement())

with p:
    print("Passed!")
```

When used as a context manager, if the requirements provided are not met then the registered `on_fail` callback is run and the registered exception type is raised.

Note: Permission ignores the result of the callback when used as a context manager so the exception type is always raised unless the callback raises an exception instead.

2.3.2 requires

If you're using factory methods to create your Flask application and extensions, it's often difficult to get ahold of a reference to the allows object. Because of this, the `requires()` helper exists as well. This is a function that calls the configured allows object when the wrapped function is invoked:

```
@requires(SomeRequirement())
def random():
    return 4
```

Danger: If you're using `requires` to guard route handlers, the `route` decorator must be applied at the top of the decorator stack (visually first, logically last):

```
@app.route('/')
@requires(SomeRequirement())
def index():
    pass
```

If the `requires` decorator comes after the `route` decorator, then the unguarded function is registered into the application:

```
@requires(SomeRequirement())
@app.route('/')
def index():
    pass
```

This invocation registers the actual `index` function into the routing map and then decorates the index function.

The `requires` decorator can also be applied to class based views by either adding it to the `decorators` property:

```
class SomeView(View):
    decorators = [requires(SomeRequirement())]
```

When passed into the decorators property, it will guard the entire view and in the case of `MethodView` apply to every action handler on the view.

You may also apply the decorator to individual methods:

```
class SomeView(MethodView):

    @requires(SomeRequirement())
    def get(self):
        return render_template('a_template.html')
```

In this instance, only the the `get` method of the view will be guarded but all other action handlers will not be.

2.4 Manipulating Requirements after the fact

Since requirements applied to route handlers are static, they can be quite difficult to manipulate after the fact. Fancy foot work with requirement factories can ease this some but at the cost of complexity, manual management and potentially tricky application or request scoped context locals.

To address this, `flask-allows` provides a mechanism for overriding and adding additional requirements itself.

Note: In order to use these features with *Class Based Requirements*, you must define both an `__eq__` and `__hash__` method on the requirement:

```
class Has(Requirement):
    def __init__(self, permission):
        self.permission = permission

    def fulfill(self, user):
        return self.permission in user.permissions

    def __eq__(self, other):
        return isinstance(other, Has) and self.permission == other.permission

    def __hash__(self):
        return hash(self.permission)
```

Since this quite a bit of boilerplate, consider using attrs in conjunction with this library as well:

```
import attr


@attr.s(frozen=True)
class Has(Requirement):
    permission = attr.ib()

    def fulfill(self, user):
        return
```

The downside here is that `Has` also becomes orderable, but see python-attrs/attrs #170 for more details.

2.4.1 Disabling Requirements

Disabling requirements can be useful to temporarily allows a specific user access to certain areas of your application. `flask-allows` exposes an `overrides` attribute on the extension object, as well as providing a `current_overrides` context local and an *`Override`* class, each of these play a separate role in the process:

- `allows.overrides` is the *`OverrideManager`* instance associated with the extension object. It is strongly recommended to use this instance rather than instantiating your own.
- `current_overrides` is a context local that points towards the current override context.
- *`Override`* is the representation of the override context.

Note: `current_overrides` is a context local managed separately from the application and requests contexts. However, the Allows extension object registers before and after request handlers to push and cleanup override contexts.

`flask-allows` automatically starts an override context at the beginning of a request so we can immediately being overriding requirements by calling *`add()`*:

```
from flask_allows import current_overrides
from .app.requirements import is_admin

current_overrides.add(is_admin)
```

We can also remove a requirement from the override context with *`remove()`*:

```
current_overrides.remove(is_admin)
```

Both `add` and `remove` accept multiple requirements but must always be passed at least one requirement.

Note: Adding and removing from `current_overrides` affects the current context directly. If this is an object you're holding a reference to, you will see the changes reflected in it.

It is possible to temporarily replace the current context with a new one with `OverrideManager`'s *`override()`* method which acts as a context manager:

