aiodjango Documentation

Release 0.2.0a

Mark Lavin

December 22, 2015

This is a proof-of-concept experiment to combine a Django WSGI app mixed with async views/websocket handlers using aiohttp. The API is highly unstable and I wouldn't recommend that you use this code for anything other than wild experimentation.

CHAPTER 1

How It Works

`aidjango.get_aio_application` builds an application which combines both request handlers/views from Django and aiohttp.web. Views are defined using the normal Django url pattern syntax but any handler which is a coroutine is handled by the `aiohttp` application while the rest of the views are handled by the normal Django app.

Internal this makes use of aiohttp-wsgi which runs the Django WSGI app in a thread-pool to minimize blocking the async portions of the app.

CHAPTER 2

Running the Demo

The example project requires Python 3.4+ to run. You should create a virtualenv to install the necessary requirements:

```
$ git clone https://github.com/mlavin/aiodjango.git
$ cd aiodjango/
$ mkvirtualenv aiodjango -p `which python3.4`
(aiodjango) $ add2virtualenv .
(aiodjango) $ cd example
(aiodjango) $ pip install -r requirements.txt
(aiodjango) $ python manage.py migrate
(aiodjango) $ python manage.py runserver
```

This starts the server on http://localhost:8000/ with a new version of Django’s built-in runserver.

CHAPTER 3

Documentation

Additional project documentation can be found on Read The Docs: https://aiodjango.readthedocs.org/

CHAPTER 4

Table of Contents

4.1 Example Usage

aidjango is meant to make it as easy as possible to integrate async view handlers into any Django project. This is handled by auto-discovering coroutine views from the URL configuration. The project defines it's views as normal and aiodjango takes care of the rest.

4.1.1 Defining an Async View

Any coroutine view will be handled by the aiohttp application but they can be defined normally along with your other views.

```
# views.py
import asyncio

from aiohttp import web


@asyncio.coroutine
def async(request):
    return web.Response(text='Hello World')
```

This would then be attached to a URL pattern as normal.

```
# urls.py
from django.core.urls import url

from .views import async


urlpatterns = [
    url(r'^hello/$', async, name='async-hello'),
]
```

aiohttp routing expects the leading slash to be included in any pattern but aiodjango takes care of this for you.

There is a large difference in how aiohttp and Django handle variables in the URL path. While Django breaks these up into args passed to the view function, aiohttp does not and makes that information available on the request.match_info dictionary. As such, all coroutine callbacks should only take a single request argument even if there variables in the path.

4.1.2 Defining the Application

To run both the `aiohttp` application and the original WSGI application, a combined application is created. This would be done in the project's `wsgi.py`.

```
# wsgi.py
import os

from django.core.wsgi import get_wsgi_application

from aiodjango import get_aio_application


os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'example.settings')

# Build WSGI application
# Any WSGI middleware would be added here
application = get_wsgi_application()

# Build aiohttp application
app = get_aio_application(application)
```

The end result is a runable `aiohttp` application so it is recommened that this be named something other than the original WSGI application to not throw off the `WSGI_APPLICATION` setting.

4.1.3 Running the Application

Adding `aiodjango` to your `INSTALLED_APPS` with override the build-in `runserver` command for running for local development. If you are also using `django.contrib.staticfiles` you should be sure to include `aiodjango` above `django.contrib.staticfiles` in the `INSTALLED_APPS` list for this to take effect.

Outside of local development you can use `Gunicorn` to run the application using the `aiohttp` worker class.

```
(example) $ gunicorn example.wsgi:app --worker-class aiohttp.worker.GunicornWebWorker
```

Here `example` is the name of the project and the virtualenv. The `app` is defined in the `example/wsgi.py` as in the previous example. For more information you can see the `aiohttp` docs on deployment.

4.1.4 Caveats

While this might seem too good to be true there are a few caveats. First, is that the URL patterns are implicitly reordered when the `aiohttp` is created to wrap the WSGI application. All of the coroutine patterns are lifted out and will be matched first before falling back to the WSGI views. Second, the URL patterns only allow for named groups and don't currently support positional regex grouping. Finally, it should be noted that the `request` passed to the coroutine callbacks is an instance of `aiohttp.web.Request` not a Django request object. These views should return `aiohttp.web.Response` instances as well. This isn't about making Django async. This is a compatibility shim between Django and `aiohttp`. To fully take advantage of this you'll need to learn the `aiohttp` APIs and use additional `aio` libraries for non-blocking I/O.

4.2 Changelog

History of released for aiodjango.

4.2.1 v0.2 (Under Development)

- Added support for coroutine routes containing named regex groups.
- Initial project documentation.

4.2.2 v0.1 (2015-12-20)

Initial packaged release.

- Basic API for detecting async views and wrapping in `aiohttp` application. No dynamic route support.
- Runserver integration for local development.