Introduction to Asyncio Stack

Igor Davydenko
2015, Lviv.py#4

Asyncio Stack is …

asyncio

- Asynchronous I/O, event loop, coroutines, and tasks
- https://docs.python.org/3/library/asyncio.html

```
import asyncio

@asyncio.coroutine
def hello():
    return "Hello, world!"

loop = asyncio.get_event_loop()
loop.run_until_complete(hello())
loop.close()
```

- Included in Python 3.4 and later
- Available in Python 3.3 with `$ pip install asyncio`
- Backported to Python 2.7 as trollius (*I don't recommend to use it anyway*)

aiohttp

- HTTP client/server for asyncio
- Latest version: `0.16.2`
- http://aiohttp.readthedocs.org/

```python
import asyncio
import aiohttp

@asyncio.coroutine
def fetch_page(url):
    response = yield from aiohttp.request('GET', url)
    assert response.status == 200
    return (yield from response.read())

loop = asyncio.get_event_loop()
content = loop.run_until_complete(fetch_page('http://lvivpy.org.ua/'))
print(content)
loop.close()
```

aiohttp.web

- Web framework for asyncio
- http://aiohttp.readthedocs.org/en/v0.16.2/web.html

```python
import asyncio
from aiohttp import web

@asyncio.coroutine
def hello(request):
    return web.Response(body='Hello, world!', content_type='text/plain')

app = web.Application()
app.router.add_route('GET', '/', hello)
```

```
$ gunicorn -k aiohttp.worker.GunicornWebWorker -w 9 -t 60 app:app
```

aiopg

- Accessing PostgreSQL database from the asyncio
- Latest version: `0.7.0`
- http://aiopg.readthedocs.org/

```
import asyncio
from aiopg import create_pool

@asyncio.coroutine
def hello(dsn):
    pool = yield from create_pool(dsn)
    with (yield from pool.cursor()) as cursor:
        yield from cursor.execute('SELECT 1')
        selected = yield from cursor.fetchone()
        assert selected == (1, )

loop = asyncio.get_event_loop()
loop.run_until_complete(hello('dbname=aiopg user=... password=... host=...'))
loop.close()
```

aioredis / asyncio_redis

- Accessing Redis datasotre from the asyncio

- `aioredis` from `aio-libs`
- Latest version: `0.1.5`
- http://aioredis.readthedocs.org/

- `asyncio_redis` from third-party developers
- Latest version: `0.13.4`
- http://asyncio-redis.readthedocs.org/
- **Supports PUB/SUB**

And many others

- Python Asyncio Resources
- MySQL: aiomysql
- Mongo: asyncio_mongo
- CouchDB: aiocouchdb
- ElasticSearch: aioes
- Memcached: aiomcache
- AMQP: aioamqp
- ØMQ: aiozmq

And many others

- All Asyncio projects @ PyPI
- S3: aio-s3
- SSH: asyncssh

- Autobahn, WebSocket & WAMP
- RxPY, Reactive Extensions for Python
- Pulsar, Concurrent framework for Python

Even web-frameworks available

- muffin
- Induction
- Spanner.py
- Growler

aiohttp.web

Architecture

- All starts from view functions (handlers)
- View functions should be a coroutine, and return `web.Response`

```
import asyncio
from aiohttp import web

@asyncio.coroutine
def index(request):
    return web.Response(body='Hello, world!', content_type='text/plain')
```

- It's good idea to put all view functions to `views.py` module

Architecture

- Next you need to create an `web.Application`
- And register handler for a request

```
from aiohttp import web

from . import views

app = web.Application()
app.router.add_route('GET', '/', views.index)
```

- Obvious to put application code to `app.py` module

Architecture

- Now you ready to serve your application
- I recommend to use Gunicorn

```
$ gunicorn -b 0.0.0.0:8000 -k aiohttp.worker.GunicornWebWorker -w 9 -t 60 project.app:app
```

- Add `--reload` flag to automatically reload Gunicorn server on code change

Handling GET/POST data

In **views.py**,

```python
@asyncio.coroutine
def search(request):
    """Search by query from GET params."""
    query = request.GET['query']
    locale = request.GET.get('locale', 'uk_UA')
    ...
    return web.Response(...)
```