```
with allows.overrides.override(Override(is_admin)):
    ...
```

When the block is entered, a new override context is pushed and when the block exits, it is popped. This context manager also yields the new context into the block for convenience sake:

```
with allows.overrides.override(Override(is_admin)) as overrides:
    ...
```

If the new context should augment rather than entirely replace the current context, you may supply the `use_parent` argument to `override`:

```
with allows.overrides.override(Override(is_admin), use_parent=True):
    ...
```

Behind the scenes, this creates a new Override instance that combines the disabled requirements from the current context and the child context rather than changing either's state directly. This makes transitioning back to the original context easier.

If we need to check if the current override context overrides a requirement, that is possible with either the `is_overridden` method or the `in` operator:

```
current_overrides.is_overridden(is_admin)
is_admin in current_overrides
```

2.4.2 Manually managing override contexts

We can also manually manage overrides on a global scale by using the manager's *push()* and *pop()* methods. This can be useful when working outside the request-response cycle, such as in a CLI context or out-of-band task runner such as celery.

Danger: *pop()* checks that the popped context belongs to the manager instance that popped the context. If a separate manager instance pushed the last context or if a context was not active when `pop` was called, a `RuntimeError` is raised to signal this error.

To begin a manual override context we must first call `push` method with an `Override` instance:

```
allows.overrides.push(Override())
```

This replaces the current context rather than augments it and `current_overrides` points at this instance. If newly pushed context should augment the existing context rather than replacing it entirely, you may supply the `use_parent` argument – this behaves the same as when provided with the manager's `override` method.

When we are done with this context, we must call the `pop` method to end the context and replace it with its parent:

```
allows.overrides.pop()
```

2.4.3 Adding More Requirements

In a similar vein as the `OverrideManager`, you may also add more requirements to the context as well. To achieve this, `flask-allows` exposes an `additional` attribute on the extension object, as well as a `current_additions` and an `Additional` class, each plays a similar role to their override counterparts:

- `allows.additional` is the *AdditionalManager* instance associated with the extension object. It's strongly recommended to use this instance rather than instantiating your own.
- `current_additions` is a context local that points towards the current additional context.
- *Additional* is the representation of the additional context.

Note: `current_additions` is a context local managed separately from the application and requests contexts. However, the Allows extension object registers before and after request handlers to push and cleanup additional contexts.

`flask-allows` manages additional contexts in the same fashion as an override context, automatically starting and ending the context in tune with the request cycle:

```
from flask_allows import current_additions
from .myapp.requirements import is_admin

current_additions.add(is_admin)
```

And removing the additional requirement:

```
current_additions.remove(is_admin)
```

`add` and `remove` can accept multiple arguments but must always be passed at least one requirement.

It is also possible to temporarily replace the current additional context with a new one by using the `AdditionalManager`'s *`additional()`* method:

```
with allows.additional.additional(Additional(is_admin)):
    ...
```

Just like with *`OverrideManager`* this method will inject the new context into the block and can accept a `use_parent` argument to combine the new context and the current context into one:

```
with allows.additional.additional(Additional(is_admin), use_parent) as added:
    assert added.is_added(is_admin)
```

Additional objects can be checked for membership using either the `is_added()` method or with `in`:

```
current_additions.add(is_admin)
current_additions.is_added(is_admin)
is_admin in current_additions
```

And Additional instances may be length checked and iterated as well:

```
current_additions.add(is_admin)
assert len(current_additions) == 1
assert list(current_additions) == [is_admin]
```

2.4.4 Manually Managing Additional Contexts

Additional contexts can also be managed manually at the global level with the *`push()`* and *`pop()`* methods. This can be useful when working outside the request cycle such as in an out of band task worker such as celery.

Danger: *`pop()`* checks that the popped context belongs to the manager instance that popped the context. If a separate manager instance pushed the last context or if a context was not active when `pop` was called, a `RuntimeError` is raised to signal this error.

To being managing the context, we must first call the manager's `push` method with an *`Additional`* instance:

