pyramid$_{r}etryDocumentation$

Release 2.1.1

Michael Merickel

Apr 22, 2022

Contents

`pyramid_retry` is an execution policy for Pyramid that wraps requests and can retry them a configurable number of times under certain "retryable" error conditions before indicating a failure to the client.

> **Warning:** This package will only work with Pyramid 1.9 and newer.

CHAPTER 1

Installation

1.1 Stable release

To install pyramid_retry, run this command in your terminal:

```
$ pip install pyramid_retry
```

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2 From sources

The sources for pyramid_retry can be downloaded from the Github repo.

```
$ git clone https://github.com/Pylons/pyramid_retry.git
```

Once you have a copy of the source, you can install it with:

```
$ pip install -e .
```

CHAPTER 2

Usage

Activate `pyramid_retry` by including it in your application:

```
def main(global_config, **settings):
    config = Configurator(settings=settings)
    config.include('pyramid_retry')
    # ...
    config.add_route('home', '/')
```

By default `pyramid_retry` will register an instance of *`pyramid_retry.RetryableExecutionPolicy()`* as an *execution policy* in your application using the `retry.attempts` setting as the maximum number of attempts per request. The default number of attempts is `3`. This number is configurable in your application's `.ini` file as follows:

```
[app:main]
# ...
retry.attempts = 3
```

The policy will handle any requests that fail because the application raised an instance of *`pyramid_retry.RetryableException`* or another exception implementing the *`pyramid_retry.IRetryableError`* interface.

The below, very contrived example, shows conceptually what's going on when a request is retried. The `failing_view` is executed initially and for the final attempt the `recovery_view` is executed.

```
@view_config(route_name='home')
def failing_view(request):
    raise RetryableException

@view_config(route_name='home', is_last_attempt=True, renderer='string')
def recovery_view(request):
    return 'success'
```

Of course you probably wouldn't write actual code that expects to fail like this. More realistically you may use a library like pyramid_tm to translate certain transactional errors marked as "transient" into retryable errors.

2.1 Custom Retryable Errors

The simple approach to marking errors as retryable is to simply catch the error and raise a *pyramid_retry.RetryableException* instead:

```
from pyramid_retry import RetryableException
import requests

def view(request):
    try:
        response = requests.get('https://www.google.com')
    except requests.Timeout:
        raise RetryableException
```

This will work but if this is the last attempt then the failed request will not actually be retried and on top of that the original exception is lost.

A better approach is to preserve the original exception and simply mark it as retryable using the *pyramid_retry.IRetryableError* marker interface:

```
from pyramid_retry import mark_error_retryable
import requests
import zope.interface

# mark requests.Timeout errors as retryable
mark_error_retryable(requests.Timeout)

def view(request):
    response = requests.get('https://www.google.com')
```

2.2 Per-Request Attempts

It may be desirable to override the attempts per-request. For example, if one endpoint on the system cannot afford to make a copy of the request via `request.make_body_seekable()` then the activate hook can be used to set `attempts=` on that endpoint.

```
def activate_hook(request):
    if request.path == '/upload':
        return 1  # disable retries on this endpoint

config.add_settings({'retry.activate_hook': activate_hook})
```

The `activate_hook` should return a number `>= 1` or `None`. If `None` then the policy will fallback to the `retry.attempts` setting.

2.3 View Predicates

When the library is included in your application it registers two new view predicates which are especially useful on exception views to determine when to handle certain errors.

`retryable_error=[True/False]` will match the exception view only if the exception is both an *retryable error* **and** there are remaining attempts in which the request would be retried. See *pyramid_retry.RetryableErrorPredicate* for more information.

`last_retry_attempt=[True/False]` will match only if, when the view is executed, there will not be another attempt for this request. See *pyramid_retry.LastAttemptPredicate* for more information.

