aiopyramid Documentation

Release 0.3.1

Jason Housley

May 15, 2018

Contents

A library for leveraging pyramid infrastructure asynchronously using the new `asyncio`.

`Aiopyramid` provides tools for making web applications with Pyramid and `asyncio`. It will not necessarily make your application run faster. Instead, it gives you some tools and patterns to build an application on asynchronous servers that handle many active connections.

This is not a fork of Pyramid and it does not rewrite any Pyramid code to run asynchronously! Pyramid is just that flexible.

CHAPTER 1

Getting Started

`Aiopyramid` includes a scaffold that creates a "hello world" application, check it out! The scaffold is designed to work with either gunicorn via a custom worker or uWSGI via the uWSGI asyncio plugin.

For example:

```
pip install aiopyramid gunicorn
pcreate -s aio_starter <project>
cd <project>
python setup.py develop
gunicorn --paste development.ini
```

There is also a websocket scaffold *aio_websocket* with basic tools for setting up a websocket server.

For a more detailed walkthrough of how to setup `Aiopyramid` see the *Tutorial*.

CHAPTER 2

Contents

2.1 Features

Rather than trying to rewrite Pyramid, `Aiopyramid` provides a set of features that will allow you to run existing code asynchronously where possible.

2.1.1 Views

`Aiopyramid` provides three view mappers for calling view callables:

- `CoroutineOrExecutorMapper` maps views to coroutines or separate threads
- `CoroutineMapper` maps views to coroutines
- `ExecutorMapper` maps views to separate threads

When you include `Aiopyramid`, the default view mapper is replaced with the `CoroutineOrExecutorMapper` which detects whether your view callable is a coroutine and does a `yield from` to call it asynchronously. If your view callable is not a coroutine, it will run it in a separate thread to avoid blocking the thread with the main loop. `asyncio` is not thread-safe, so you will need to guarantee that either in memory resources are not shared between view callables running in the executor or that such resources are synchronized.

This means that you should not necessarily have to change existing views. Also, it is possible to restore the default view mapper, but note that this will mean that coroutine views that do not specify `CoroutineMapper` as their view mapper will fail.

If most of your view needs to be a coroutine but you want to call out to code that blocks, you can always use run_in_executor. *Aiopyramid* also provides a decorator, `use_executor()`, for specifying declaratively that a particular routine should run in a separate thread.

For example:

```
import asyncio
from aiopyramid.helpers import use_executor
```

(continues on next page)

(continued from previous page)

```
class DatabaseUtilies:

    @use_executor  # query_it is now a coroutine
    def query_it():
        # some code that blocks
```

2.1.2 Authorization

If you are using the default authorization policy, then you will generally not need to make any modifications to authorize users with `Aiopyramid`. The exception is if you want to use a callable that performs some io for your __acl__. In that case you will simply need to use a synchronized coroutine so that the authorization policy can call your coroutine like a normal Python function during view lookup.

For example:

```
import asyncio

from aiopyramid.helpers import synchronize


class MyResource:
    """
    This resource uses a callable for it's
    __acl__ that accesses the db.
    """

    # this
    __acl__ = synchronize(my_coroutine)

    # or this

    @synchronize
    @asyncio.coroutine
    def __acl__(self):
        ...

    # will work
```

If you are using a custom authorization policy, most likely it will work with `Aiopyramid` in the same fashion, but it is up to you to guarantee that it does.

2.1.3 Authentication

Authentication poses a problem because the interface for authentication policies uses normal Python methods that the framework expects to call noramlly but at the same time it is usually necessary to perform some io to retrieve relevant information. The built-in authentication policies generally accept a callback function that delegates retrieving principals to the application, but this callback function is also expected to be called in the regular fashion. So, it is necessary to use a synchronized coroutine as a callback function.

The final problem is that synchronized coroutines are expected to be called from within a child greenlet, or in other words from within framework code (see *Architecture*). However, it is often the case that we will want to access the policy through `pyramid.request.Request.authenticated_userid` or by calling `remember()`, etc. from within another coroutine such as a view callable.