```
allows.overrides.push(Additional(is_admin))
```

This replaces the current context rather than augmenting it and `current_additions` will being pointing at this context. If augmenting is preferred, the `use_parent` argument can be passed, this behaves the same as when provided to the `additional` method.

When we are finished with this context, we must called the `pop` method to remove the context and restore its parent:

```
allows.additional.pop()
```

2.5 Controlling Failure

When dealing with permissioning, failure is an expected and desired outcome. And Flask-Allows provides several measures to deal with this failure.

2.5.1 Throwing an exception

The first measure is the ability to configure requirements runners to throw an exception. By default this will be werkzeug's Forbidden exception. However, this can be set to be any exception type or specific instance. The easiest way to set this is through the *Allows* constructor:

```
class PermissionError(Exception):
    def __init__(self):
        super().__init__("I'm sorry Dave, I'm afraid I can't do that")


allows = Allows(throws=PermissionError)

# alternatively
allows = Allows(throws=PermissionError())
```

If a particular exception is desirable most of but not all of the time, an exception type or instance can be provided each requirements runner:

```
# to Permission helper
Permission(SomeRequirement(), throws=PermissionError)

# to decorators
@allows.requires(SomeRequirement(), throws=PermissionError)
@requires(SomeRequirement(), throws=PermissionError)
```

When an exception type or instance is provided to a requirements runner, it takes precedence over the type or instance registered on the extension object. If one is not supplied to a requirement runner, it uses the type or instance registered on the extension object.

2.5.2 Failure Callback

Another way to handle failure is providing an `on_fail` argument that will be invoked when failure happens. The value provided to `on_fail` doesn't have to be a callable, so any value is appropriate. If the value provided is a callable it should be prepared to accept any arbitrary arguments that were provided when the requirement runner that was invoked.

To add a failure callback, it can be provided to the *Allows* constructor:

```
def flash_failure_message(*args, **kwargs):
    flash("I'm sorry Dave, I'm afraid I can't do that", "error")

allows = Allows(on_fail=flash_failure_message)
```

If `on_fails` return a non-`None` value, that will be used as the return value from the requirement runner. However, if a `None` is returned from the callback, then the configured exception is raised instead. In the above example, since a `None` is implicitly returned from the callback, a werkzeug Forbidden exception would be raised from any requirements runners.

An example of returning a value from the callback would be returning a redirect to another page:

```
def redirect_to_home(*args, **kwargs):
    flash("I'm sorry Dave, I'm afraid I can't do that", "error")
    return redirect(url_for("index"))
```

However, any value can be returned from this wrapper.

Note: When used with the *Permission* helper, the callback will be invoked with no arguments and the return value isn't considered.

Danger: When using `on_fail` with route decorators, be sure to return an appropriate value for Flask to turn into a response.

Similar to exception configuration, `on_fail` can be passed to any requirements runner:

```
# to Permission helper
Permission(SomeRequirement(), on_fail=flash_failure_message)