2.4 Receiving Retry Notifications

The *pyramid_retry.IBeforeRetry* event can be subscribed to receive a callback with the `request` and `environ` prior to the pipeline being completely torn down. This can be very helpful if any state is stored on the `environ` itself that needs to be reset prior to the retry attempt.

```
from pyramid.events import subscriber
from pyramid_retry import IBeforeRetry

@subscriber(IBeforeRetry)
def retry_event(event):
    print(f'A retry is about to occur due to {event.exception}.')
```

The `exception` attribute indicates the exception that triggered the retry. The exception may come from either `request.exception` if it was caught and a response was rendered, or it may come from an uncaught exception.

CHAPTER 3

Caveats

- In order to guarantee that a request can be retried it must make the body seekable. This is done via `request.make_body_seekable()`. Generally the body is loaded directly from `environ['wsgi.input']` which is controlled by the WSGI server. However to make the body seekable it is copied into a seekable wrapper. In some cases this can lead to a very large copy operation before the request is executed.
- `pyramid_retry` does not copy the `environ` or make any attempt to restore it to its original state before retrying a request. This means anything stored on the `environ` will persist across requests created for that `environ`.

CHAPTER 4

More Information

4.1 pyramid_retry API

pyramid_retry.**includeme**(*config*)

Activate the pyramid_retry execution policy in your application.

This will add the pyramid_retry.RetryableErrorPolicy() with attempts pulled from the retry.attempts setting.

The last_retry_attempt and retryable_error view predicates are registered.

This should be included in your Pyramid application via config.include('pyramid_retry').

pyramid_retry.**RetryableExecutionPolicy**(*attempts=3*, *activate_hook=None*)

Create a *execution policy* that catches any *retryable error* and sends it through the pipeline again up to a maximum of attempts attempts.

If activate_hook is set it will be consulted prior to each request to determine if retries should be enabled. It should return a number > 0 of attempts to be used or None which will indicate to use the default number of attempts.

pyramid_retry.**mark_error_retryable**(*error*)

Mark an exception instance or type as retryable. If this exception is caught by pyramid_retry then it may retry the request.

pyramid_retry.**is_error_retryable**(*request*, *exc*)

Return True if the exception is recognized as *retryable error*.

This will return False if the request is on its last attempt. This will return False if pyramid_retry is inactive for the request.

pyramid_retry.**is_last_attempt**(*request*)

Return True if the request is on its last attempt, meaning that pyramid_retry will not be issuing any new attempts, regardless of what happens when executing this request.

This will return True if pyramid_retry is inactive for the request.

class pyramid_retry.**LastAttemptPredicate**(*val*, *config*)
A *view predicate* registered as `last_retry_attempt`. Can be used to determine if an exception view should execute based on whether it's the last retry attempt before aborting the request.

See also:

See *pyramid_retry.is_last_attempt()*.

class pyramid_retry.**RetryableErrorPredicate**(*val*, *config*)
A *view predicate* registered as `retryable_error`. Can be used to determine if an exception view should execute based on whether the exception is a *retryable error*.

See also:

See *pyramid_retry.is_error_retryable()*.

exception pyramid_retry.**RetryableException**
A retryable exception should be raised when an error occurs.

interface pyramid_retry.**IRetryableError**
A marker interface for retryable errors.

An interface can be applied to any `Exception` class or object to indicate that it should be treated as a *retryable error*.

interface pyramid_retry.**IBeforeRetry**
An event emitted immediately prior to throwing away the request and creating a new one.

This event may be useful when state is stored on the `request.environ` that needs to be updated before a new request is created.

environ
The environ object that is reused between requests.

request
The request object that is being discarded.

exception
The exception that request processing raised.

response
The response object that is being discarded. This may be `None` if no response was generated, which happens when request processing raises an exception that isn't caught by any exception view.

4.2 Glossary

execution policy A hook in *Pyramid* which can control the entire request lifecycle.

Pyramid A web framework.

retryable error An exception indicating that a request failed due to a transient error which may succeed if tried again. Examples might include lock contention or a flaky network connection to a third party service.

