Welcome to async/await era

Igor Davydenko
PyCon Finland 2015

I am...

- **Igor Davydenko**
- Python & React.js developer
- From Kyiv, Ukraine
- Works on Ezhome Inc.
- Primarly designs & develops backend API

- Personal Site
- @ GitHub
- @ Twitter

PEP 492

Coroutines with async and await syntax

Quick Summary

- Created by **Yuri Selivanov** in April, 2015
- Included in Python 3.5
- Added 4 new statements to the standard library:
  - `async def`
  - `await`
  - `async with`
  - `async for`
- Received great response from the Python community
- PEP 492 @ Python.org

@asyncio.coroutine

```
import asyncio

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

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

async def

```
import asyncio

async def hello():
    return 'Hello, world!'

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

@asyncio.coroutine + yield from

```
import asyncio
from aiohttp import client

@asyncio.coroutine
def fetch_page(url):
    response = yield from client.get(url)
    return (yield from response.text())

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

async def + await

```
import asyncio
from aiohttp import client

async def fetch_page(url):
    response = await client.get(url)
    return await response.text()

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

with (yield from ...)

```
import asyncio
import sqlalchemy as sa
from aiopg.sa import create_engine

@asyncio.coroutine
def count_data(dsn):
    engine = yield from create_engine(dsn)
    with (yield from engine) as conn:
        query = sa.select(...).count()
        return (yield from conn.scalar(query))

loop = asyncio.get_event_loop()
counter = loop.run_until_complete(count_data('postgresql://...'))
print(counter)
loop.close()
```

async with

```
import asyncio
import sqlalchemy as sa
from aiopg.sa import create_engine
from .utils import ConnectionContextManager

async def count_data(dsn):
    engine = await create_engine(dsn)
    async with ConnectionContextManager(engine) as conn:
        query = sa.select(...).count()
        return await conn.scalar(query)

loop = asyncio.get_event_loop()
counter = loop.run_until_complete(count_data('postgresql://...'))
print(counter)
loop.close()
```

async with

utils.py

```
class ConnectionContextManager(object):

    def __init__(self, engine):
        self.conn = None
        self.engine = engine

    async def __aenter__(self):
        self.conn = await self.engine.acquire()
        return self.conn

    async def __aexit__(self, exc_type, exc, tb):
        try:
            self.engine.release(self.conn)
        finally:
            self.conn = None
            self.engine = None
```

for row in (yield from ...):

```
import asyncio
import sqlalchemy as sa
from aiopg.sa import create_engine

@asyncio.coroutine
def fetch_data(dsn):
    data = []
    engine = yield from create_engine(dsn)
    with (yield from engine) as conn:
        result = yield from conn.execute(sa.select(...))
        for row in result:
            data.append(row)
    return data

loop = asyncio.get_event_loop()
data = loop.run_until_complete(fetch_data('postgresql://...'))
loop.close()
```

async for

```
import asyncio
import sqlalchemy as sa
from aiopg.sa import create_engine
from .utils import ConnectionContextManager, ResultIter

async def fetch_data(dsn):
    data = []
    engine = await create_engine(dsn)
    async with ConnectionContextManager(engine) as conn:
        async for row in ResultIter(await conn.execute(sa.select(...))):
            data.append(row)
    return data

loop = asyncio.get_event_loop()
data = loop.run_until_complete(fetch_data('postgresql://...'))
loop.close()
```

async for

utils.py

```
from aiopg.sa.exc import ResourceClosedError

class ResultIter(object):

    def __init__(self, result):
        self.result = result

    async def __aiter__(self):
        return self

    async def __anext__(self):
        try:
            data = await self.result.fetchone()
        except ResourceClosedError:
            data = None
        if data:
            return data
        raise StopAsyncIteration
```

Other additions to standard library

- `@types.coroutine` bridge between generator based and native coroutines
- New `__await__` magic method
- New functions in `inspect` library as `iscoroutine`, `isawaitable`, etc
- New abstract base classes: `abc.Awaitable`, `abc.Coroutine`, `abc.AsyncIterable`, `abc.AsyncIterator`