# to decorators
@allow.requires(SomeRequirement(), on_fail=redirect_to_home)
@requires(SomeRequirement(), on_fail=redirect_to_home)
```

When `on_fail` is passed to a requirements runner, it takes precedence over the `on_fail` registered on the extension object. If an `on_fail` isn't provided then the one registered on the extension object is used.

2.6 flask_allows API

2.6.1 Extension

class flask_allows.allows.**Allows**(*app=None*, *identity_loader=None*, *throws=<class 'werkzeug.exceptions.Forbidden'>*, *on_fail=None*)

The Flask-Allows extension object used to control defaults and drive behavior.

Parameters

- **app** – Optional. Flask application instance.
- **identity_loader** – Optional. Callable that will load the current user
- **throws** – Optional. Exception type to raise by default when authorization fails.
- **on_fail** – Optional. A value to return or function to call when authorization fails.

clear_all_additional()

Helper method to remove all additional contexts, this is called automatically during the after request phase in Flask. However it is provided here if additional contexts need to be cleared independent of the request cycle.

If an additional context is found that originated from an AdditionalManager instance not controlled by the Allows object, a `RuntimeError` will be raised.

clear_all_overrides()

Helper method to remove all override contexts, this is called automatically during the after request phase in Flask. However it is provided here if override contexts need to be cleared independent of the application context.

If an override context is found that originated from an OverrideManager instance not controlled by the Allows object, a `RuntimeError` will be raised.

fulfill(*requirements*, *identity=None*)
Checks that the provided or current identity meets each requirement passed to this method.

This method takes into account both additional and overridden requirements, with overridden requirements taking precedence:

```
allows.additional.push(Additional(Has('foo')))
allows.overrides.push(Override(Has('foo')))

allows.fulfill([], user_without_foo)  # return True
```

Parameters

- **requirements** – The requirements to check the identity against.
- **identity** – Optional. Identity to use in place of the current identity.

identity_loader(*f*)
Used to provide an identity loader after initialization of the extension.

Can be used as a method:

```
allows.identity_loader(lambda: a_user)
```

Or as a decorator:

```
@allows.identity_loader
def load_user():
    return a_user
```

If an identity loader is provided at initialization, this method will overwrite it.

Parameters **f** – Callable to load the current user

init_app(*app*)
Initializes the Flask-Allows object against the provided application

requires(**requirements*, ***opts*)
Decorator to enforce requirements on routes

Parameters

- **requirements** – Collection of requirements to impose on view
- **throws** – Optional, keyword only. Exception to throw for this route, if provided it takes precedence over the exception stored on the instance
- **on_fail** – Optional, keyword only. Value or function to use as the on_fail for this route, takes precedence over the on_fail configured on the instance.

run(*requirements*, *identity=None*, *throws=None*, *on_fail=None*, *f_args=()*, *f_kwargs={}*, *use_on_fail_return=True*)
Used to preform a full run of the requirements and the options given, this method will invoke on_fail and/or throw the appropriate exception type. Can be passed arguments to call on_fail with via f_args (which are passed positionally) and f_kwargs (which are passed as keyword).

Parameters

- **requirements** – The requirements to check

- **identity** – Optional. A specific identity to use for the check
- **throws** – Optional. A specific exception to throw for this check
- **on_fail** – Optional. A callback to invoke after failure, alternatively a value to return when failure happens
- **f_args** – Positional arguments to pass to the on_fail callback
- **f_kwargs** – Keyword arguments to pass to the on_fail callback
- **use_on_fail_return** – Boolean (default True) flag to determine if the return value should be used. If true, the return value will be considered, else failure will always progress to exception raising.

2.6.2 Permission Helper

class flask_allows.permission.**Permission**(**requirements*, ***opts*)

Used to check requirements as a boolean or context manager. When used as a boolean, it only runs the requirements and returns the raw boolean result:

```
p = Permission(is_admin)

if p:
    print("Welcome to the club!")
```

When used as a context manager, it runs both the check and the failure handlers if the requirements are not met:

```
p = Permission(is_admin)