A retryable error is usually an exception that inherits from *pyramid_retry.RetryableException* but may also be any other exception that implements the *pyramid_retry.IRetryableError* interface.

view predicate A predicate in *Pyramid* which can help determine which view should be executed for a given request. Many views may be registered for a similar url, query strings etc and all predicates must pass in order for the view to be considered.

4.3 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.3.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/Pylons/pyramid_retry/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

pyramid_retry could always use more documentation, whether as part of the official pyramid_retry docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/Pylons/pyramid_retry/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.3.2 Get Started!

Ready to contribute? Here's how to set up *pyramid_retry* for local development.

1. Fork the *pyramid_retry* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pyramid_retry.git
```

3. Install your local copy into a virtualenv:

```
$ python3 -m venv env
$ env/bin/pip install -e .[docs,testing]
$ env/bin/pip install tox
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ env/bin/tox
```

6. Add your name to the `CONTRIBUTORS.txt` file in the root of the repository.

7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

4.3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5, 3.6, 3.7, 3.8 and 3.9, and for PyPy. Check https://travis-ci.org/Pylons/pyramid_retry/pull_requests and make sure that the tests pass for all supported Python versions.

4.3.4 Tips

To run a subset of tests:

```
$ env/bin/py.test tests.test_it
```

4.4 Changes

4.4.1 2.1.1 (2020-03-21)

- Ensure the threadlocals are properly popped if the `activate_hook` throws an error or the request body fails to read due to a client disconnect. See https://github.com/Pylons/pyramid_retry/pull/20

4.4.2 2.1 (2019-09-30)

- Add `exception` and `response` attributes to the `pyramid_retry.IBeforeRetry` event. See https://github.com/Pylons/pyramid_retry/pull/19

4.4.3 2.0 (2019-06-06)

- No longer call `invoke_exception_view` if the policy catches an exception. If on the last attempt or non-retryable then the exception will now bubble out of the app and into WSGI middleware. See https://github.com/Pylons/pyramid_retry/pull/17

4.4.4 1.0 (2018-10-18)

- Support Python 3.7.
- Update the version we require for Pyramid to a non-prerelease so that pip and other tools don't accidentally install pre-release software. See https://github.com/Pylons/pyramid_retry/pull/13

4.4.5 0.5 (2017-06-19)

- Update the policy to to track changes in Pyramid 1.9b1. This is an incompatible change and requires at least Pyramid 1.9b1. See https://github.com/Pylons/pyramid_retry/pull/11

4.4.6 0.4 (2017-06-12)

- Add the `mark_error_retryable` function in order to easily mark certain errors as retryable for `pyramid_retry` to detect. See https://github.com/Pylons/pyramid_retry/pull/8
- Add the `IBeforeRetry` event that can be subscribed to be notified when a retry is about to occur in order to perform cleanup on the `environ`. See https://github.com/Pylons/pyramid_retry/pull/9

4.4.7 0.3 (2017-04-10)

- Support a `retry.activate_hook` setting which can return a per-request number of retries. See https://github.com/Pylons/pyramid_retry/pull/4
- Configuration is deferred so that settings may be changed after `config.include('pyramid_retry')` is invoked until the configurator is committed. See https://github.com/Pylons/pyramid_retry/pull/4
- Rename the view predicates to `last_retry_attempt` and `retryable_error`. See https://github.com/Pylons/pyramid_retry/pull/3
- Rename `pyramid_retry.is_exc_retryable` to `pyramid_retry.is_error_retryable`. See https://github.com/Pylons/pyramid_retry/pull/3

4.4.8 0.2 (2017-03-02)

- Change the default attempts to 3 instead of 1.
- Rename the view predicates to `is_last_attempt` and `is_exc_retryable`.
- Drop support for the `tm.attempts` setting.

- The `retry.attempts` setting is always set now in `registry.settings['retry.attempts']` so that apps can inspect it.

4.4.9 0.1 (2017-03-01)

- Initial release.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index