To handle both situations, `Aiopyramid` provides tools for wrapping a callback-based authentication policy to work asynchronously. For example, the following code in your app constructor will allow you to use a coroutine as a callback.

```
from pyramid.authentication import AuthTktAuthenticationPolicy
from aiopyramid.auth import authn_policy_factory

from .myauth import get_principals

...

# In the includeme or constructor
authentication = authn_policy_factory(
    AuthTktAuthenticationPolicy,
    get_principals,
    'sosecret',
    hashalg='sha512'
)
config.set_authentication_policy(authentication)
```

Relevant authentication tools will now return a coroutine when called from another coroutine, so you would access the authentication policy using `yield from` in your view callable since it performs io.

```
from pyramid.security import remember, forget

...

# in some coroutine

maybe = yield from request.unauthenticated_userid
checked = yield from request.authenticated_userid
principals = yield from request.effective_principals
headers = yield from remember(request, 'george')
fheaders = yield from forget(request)
```

Note: If you don't perform asynchronous io or wrap the authentication policy as above, then don't use `yield from` in your view. This approach only works for coroutine views. If you have both coroutine views and legacy views running in an executor, you will probably need to write a custom authentication policy.

2.1.4 Tweens

Pyramid allows you to write tweens which wrap the request/response chain. Most existing tweens expect those tweens above and below them to run synchronously. Therefore, if you have a tween that needs to run asynchronously (e.g. it looks up some data from a database for each request), then you will need to write that *tween* so that it can wait without other tweens needing to explicitly `yield from` it. For example:

```
import asyncio

from aiopyramid.helpers import synchronize


def coroutine_logger_tween_factory(handler, registry):
    """
    Example of an asynchronous tween that delegates
```

(continues on next page)

(continued from previous page)

```
    a synchronous function to a child thread.
    This tween asynchronously logs all requests and responses.
    """

    # We use the synchronize decorator because we will call this
    # coroutine from a normal python context
    @synchronize
    # this is a coroutine
    @asyncio.coroutine
    def _async_print(content):
        # print doesn't really need to be run in a separate thread
        # but it works for demonstration purposes

        yield from asyncio.get_event_loop().run_in_executor(
            None,
            print,
            content
        )

    def coroutine_logger_tween(request):
        # The following calls are guaranteed to happen in order
        # but they do not block the event loop

        # print the request on the aio event loop
        # without needing to say yield
        # at this point,
        # other coroutines and requests can be handled
        _async_print(request)

        # get response, this should be done in this greenlet
        # and not as a coroutine because this will call
        # the next tween and subsequently yield if necessary
        response = handler(request)

        # print the response on the aio event loop
        _async_print(request)

        # return response after logging is done
        return response

    return coroutine_logger_tween
```

2.1.5 Traversal

When using Pyramid's traversal view lookup, it is often the case that you will want to make some io calls to a database or storage when traversing via *__getitem__*. When using the default traverser, Pyramid will call *__getitem__* as a normal Python function. Therefore, it is necessary to synchronize *__getitem__* on any asynchronous resources like so:

```
import asyncio

from aiopyramid.helpers import synchronize


class MyResource:
    """ This resource performs some asynchronous io. """
```

(continues on next page)

(continued from previous page)

```
    __name__ = "example"
    __parent__ = None

    @synchronize
    @asyncio.coroutine
    def __getitem__(self, key):
        yield from self.example_coroutine()
        return self  # no matter the path, this is the context

    @asyncio.coroutine
    def example_coroutine(self):
        yield from asyncio.sleep(0.1)
        print('I am some async task.')
```

2.1.6 Servers

`Aiopyramid` supports both asynchronous gunicorn and the uWSGI asyncio plugin.

Example gunicorn config:

```
[server:main]
use = egg:gunicorn#main
host = 0.0.0.0
port = 6543
worker_class = aiopyramid.gunicorn.worker.AsyncGunicornWorker
```

Example uWSGI config:

```
[uwsgi]
http-socket = 0.0.0.0:6543
workers = 1
plugins =
    asyncio = 50
    greenlet
```

For those setting up `Aiopyramid` on a Mac, Ander Ustarroz's tutorial may prove useful. Rickert Mulder has also provided a fork of uWSGI that allows for quick installation by running *pip install git+git://github.com/circlingthesun/uwsgi.git* in a virtualenv.

2.1.7 Websockets

`Aiopyramid` provides additional view mappers for handling websocket connections with either gunicorn or uWSGI. Websockets with gunicorn use the websockets library whereas uWSGI has native websocket support. In either case, the interface is the same.