Handling GET/POST data

In **views.py**,

```
@asyncio.coroutine
def submit(request):
    """Submit form POST data."""
    data = yield from request.post()
    # Now POST data available as ``request.POST``
    ...
    return web.Response(...)
```

Handling GET/POST data

In **app.py**,

```
app.router.add_route('GET', '/search', views.search)
app.router.add_route('POST', '/submit', views.submit)
```

Handling variable routes

In **views.py**,

```
@asyncio.coroutine
def project(request):
    project_id = request.match_info['project_id']
    ...
    return web.Response(...)
```

Handling variable routes

In **app.py**,

```
app.router.add_route('GET', '/projects/{project_id}', views.project)
```

Or even,

```
app.router.add_route('GET', '/projects/{project_id:\d+}', views.project)
```

Named routes, reverse constructing, and redirect

In **app.py**,

```
app.router.add_route('GET', '/projects', views.projects, name='projects')
app.router.add_route('POST', '/projects', views.add_project)
```

Named routes, reverse constructing, and redirect

In **views.py**,

```
@asyncio.coroutine
def add_project(request):
    data = yield from request.post()
    ...
    url = request.app.router['projects'].url()
    return web.HTTPFound(url)

@asyncio.coroutine
def projects(request):
    ...
```

You don't need to `from app import app`

- Or `from flask import current_app` either
- Request contains `app` instance for all your needs

In **app.py**,

```
from . import settings
...
app['settings'] = settings
```

You don't need to `from app import app`

In **views.py**,

```python
@asyncio.coroutine
def search(request):
    settings = request.app['settings']
    query = request.GET['query']
    locale = request.GET.get('locale', settings.DEFAULT_LOCALE)
    ...
    return web.Response(...)
```

Middlewares

- `web.Application` accepts optional middlewares factories sequence
- Middleware Factory should be a coroutine and returns a coroutine

In **app.py**,

```
@asyncio.coroutine
def trivial_middleware(app, handler):
    @asyncio.coroutine
    def middleware(request):
        return (yield from handler(request))
    return middleware

...

app = web.Application(middlewares=[trivial_middleware])
```

Middlewares

Ready to use Middlewares

- User Sessions, `aiohttp_session`
- Debug Toolbar, `aiohttp_debugtoolbar`

In **app.py**,

```
import aiohttp_debugtoolbar
from aiohttp_debugtoolbar import toolbar_middleware_factory
from aiohttp_session import session_middleware
from aiohttp_session.cookie_storage import EncryptedCookieStorage

app = web.Application(middlewares=[
    toolbar_middleware_factory,
    session_middleware(EncryptedCookieStorage(b'1234567890123456'))
])
aiohttp_debugtoolbar.setup(app)
```

Middlewares

Handling Exceptions

In **app.py**,

```
@asyncio.coroutine
def errorhandler_middleware(app, handler):
    @asyncio.coroutine
    def middleware(request):
        try:
            return (yield from handler(request))
        except web.HTTPError as err:
            # As it special case we could pass ``err`` as second argument to
            # error handler
            return (yield from views.error(request, err))
    return middleware
```

User Sessions

- aiohttp_session
- Latest version: `0.1.1`
- Supports storing session data in encrypted cookie or redis
- Enabled by passing `session_middleware` to middleware factories sequence

In **views.py**,

```
from aiohttp_session import get_session

@asyncio.coroutine
def login(request):
    ...
    session = yield from get_session(request)
    session['user_id'] = user_id
    return web.Response(...)
```

Rendering Templates

Jinja2 & Mako supported via aiohttp_jinja2 & aiohttp_mako

Jinja2 Support

In **app.py**,

```
import aiohttp_jinja2
import jinja2

...

aiohttp_jinja2.setup(
    app,
    loader=jinja2.FileSystemLoader('/path/to/templates')
)
```

Rendering Templates

Jinja2 Support

In **views.py**,

```
import aiohttp_jinja2

@aiohttp_jinja2.template('index.html')
def index(request):
    return {'is_index': True}
```

Rendering Templates

Jinja2 Support

In **views.py**,

```python
from aiohttp_jinja2 import render_template

@asyncio.coroutine
def index(request):
    return render_template('index.html', request, {'is_index': True})
```

Rendering JSON

In **utils.py**,

```
import ujson

from aiohttp import web

def json_response(data, **kwargs):
    # Sometimes user needs to override default content type for JSON
    kwargs.setdefault('content_type', 'application/json')
    return web.Response(ujson.dumps(data), **kwargs)
```

Note: I recommend to use ujson for work with JSON data in Python, cause of speed.