with p:
    # will run on_fail and throw before reaching if the
    # requirements on p return False
    print("Welcome to the club!")
```

Note: Both the context manager and boolean usages require an active application context to use.

Parameters

- **requirements** – The requirements to check against
- **throws** – Optional, keyword only. Exception to throw when used as a context manager, if provided it takes precedence over the exception stored on the current application's registered *Allows* instance
- **on_fail** – Optional, keyword only. Value or function to use as the on_fail when used as a context manager, if provided takes precedence over the on_fail configured on current application's registered *Allows* instance
- **identity** – Optional, keyword only. An identity to verify against instead of the using the loader configured on the current application's registered *Allows* instance

2.6.3 Requirements Base Classes

class flask_allows.requirements.**Requirement**

Base for object based Requirements in Flask-Allows. This is quite useful for requirements that have complex

logic that is too much to fit inside of a single function.

fulfill(*user*, *request=None*)

Abstract method called to verify the requirement against the current user and request.

Changed in version 0.5.0: Passing request is now deprecated, pending removal in version 1.0.0

Parameters

- **user** – The current identity
- **request** – The current request.

class flask_allows.requirements.**ConditionalRequirement**(**requirements*, ***kwargs*)

Used to combine requirements together in ways other than all-or-nothing, such as with an or-reducer (any requirement must be True):

```
from flask_allows import Or

requires(Or(user_is_admin, user_is_moderator))
```

or negating a requirement:

```
from flask_allows import Not

requires(Not(user_logged_in))
```

Combinations may also nested:

```
Or(user_is_admin, And(user_is_moderator, HasPermission('view_admin')))
```

Custom combinators may be built by creating an instance of ConditionalRequirement and supplying any combination of its keyword parameters

This class is also exported under the `C` alias.

Parameters

- **requirements** – Collection of requirements to combine into one logical requirement
- **op** – Optional, Keyword only. A binary operator that accepts two booleans and returns a boolean.
- **until** – Optional, Keyword only. A boolean to short circuit on (e.g. if provided with True, then the first True evaluation to return from a requirement ends verification)
- **negated** – Optional, Keyword only. If true, then the ConditionalRequirement will return the opposite of what it actually evaluated to (e.g. `ConditionalRequirement(user_logged_in, negated=True)` returns False if the user is logged in)

classmethod And(**requirements*)

Short cut helper to construct a combinator that uses `operator.and_()` to reduce requirement results and stops evaluating on the first False.

This is also exported at the module level as `And`

classmethod Not(**requirements*)

Shortcut helper to negate a requirement or requirements.

This is also exported at the module as `Not`

classmethod Or(**requirements*)
Short cut helper to construct a combinator that uses `operator.or_()` to reduce requirement results and stops evaluating on the first True.

This is also exported at the module level as `Or`

fulfill(*user*, *request*)
Abstract method called to verify the requirement against the current user and request.

Changed in version 0.5.0: Passing request is now deprecated, pending removal in version 1.0.0

Parameters

- **user** – The current identity
- **request** – The current request.

2.6.4 Override Management

class flask_allows.overrides.**Override**(**requirements*)
Container object that allows selectively disabling requirements.

Requirements can be disabled by passing them to the constructor or by calling the `add` method. They can be re-enabled by calling the `remove` method. To check if a requirement is currently disabled, you may call either `is_overridden` or use `in`.

Override objects can be combined and compared to each other with the following operators:

+ creates a new overide object by combining two others, the new override overrides all requirements that both parents did.

+= similar to + except it is an inplace update.

− creates a new override instance by removing any overrides from the first instance that are contained in the second instance.

−= similar to − except it is an inplace update

== compares two overrides and returns true if both have the same disabled requirements.

!= similar to == except returns true if both have different disabled requirements.

add(*requirement*, **requirements*)
Adds one or more requirements to the override context.

is_overridden(*requirement*)
Checks if a particular requirement is current overridden. Can also be used as `in`:

```
override = Override()
override.add(is_admin)
override.is_overridden(is_admin)   # True
is_admin in override  # True
```

remove(*requirement*, **requirements*)
Removes one or more requirements from the override context.

class flask_allows.overrides.**OverrideManager**
Used to manage the process of overriding and removing overrides. This class shouldn't be used directly, instead use `allows.overrides` to access these controls.

current
Returns the current override context if set otherwise None

override(*override*, *use_parent=False*)
Allows temporarily pushing an override context, yields the new context into the following block.

pop()
Pops the latest override context.

If the override context was pushed by a different override manager, a `RuntimeError` is raised.

push(*override*, *use_parent=False*)
Binds an override to the current context, optionally use the current overrides in conjunction with this override

If `use_parent` is true, a new override is created from the parent and child overrides rather than manipulating either directly.

class flask_allows.additional.**Additional**(**requirements*)
Container object that allows to run extra requirements on checks. These additional requirements will be run at most once per check and will occur in no guarenteed order.

Requirements can be added by passing them into the constructor or by calling the `add` method. They can be removed from this object by calling the `remove` method. To check if a requirement has been added to the current conext, you may call `is_added` or use `in`:

```
some_req in additional
additional.is_added(some)req)
```

Additional objects can be iterated and length checked:

```
additional = Additional(some_req)
assert len(additional) == 1
assert list(additional) == [some_req]
```

Additional objects may be combined and compared to each other with the following operators:

+ creates a new additional object by combining two others, the new additional supplies all requirements that both parents did.

+= similar to + except it is an inplace update.

− creates a new additional instance by removing any requirements from the first instance that are contained in the second instance.

−= similar to − except it is an inplace update.

== compares two additional instances and returns true if both have the same added requirements.

!= similar to != except returns true if both have different requirements contained in them.

class flask_allows.additional.**AdditionalManager**
Used to manage the process of adding and removing additional requirements to be run. This class shouldn't be used directly, instead use `allows.additional` to access these controls.

additional(*additional*, *use_parent=False*)
Allows temporarily pushing an additional context, yields the new context into the following block.

current
Returns the current additional context if set otherwise None

pop()
Pops the latest additional context.

If the additional context was pushed by a different additional manager, a `RuntimeError` is raised.

push(*additional*, *use_parent=False*)
Binds an additional to the current context, optionally use the current additionals in conjunction with this additional

If `use_parent` is true, a new additional is created from the parent and child additionals rather than manipulating either directly.

2.6.5 Utilities

`flask_allows.views.`**`requires`**(**requirements*, ***opts*)
Standalone decorator to apply requirements to routes, either function handlers or class based views:

```
@requires(MyRequirement())
def a_view():
    pass

class AView(View):
    decorators = [requires(MyRequirement())]
```

Parameters

- **`requirements`** – The requirements to apply to this route
- **`throws`** – Optional. Exception or exception instance to throw if authorization fails.
- **`on_fail`** – Optional. Value or function to use when authorization fails.
- **`identity`** – Optional. An identity to use in place of the currently loaded identity.

`flask_allows.views.`**`exempt_from_requirements`**(*f*)
Used to exempt a route handler from ambient requirement handling unless it is explicitly decorated with a requirement runner:

```
@bp.route('/')
@exempt_from_requirements
def greeting_area():
    return "Hello!"
```

To use with a class based view, apply it to the class level decorators attribute:

```
class SomeCBV(View):
    decorators = [exempt_from_requirements]

    def get(self):
        return "Hello!"
```

Note: You cannot exempt individual methods of a class based view with this decorator, e.g. the follow will not work:

```
class SomeCBV(MethodView):

    @exempt_from_requirements
    def get(self):
        return "Hello!"
```

Any permissioning applied at the blueprint level would still affect this route.

Parameters **f** – The route handler to be decorated.

New in version 0.7.0.

flask_allows.views.**guard_entire**(*requirements*, *identity=None*, *throws=None*, *on_fail=None*)

Used to protect an entire blueprint with a set of requirements. If a route handler inside the blueprint should be exempt, then it may be decorated with the *exempt_from_requirements()* decorator.

This function should be registered as a before_request handler on the blueprint and provided with the requirements to guard the blueprint with:

```
my_bp = Blueprint(__name__, 'namespace')
my_bp.before_request(guard_entire(MustBeLoggedIn()))
```

identity, on_fail and throws may also be provided but are optional. If on_fails returns a non-None result, that will be considered the return value of the routing:

```
from flask import flash, redirect

def flash_and_redirect(message, level, endpoint):
    def _(*a, **k):
        flash(message, level)
        return redirect(endpoint)
    return _

bp = Blueprint(__name__, 'namespace')
bp.before_request(
    guard_entire(
        [MustBeLoggedIn()],
        on_fail=flash_and_redirect(
            "Please login in first",
            "warning",
            "login"
        )
    )
)
```

on_fail will also receive anything found in flask.request.view_args as keyword arguments.

If needed, this guard may be applied multiple times. This may be useful if different conditions should result in different *on_fail* mechanisms being invoked:

```
bp = Blueprint(__name__, "admin_panel")
bp.before_request(
    guard_entire(
        [MustBeLoggedIn()],
        on_fail=flash_and_redirect(
            "Please login in first",
            "warning",
            "login"
        )
    )
)
bp.before_request(
    guard_entire(
        [MustBeAdmin()],
        on_fail=flash_and_redirect(
            "You are not an admin.",
```

(continues on next page)

(continued from previous page)

```
            "danger",
            "index"
        )
    )
)
```

Parameters

- **requirements** – An iterable of requirements to apply to every request routed to the blueprint.
- **identity** – Optional. The identity that should be used for fulfilling requirements on the blueprint level.
- **throws** – Optional. Exception or exception type to be thrown if authorization fails.
- **on_fail** – Optional. Value or function to use if authorization fails.

flask_allows.requirements.**wants_request**(*f*)

Helper decorator for transitioning to user-only requirements, this aids in situations where the request may be marked optional and causes an incorrect flow into user-only requirements.

This decorator causes the requirement to look like a user-only requirement but passes the current request context internally to the requirement.

This decorator is intended only to assist during a transitionary phase and will be removed in flask-allows 1.0

See: #20#27

2.7 Change Log

2.7.1 Version 0.7 (2018-XX-XX)

- Added `flask_allows.view.guard_entire` and `flask_allows.views.exempt_from_requirements` to make protecting entire blueprints easier.
- Added `__all__` export markers to flask_allows modules to prevent accidental re-export of other symbols when using `from flask_allows.module import *`

2.7.2 Version 0.6 (2018-05-26)

- `Permission` no longer needs an application context for construction but does require one for evaluation now.
- Added `Allows.run` a public helper that performs the complete fulfill and fail cycle. This cleans up the duplication between `Allows.requires`, `requires` and `Permission.__enter__`
- Removed `Permission.throw_type`
- Removed `PermissionedView`, `PermissionedMethodView`
- Added ability to disable requirements already registered on routes and inside handlers via the `flask_allows.overrides.OverrideManager` class
- Added ability to add more requirements to be run during checks on routes and other handlers via the `flask_allows.additional.AdditionalManager` class

2.7.3 Version 0.5.1 (2018-04-22)

- Added `wants_request` to ease transition to user only requirements

2.7.4 Version 0.5 (2018-04-17)

- Real documentation
- Accepting request in requirements is now deprecated, pending removal in 1.0
- Promoted internal `_allows` context local to part of the public interface
- **Bug Fix: ConditionalRequirement returned False when no requirements were provided** it now returns True and mimics the behavior of `all` better

2.7.5 Version 0.4 (2017-08-29)

- Clarify deprecation message for PermissionedView, will be removed in 0.6
- Deprecate Permission.throw_type for 0.6 removak
- Add optional on_fail for *allows.requires* and *requires* decoration
- Fix coverage path problem with local testing

2.7.6 Version 0.3.2 (2017-08-29)

- Fix package data not being included

2.7.7 Version 0.3.1 (2017-08-20)

- Fix error that prevent sdist builds on 2.7

2.7.8 Version 0.3 (2017-08-20)

- Drop official support for Python 3.3
- Deprecate implicit decoration on class based views via the requirements attribute

Index