A function view callable for a websocket connection follows this pattern:

```
@view_config(mapper=<WebsocketMapper>)
def websocket_callable(ws):
    # do stuff with ws
```

The `ws` argument passed to the callable has three methods for communicating with the websocket `recv()`, `send()`, and `close()` methods, which correspond to similar methods in the websockets library. A websocket connection that echoes all messages using gunicorn would be:

```
from pyramid.view import view_config
from aiopyramid.websocket.config import WebsocketMapper

@view_config(route_name="ws", mapper=WebsocketMapper)
def echo(ws):
    while True:
        message = yield from ws.recv()
        if message is None:
            break
        yield from ws.send(message)
```

Aiopyramid also provides a view callable class WebsocketConnectionView that has on_message(), on_open(), and on_close() callbacks. Class-based websocket views also have a send() convenience method, otherwise the underyling ws may be accessed as self.ws. Simply extend WebsocketConnectionView specifying the correct view mapper for your server either via the __view_mapper__ attribute or the view_config decorator. The above example could be rewritten in a larger project, this time using uWSGI, as follows:

```
from pyramid.view import view_config
from aiopyramid.websocket.view import WebsocketConnectionView
from aiopyramid.websocket.config import UWSGIWebsocketMapper

from myproject.resources import MyWebsocketContext

class MyWebsocket(WebsocketConnectionView):
    __view_mapper__ = UWSGIWebsocketMapper


@view_config(context=MyWebsocketContext)
class EchoWebsocket(MyWebsocket):

    def on_message(self, message):
        yield from self.send(message)
```

The underyling websocket implementations of uWSGI and websockets differ in how they pass on the WebSocket message. uWSGI always sends *bytes* even when the WebSocket frame indicates that the message is text, whereas websockets decodes text messages to *str*. *Aiopyramid* attempts to match the behavior of websockets by default, which means that it coerces messages from uWSGI to *str* where possible. To adjust this behavior, you can set the use_str flag to *False*, or alternatively to coerce websockets messages back to bytes, set the use_bytes flag to True:

```
# In your app constructor
from aiopyramid.websocket.config import WebsocketMapper

WebsocketMapper.use_bytes = True
```

uWSGI Special Note

Aiopyramid uses a special WebsocketClosed exception to disconnect a greenlet after a websocket has been closed. This exception will be visible in log ouput when using uWSGI. In order to squelch this message, wrap the wsgi application in the ignore_websocket_closed() middleware in your application's constructor like so:

```
from aiopyramid.websocket.helpers import ignore_websocket_closed

...
app = config.make_wsgi_app()
return ignore_websocket_closed(app)
```

2.2 Tutorial

This is a basic tutorial for setting up a new project with *Aiopyramid*.

2.2.1 Install Aiopyramid and Initialize Project

It is highly recommended that you use a virtual environment for your project. The tutorial will assume that you are using virtualenvwrapper with a virtualenv created like so:

```
mkvirtualenv aiotutorial --python=/path/to/python3.4/interpreter
```

Once you have your tutorial environment active, install `Aiopyramid`:

```
pip install aiopyramid
```

This will also install the Pyramid framework. Now create a new project using the `aio_websocket` scaffold.

```
pcreate -s aio_websocket aiotutorial
```

This will make an `aiotutorial` directory with the following structure:

```
.
├── aiotutorial          << Our Python package
│   ├── __init__.py      << main file, contains the app constructor
│   ├── templates        << directory for storing jinja templates
│   │   └── home.jinja2 << template for the example homepage, contains a websocket␣
↪test
│   ├── tests.py         << tests module, contains tests for each of our existing views
│   └── views.py         << views module, contains view callables
├── CHANGES.rst          << file for tracking changes to the library
├── development.ini      << config file, contains project and server settings
├── MANIFEST.in          << manifest file for distributing the project
├── README.rst           << readme for bragging about the project
└── setup.py             << Python module for distributing the package and managing␣
↪dependencies
```

Let's look at some of these files a little closer.

2.2.2 App Constructor

The `aiotutorial/__init__.py` file contains the constructor for our app. It loads the logging config from the `development.ini` config file and sets up Python logging. This is necessary because the logging configuration won't be automatically detected when using Python3. Then, it sets up two routes `home` and `echo` that we can tie into with our views. Finally, the constructor scans the project for configuration decorators and builds the wsgi callable.