Rendering JSON

In **views.py**,

```
from .utils import json_response

@asyncio.coroutine
def api_browser(request):
    return json_response({
        'projects_url': request.app.router['projects'].url(),
    })
```

Serving Static Files

Important: It's highly recommend to use nginx, Apache, or other web server for serving static files in production.

In **app.py**,

```
app.router.add_static('/static', '/path/to/static', name='static')
```

Serving Static Files

In **views.py**,

```
@aiohttp_jinja2.template('index.html')
def index(request):
    return {'app': request.app, 'is_index': True}
```

Serving Static Files

In **index.html**,

```
<script src="{{ app.router.static.url(filename="dist/js/project.js") }}"
        type="text/javascript"></script>
```

And More

- WebSockets
- Expect Header
- Custom Conditions for Routes Lookup
- Class Based Handlers

- And I say it again, **WebSockets**

aiopg

Basics

- Supports SQLAlchemy core (functional SQL layer)
- So you able to describe tables with SQLAlchemy and then execute queries
- Maybe you will miss ORM, but man, ORM is overrated :)

- Also implements asyncio DBAPI like interface for PostgreSQL

Basics

```
import sqlalchemy as sa

metadata = sa.MetaData()

users_table = sa.Table(
    'users',
    metadata,

    sa.Column('id', sa.Integer, primary_key=True),
    sa.Column('email', sa.Unicode(255), unique=True),
    sa.Column('display_name', sa.Unicode(64), nullable=True),
)
```

Basics

```
import asyncio

from aiopg.sa import create_engine

@asyncio.coroutine
def get_user(dsn, user_id):
    engine = yield from create_engine(dsn)
    with (yield from engine) as conn:
        query = users_table.select(users_table.c.id == user_id)
        return (yield from (yield from conn.execute(query)).first())

loop = asyncio.get_event_loop()
user = loop.run_until_complete(get_user('postgres://...', 1))
print(user)
loop.close()
```

Integration with aiohttp.web

- Common practice is describing tables and place functions to manipulate data in one place, e.g., `storage.py` module
- Put `_table` suffix to every described table, e.g., `users_table`, not just `users`
- Connect to database in middleware factory
- Store database connection in *main* app instance to avoid overflow

Integration with aiohttp.web

In **app.py**,

```
@asyncio.coroutine
def db_middleware(app, handler):
    @asyncio.coroutine
    def middleware(request):
        db = app.get('db')
        if not db:
            app['db'] = db = yield from create_engine(app['dsn'])
        request.app['db'] = db
        return (yield from handler(request))
    return middleware

app = web.Application(middlewares=[db_middleware])
app['dsn'] = 'postgres://user:password@127.0.0.1:5432/database'
app.router.add_route('GET', '/users/{user_id:\d+}', views.user_profile)
```

Integration with aiohttp.web

In **views.py**,

```
from .storage import users_table

@aiohttp_jinja2.template('user_profile.html')
def user_profile(request):
    user_id = request.match_info['user_id']

    with (yield from request.app['db']) as conn:
        query = users_table.select(users_table.c.id == user_id)
        user = yield from (yield from conn.execute(query)).first()

    if not user:
        raise web.HTTPNotFound()

    return {'user': user}
```

More?