Conclusion on functions and methods

Method	Can contain	Can't contain
`async def func`	`await, return value`	`yield, yield from`
`async def __a*__`	`await, return value`	`yield, yield from`
`def __a*__`	`return awaitable`	`await`
`def __await__`	`yield, yield from, return iterable`	`await`
`generator`	`yield, yield from, return value`	`await`

async/await in real life

Libraries

aiohttp

- HTTP client/server for asyncio
- Latest version: `0.17.4`
- http://aiohttp.readthedocs.org/

```
import asyncio
import ujson
from aiohttp import client

async def github_search(query):
    response = await client.get('https://api.github.com/search/repositories',
                                params={'q': query})
    return ujson.loads(await response.read())

loop = asyncio.get_event_loop()
response = loop.run_until_complete(github_search('asyncio'))
print('\n'.join(repo['full_name'] for repo in response['items']))
loop.close()
```

aiohttp.web

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

```
import ujson
from aiohttp import web

async def api_index(request):
    output = ujson.dumps({...})
    return web.Response(body=output, content_type='application/json')

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

```
$ 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

async def select_one(dsn):
    pool = await create_pool(dsn)
    with (await pool.cursor()) as cursor:
        await cursor.execute('SELECT 1')
        selected = await cursor.fetchone()
        assert selected == (1, )

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

aiopg.sa

- Layer for executing SQLAlchemy Core queries
- http://aiopg.readthedocs.org/en/stable/sa.html

```
import asyncio
import sqlalchemy as sa
from aiopg.sa import create_engine
from .utils import ConnectionContextManager

async def select_one(dsn):
    engine = await create_engine(dsn)
    async with ConnectionContextManager(engine) as conn:
        await conn.execute(sa.select([sa.text('1')]))
        selected = await conn.fetchone()
        assert selected == (1, )

loop = asyncio.get_event_loop()
loop.run_until_complete(select_one('postgresql://...'))
loop.close()
```

aioredis

- Redis client library
- Latest version: `0.2.4`
- http://aioredis.readthedocs.org/

```
import asyncio
from aioredis import create_redis

async def redis_set_get_delete(address, **options):
    options.setdefault('encoding', 'utf-8')
    redis = await create_redis(address, **options)
    assert await redis.set('key', 'value') is True
    assert await redis.get('key') == 'value'
    assert await redis.delete('key') == 1

loop = asyncio.get_event_loop()
loop.run_until_complete(redis_set_get_delete(('localhost', 6379)))
loop.close()
```

And others

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

async/await in real life

Fetching data from remote API

Task. Fetch data for the NFL Season

- NFL Season lasts *5 preseason* and *17 regular* season weeks
- This totals *22 requests* to NFL.com endpoint
- Each request can be processed asyncronously
- After all requests are done we need to call extra function

Step 1. Sync fetch week data

```
import requests
from lxml import etree

NFL_URL = 'http://www.nfl.com/ajax/scorestrip'

def fetch_week(season, week, is_preseason=False):
    response = requests.get(NFL_URL, params={
        'season': season,
        'seasonType': 'PRE' if is_preseason else 'REG',
        'week': week,
    })
    return etree.fromstring(response.content)
```

Step 2. Sync fetch season data

```
def fetch_season(season):
    pre_call(...)

    for is_preseason, weeks in ((False, range(5)), (True, range(1, 18))):
        for week in weeks:
            fetch_week(season, week, is_preseason)
            ...

    post_call(...)
```

Step 3. Async fetch week data

```
from aiohttp import client
from lxml import etree

async def aio_fetch_week(season, week, is_preseason=False):
    response = await client.get(NFL_URL, params={
        'season': season,
        'seasonType': 'PRE' if is_preseason else 'REG',
        'week': week,
    })
    return etree.fromstring(await response.read())
```

Step 4. Async fetch season data

```
import asyncio

def fetch_season(season):
    loop = asyncio.get_event_loop()
    loop.run_until_complete(pre_call(...))

    tasks = [
        aio_fetch_week(season, year, is_preseason)
        for is_preseason, weeks in ((False, range(5)), (True, range(1, 18)))
        for week in weeks
    ]
    loop.run_until_complete(asyncio.wait(tasks, loop=loop))

    loop.run_until_complete(post_call(...))
    loop.close()
```

Step 5. Running fetching season data

scripts/fetch_season_data.py

```
import sys

def main(*args):
    fetch_season(int(args[0]))
    return False

if __name__ == '__main__':
    sys.exit(int(main(*sys.argv[1:])))
```

```
$ python scripts/fetch_season_data.py 2015
```