The app constructor is the place where we will connect Python libraries to our application and perform other configuration tasks.

```
1 import logging.config
2
3 from pyramid.config import Configurator
4
5
6 def main(global_config, **settings):
7     """ This function returns a Pyramid WSGI application.
```

(continues on next page)

(continued from previous page)

```
    """

    # support logging in python3
    logging.config.fileConfig(
        settings['logging.config'],
        disable_existing_loggers=False
    )

    config = Configurator(settings=settings)
    config.add_route('home', '/')
    config.add_route('echo', '/echo')
    config.scan()
    return config.make_wsgi_app()
```

Note: *Thinking Asynchronously*

The app constructor is called once to setup the application, which means that it is a synchronous context. The app is constructed before any requests are served, so it is safe to call blocking code here.

2.2.3 Tests

The `aiotutorial/tests.py` file is a Python module with unittests for each of our views. Let's look at the test case for the home page:

```
class HomeTestCase(unittest.TestCase):

    def test_home_view(self):
        from .views import home

        request = testing.DummyRequest()
        info = asyncio.get_event_loop().run_until_complete(home(request))
        self.assertEqual(info['title'], 'aiotutorial websocket test')
```

Since test runners for unittest expect tests, such as `test_home_view`, to run synchronously but our home view is a coroutine, we need to manually obtain an `asyncio` event loop and run our view. Line 6 obtains a dummy request from `pyramid.testing`. We then pass that request to our view and run it on line 7. Finally, line 8 makes assertions about the kind of output we expect from our view.

2.2.4 Views

This is the brains of our application, the place where decisions about how to respond to a particular request are made, and as such this is the place where you will most often start chaining together coroutines to perform asynchronous tasks. Let's look at each of the example views in turn:

```
@view_config(route_name='home', renderer='aiotutorial:templates/home.jinja2')
@asyncio.coroutine
def home(request):
    wait_time = float(request.params.get('sleep', 0.1))
    yield from asyncio.sleep(wait_time)
    return {'title': 'aiotutorial websocket test', 'wait_time': wait_time}
```

For those already familiar with Pyramid most of this view should require no explanation. The important parts for running asynchronously are lines 2 and 5.

The `view_config()` decorator on line 1 ties this view to the 'home' route declared in the app constructor. It also assigns a renderer to the view that will render the data returned into the `template/home.jinja` template and return a response to the user. Line 2 wraps the view in a coroutine which differentiates it from a generator or native coroutine. Line 3 is the signature for the coroutine. `Aiopyramid` view mappers do not change the two default signarures for views, i.e. views that accept a request and views that accept a context and a request. On line 4, we retrieve a sleep parameter, from the request (the parameter can be either part of the querystring or the body). If the request doesn't include a sleep parameter, the view defaults to 0.1. We don't need to use `yield from` because `request.params.get` doesn't return a coroutine or future. The data for the request exists in memory so retrieving the parameter should be very fast. Line 5 simulates performing some asynchronous task by suspending the coroutine and delegating to another coroutine, `asyncio.sleep()`, which uses events to wait for `wait_time` seconds. Using `yield from` is very important, without it the coroutine would continue without sleeping. Line 6 returns a Python dictionary that will be passed to the jinja2 renderer.

The second view accepts a websocket connection:

```
1 @view_config(route_name='echo', mapper=WebsocketMapper)
2 @asyncio.coroutine
3 def echo(ws):
4     while True:
5         message = yield from ws.recv()
6         if message is None:
7             break
8         yield from ws.send(message)
```

This view is tied to the 'echo' route from the app constructor. Note that we use a special view mapper for websocket connections. The `aiopyramid.websocket.config.WebsocketMapper` changes the signature of the view to accept a single websocket connection instead of a request. The connection object has three methods for communicating with the websocket `recv()`, `send()`, and `close()` that correspond to similar methods in the websockets library.

This websocket view will run echoing the data it recieves until the connection is closed. On line 5 we use `yield from` to wait until a message is received. If the message is None, then we know that the websocket has closed and we break the loop to complete the echo coroutine. Otherwise, line 7 simply returns the same message back to the websocket. Very simple. In both cases when we need to perform some io we use `yield from` to suspend our coroutine and delegate to another.

This kind of explicit yielding is a nice advantage for readability in Python code. It shows us exactly where we are calling asynchronous code.

2.2.5 Development.ini

The `development.ini` file contains the config for the project. Most of these settings could be specified in the app constructor but it makes sense to separate out these values from procedural code. Here is an overview of the two most important sections:

```
[app:main]
use = egg:aiotutorial

pyramid.includes =
    aiopyramid
    pyramid_jinja2
```

(continues on next page)

(continued from previous page)

```
# for py3
logging.config = %(here)s/development.ini
```

The `[app:main]` section contains the settings that will be passed to the app constructor as `settings`. This is where we include extensions for Pyramid such as `Aiopyramid` and the `jinja` templating library.