- You have SQLAlchemy under the hood
- And SQLAlchemy still saving the world

aioredis / asyncio_redis

Real World Usage

Structure

yourproject

- **api**
  - storage.py
  - views.py
- **auth**
  - api.py
  - views.py
- **static**
- **templates**
- app.py
- settings.py
- storage.py
- views.py

Add Route Context Manager

```python
from contextlib import contextmanager

@contextmanager
def add_route_context(app, views, url_prefix=None, name_prefix=None):
    def add_route(method, url, name):
        view = getattr(views, name)
        url = ('/'.join((url_prefix.rstrip('/'), url.lstrip('/')))
               if url_prefix
               else url)
        name = '.'.join((name_prefix, name)) if name_prefix else name
        return app.router.add_route(method, url, view, name=name)
    return add_route
```

Add Route Context Manager

In **app.py**,

```
from .api import views as api_views

with add_route_context(app, api_views, '/api', 'api') as add_route:
    add_route('GET', '/', 'index')
    add_route('GET', '/projects', 'projects')
    add_route('POST', '/projects', 'add_project')
    add_route('GET', '/project/{project_id:\d+}', 'project')
    add_route('PUT', '/project/{project_id:\d+}', 'edit_project')
    add_route('DELETE', '/project/{project_id:\d+}', 'delete_project')
```

Add Route Context Manager

In **api/views.py**,

```
import asyncio

from ..utils import json_response

@asyncio.coroutine
def index(request):
    router = request.app.router

    project_url = router['api.proejct'].url(parts={'project_id': 42})
    project_url = project_url.replace('42', '{project_id}')

    return json_response({
        'urls': {
            'add_project': router['api.add_project'].url(),
            'project': project_url,
            'projects': router['api.projects'].url(),
        }
    })
```

Immutable Settings

- Python 3.3+ has `MappingTypeProxy`
- Settings shouldn't be changed across the app
- Welcome, rororo.settings

In **app.py**,

```
from rororo.settings import immutable_settings

from . import settings

def create_app(**options):
    settings_dict = immutable_settings(settings, **options)
    app = web.Application()
    app['settings'] = settings_dict
    ...
    return app
```

Other Settings & Logging Helpers

rororo.settings

- `inject_settings`
- `setup_locale`
- `setup_logging`
- `setup_timezome`
- `to_bool`

rororo.logger

- `default_logging_dict`
- `update_sentry_logging`

Supports DEBUG Mode from Settings

```
class Application(web.Application):

    def __init__(self, **kwargs):
        self['settings'] = kwargs.pop('settings')
        super().__init__(**kwargs)

    def make_handler(self, **kwargs):
        kwargs['debug'] = self['settings']['DEBUG']
        kwargs['loop'] = self.loop
        return self._handler_factory(self, self.router, **kwargs)
```

Schemas

- You need to validate request & response data
- Use JSON Schema for validation
- Describe Schemas with jsl
- Welcome, rororo.schemas

In **api/views.py**,

```
from rororo.schemas import Schema

from . import schemas

@asyncio.coroutine
def add_project(request):
    schema = Schema(schemas.add_project, response_factory=json_response)
    data = schema.validate_request((yield from request.post()))
    ...
    return schema.make_response(project_dict)
```

Schemas

Describing Schemas

In **api/schemas/add_project.py**,

```
from jsl import Document, IntegerField, StringField

class Project(Document):
    name = StringField(min_length=1, max_length=32, required=True)
    slug = StringField(min_length=1, max_length=32, required=True)
    description = StringField(max_length=255)

class Response(Project):
    id = IntegerField(minimum=0, required=True)

request = Project.get_schema()
response = Response.get_schema()
```