The `[server:main]` configures the default server for the project, which in this case is `gunicorn`:

```
[server:main]
use = egg:gunicorn#main
host = 0.0.0.0
port = 6543
worker_class = aiopyramid.gunicorn.worker.AsyncGunicornWorker
```

The `port` setting here is the port that we will use to access the application, such as in a browser. The `worker_class` is set to the `aiopyramid.gunicorn.worker.AsyncGunicornWorker` because we need to have `gunicorn` setup the *Aiopyramid Architecture* for us.

2.2.6 Setup

The `setup.py` file makes the `aiotutorial` package easy to distirbute, and it is also a good way, although not the only good way, to manage dependencies for our project. Lines 18-21 list the Python packages that we need for this project.

```
requires = [
    'aiopyramid[gunicorn]',
    'pyramid_jinja2',
]
```

2.2.7 Note about View Mappers

The default view mapper that `Aiopyramid` sets up when it is included by the application tries to be as robust as possible. It will inspect all of the views that we configure and try to guess whether or not they are coroutines. If the view looks like a coroutine, in other words if it has a `yield from` in it, the framework will treat it as a coroutine, otherwise it will assume it is legacy code and will run it in a separate thread to avoid blocking the event loop. This is very important.

When using `Aiopyramid` view mappers, it is actually not necessary to explicitly decorate view callables with `asyncio.coroutine()` as in the examples because the mapper will wrap views that appear to be coroutines for you. It is still good practice to explicitly wrap your views because it facilitates using them in places where a view mapper may not be active, but if you are annoyed by the repetition, then you can skip writing `@asyncio.coroutine` before every view as long as you remember what is a coroutine.

2.2.8 Making Sure it Works

The last step in initializing the project is to install out dependencies and test out that the scaffold works as we expect:

```
python setup.py develop
```

You can also use `setup.py` to run unittests:

```
python setup.py test
```

You should see the following at the end of the output:

```
test_home_view (aiotutorial.tests.HomeTestCase) ... ok
test_echo_view (aiotutorial.tests.WSTest) ... ok

----------------------------------------------------------------------
Ran 2 tests in 1.709s

OK
```

If you don't like the test output from `setup.py`, consider using a test runner like pytest.

Now try running the server and visiting the homepage:

```
gunicorn --paste development.ini
```

Open your browser to http://127.0.0.1:6543 to see the JavaScript test of the our echo websocket. You should see the following output:

```
aiotutorial websocket test

CONNECTED

SENT: Aiopyramid echo test.

RESPONSE: Aiopyramid echo test.

DISCONNECTED
```

This shows that the websocket is working. If you want to verify that the server is able to handle multiple requests on a single thread, simply open a different browser (to avoid browser connection limitations) and go to http://127.0.0.1:6543?sleep=10. The new browser should take roughly ten seconds to load the page because our view is waiting for the value of `sleep`. However, while that request is ongoing, you can refresh your first browser and see that the server is still able to fulfill requests.

Congratulations! You have successfuly setup a highly configurable asynchronous server using `Aiopyramid`!

Note: *Extra Credit*

If you really want to see the power of asynchronous programming in Python, obtain a copy of slowloris and run it against your knew `Aiopyramid` server and some non-asynchronous server. For example, you could run a simple `Django` application with gunicorn. You should see that the `Aiopyramid` server is still able to respond to requests whereas the `Django` server is bogged down. You could also use a simple PHP application using Apache to see this difference.

2.3 Architecture

`Aiopyramid` uses a design similar to the uWSGI asyncio plugin. The `asyncio` event loop runs in a parent greenlet, while wsgi callables run in child greenlets. Because the callables are running in greenlets, it is possible to suspend a callable and switch to parent to run coroutines all on one event loop. Each task tracks which child greenlet it belongs to and switches back to the appropriate callable when it is done.

The greenlet model makes it possible to have any Python code wait for a coroutine even when that code is unaware of asyncio. The uWSGI asyncio plugin sets up the architecture by itself, but it is also possible to setup this architecture whenever we have a running asyncio event loop using spawn_greenlet().

For example, there may be times when a coroutine would need to call some function a that later calls a coroutine b. Since coroutines run in the parent greenlet (i.e. on the event loop) and the function a cannot yield from b because it is not a coroutine itself, the parent coroutine will need to set up the Aiopyramid architecture so that b can be synchronized with synchronize() and called like a normal function from inside a.

The following code demonstrates this usage without needing to setup a server.

```
>>> import asyncio
>>> from aiopyramid.helpers import synchronize, spawn_greenlet
>>>
>>> @synchronize
... @asyncio.coroutine
... def some_async_task():
...   print('I am a synchronized coroutine.')
...   yield from asyncio.sleep(0.2)
...   print('Synchronized task done.')
...
>>> def normal_function():
...   print('I am normal function that needs to call some_async_task')
...   some_async_task()
...   print('I (normal_function) called it, and it is done now like I expect.')
...
>>> @asyncio.coroutine
... def parent():
...   print('I am a traditional coroutine that needs to call the naive normal_function
↪')
...   yield from spawn_greenlet(normal_function)
...   print('All is done.')
...
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(parent())
I am a traditional coroutine that needs to call the naive normal_function
I am normal function that needs to call some_async_task
I am a synchronized coroutine.
Synchronized task done.
I (normal_function) called it, and it is done now like I expect.
All is done.
```

Please feel free to use this in other asyncio projects that don't use Pyramid because it's awesome.

To avoid confusion, it is worth making explicit the fact that this approach is for incorporating code that is fast and non-blocking itself but needs to call a coroutine to do some io. Don't try to use this to call long-running or blocking Python functions. Instead, use run_in_executor, which is what Aiopyramid does by default with view callables that don't appear to be coroutines.

2.3.1 History

Aiopyramid was originally based on pyramid_asyncio, but I chose a different approach for the following reasons:

- The pyramid_asyncio library depends on patches made to the Pyramid router that prevent it from working with the uWSGI asyncio plugin.
- The pyramid_asyncio rewrites various parts of Pyramid, including tweens, to expect coroutines from Pyramid internals.

On the other hand `Aiopyramid` is designed to follow these principles:

- `Aiopyramid` should extend Pyramid through existing Pyramid mechanisms where possible.
- Asynchronous code should be wrapped so that existing callers can treat it as synchronous code.
- Ultimately, no framework can guarantee that all io calls are non-blocking because it is always possible for a programmer to call out to some function that blocks (in other words, the programmer forgets to wrap long-running calls in run_in_executor). So, frameworks should leave the determination of what code is safe to the programmer and instead provide tools for programmers to make educated decisions about what Python libraries can be used on an asynchronous server. Following the Pyramid philosophy, frameworks should not get in the way.

The first principle is one of the reasons why I used view mappers rather than patching the router. View mappers are a mechanism already in place to handle how views are called. We don't need to rewrite vast parts of Pyramid to run a view in the `asyncio` event loop. Yes, Pyramid is that awesome.

The second principle is what allows `Aiopyramid` to support existing extensions. The goal is to isolate asynchronous code from code that expects a synchronous response. Those methods that already exist in Pyramid should not be rewritten as coroutines because we don't know who will try to call them as regular methods.

Most of the Pyramid framework does not run io blocking code. So, it is not actually necessary to change the framework itself. Instead we need tools for making application code asynchronous. It should be possible to run an existing simple url dispatch application asynchronously without modification. Blocking code will naturally end up being run in a separate thread via the run_in_executor method. This allows you to optimize only those highly concurrent views in your application or add in websocket support without needing to refactor all of the code.

It is easy to simulate a multithreaded server by increasing the number of threads available to the executor.

For example, include the following in your application's constructor:

```
import asyncio
from concurrent.futures import ThreadPoolExecutor
...
asyncio.get_event_loop().set_default_executor(ThreadPoolExecutor(max_workers=150))
```

Note: It should be noted that `Aiopyramid` is not thread-safe by nature. You will need to ensure that in memory resources are not modified by multiple non-coroutine view callables. For most existing applications, this should not be a problem.

2.4 Tests

Core functionality is backed by tests. The `Aiopyramid` requires pytest. To run the tests, grab the code on github, install pytest, and run it like so:

```
git clone https://github.com/housleyjk/aiopyramid
cd aiopyramid
pip install pytest
py.test
```

2.5 Indices and Tables

- genindex

- modindex
- search
- glossary

CHAPTER 3

Contributors

- Jason Housley
- Guillaume Gauvrit
- Tiago Requeijo
- Ander Ustarroz
- Ramon Navarro Bosch
- Rickert Mulder

CHAPTER 4

Indices and Tables

- genindex
- modindex
- search
- glossary