Or use plain Python dicts instead

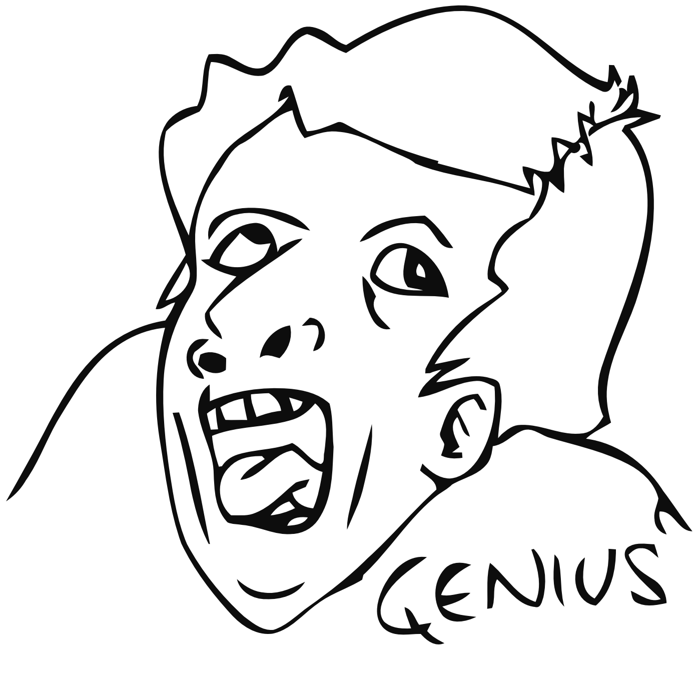
Schemas

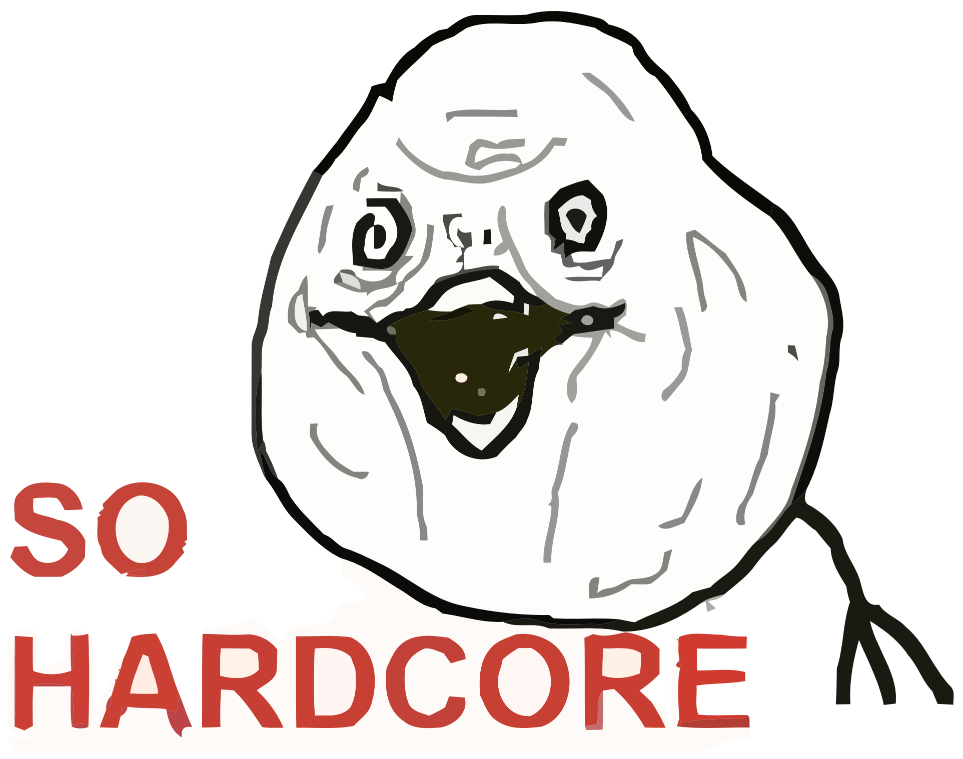
Describing Schemas

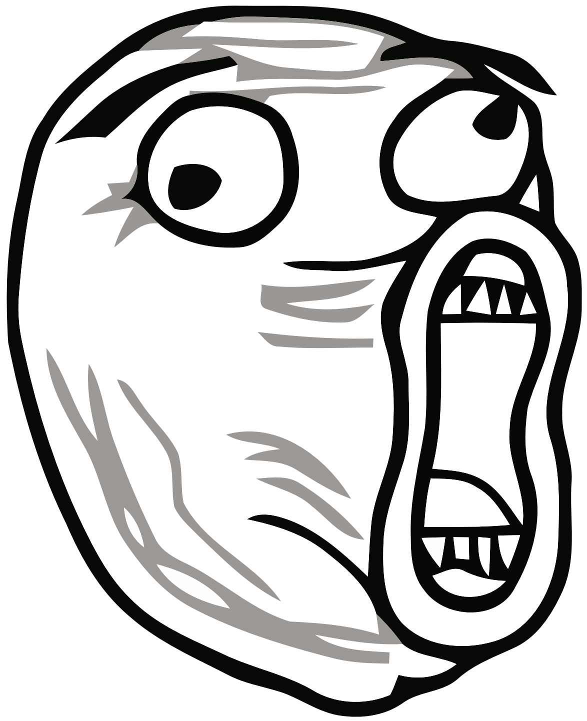
In **api/schemas/__init__.py**,

```
from . import add_project  # noqa
```

Migration from Django


GENIUS


SO
HARDCORE



LOL

Um, Really?

- I don't believe in this
- Django and aiohttp.web are frameworks for different tasks
- You would miss everything you know

Migration from Flask

CHALLENGE ACCEPTED

My Success Story

I migrated Python backend from Flask to aiohttp.web

- 1 working day
- Quick Notes:
    - *Global* state -> read all from request
    - Extensions -> ?
    - Blueprints -> list of routes or custom router
    - Flask before/after request -> aiohttp.web middleware
    - Error handlers -> middleware
    - `{{ url_for(...) }}` -> `{{ app.router[...].url(...) }}`

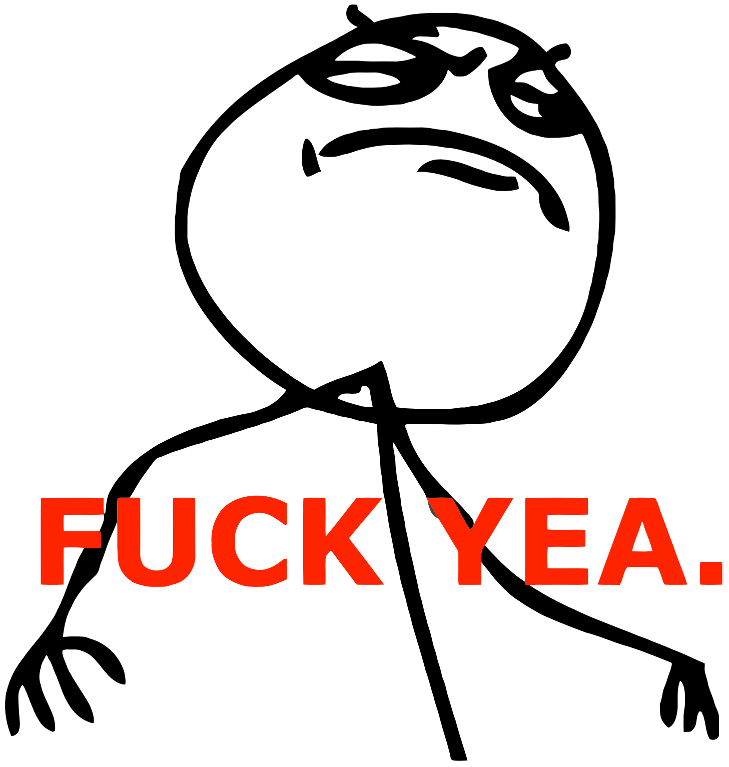
Other Thoughts

- I was a **huge** Flask fanboy
- http://igordavydenko.com/talks/ua-pycon-2012.pdf

- aiohttp.web solves my problem with large Flask applications
- I don't need to use *lazy views* concept to keep my large apps Pythonic

- aiohttp.web solves my problem with *global* state and contexts
- Everything read from the request. It's just works

- **I'm really tired of Armin's complaints about Python 3**


FUCK YEA.

Summary

Asyncio Stack is ready for usage

- **The Future is Here!**
- Use Python 3.4 for all good things

- `aiohttp.web`, easy to start and easy to use
- `aiopg.sa` allows you to forget about ORM
- `rororo` contains useful helpers for aiohttp.web apps

- If you miss something, port it to Asyncio stack by **yourself**
- Don't forget to payback to Open Source Software

Questions?