Step 6. See in action

```
...
All games for Season 2015/REG. Week 17 processed!
Season 2015/REG. Week 12. Game ID 2015113000, CLE @ BAL, scheduled at 2015-11-30T20:30:00-05:00 updated in database
All games for Season 2015/REG. Week 12 processed!
Season 2015/PRE. Week 4. Game ID 2015090361, DAL @ HOU, scheduled at 2015-09-03T20:00:00-04:00 updated in database
Season 2015/REG. Week 16. Game ID 2015122711, SEA @ STL, scheduled at 2015-12-27T16:25:00-05:00 updated in database
Season 2015/PRE. Week 4. Game ID 2015090362, STL @ KC, scheduled at 2015-09-03T20:00:00-04:00 updated in database
Season 2015/REG. Week 16. Game ID 2015122712, BAL @ PIT, scheduled at 2015-12-27T20:30:00-05:00 updated in database
Season 2015/PRE. Week 4. Game ID 2015090363, TEN @ MIN, scheduled at 2015-09-03T20:00:00-04:00 updated in database
Season 2015/REG. Week 16. Game ID 2015122800, DEN @ CIN, scheduled at 2015-12-28T20:30:00-05:00 updated in database
All games for Season 2015/REG. Week 16 processed!
Season 2015/PRE. Week 4. Game ID 2015090364, DEN @ ARI, scheduled at 2015-09-03T21:00:00-04:00 updated in database
Season 2015/PRE. Week 4. Game ID 2015090365, SF @ SD, scheduled at 2015-09-03T22:00:00-04:00 updated in database
Season 2015/PRE. Week 4. Game ID 2015090366, SEA @ OAK, scheduled at 2015-09-03T22:00:00-04:00 updated in database
All games for Season 2015/PRE. Week 4 processed!
...
```

async/await in real life

Backend API application

Task. Provide an API for week data

- NFL.com still returns data in `XML`
- We need this data in `GraphQL`
- Okay, okay, actually in `JSON`
- And if there are no live games, we don't need to fetch data directly from NFL

Step 0. Original NFL.com data


```
▼<ss>
 ▼<gms gd="0" w="6" y="2015" t="R">
    <g eid="2015101500" gsis="56580" d="Thu" t="8:25" q="F" k="" h="NO" hnn="saints" hs="31" v="ATL" vnn="falcons" vs="21" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101801" gsis="56586" d="Sun" t="1:00" q="F" k="" h="NYJ" hnn="jets" hs="34" v="WAS" vnn="redskins" vs="20" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101802" gsis="56587" d="Sun" t="1:00" q="F" k="" h="PIT" hnn="steelers" hs="25" v="ARI" vnn="cardinals" vs="13" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101803" gsis="56585" d="Sun" t="1:00" q="F" k="" h="MIN" hnn="vikings" hs="16" v="KC" vnn="chiefs" vs="10" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101804" gsis="56581" d="Sun" t="1:00" q="F" k="" h="BUF" hnn="bills" hs="21" v="CIN" vnn="bengals" vs="34" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101800" gsis="56583" d="Sun" t="1:00" q="FO" k="" h="DET" hnn="lions" hs="37" v="CHI" vnn="bears" vs="34" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101805" gsis="56582" d="Sun" t="1:00" q="FO" k="" h="CLE" hnn="browns" hs="23" v="DEN" vnn="broncos" vs="26" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101806" gsis="56588" d="Sun" t="1:00" q="F" k="" h="TEN" hnn="titans" hs="10" v="MIA" vnn="dolphins" vs="38" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101807" gsis="56584" d="Sun" t="1:00" q="F" k="" h="JAC" hnn="jaguars" hs="20" v="HOU" vnn="texans" vs="31" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101808" gsis="56589" d="Sun" t="4:05" q="F" k="" h="SEA" hnn="seahawks" hs="23" v="CAR" vnn="panthers" vs="27" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101809" gsis="56590" d="Sun" t="4:25" q="F" k="" h="GB" hnn="packers" hs="27" v="SD" vnn="chargers" vs="20" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101810" gsis="56591" d="Sun" t="4:25" q="F" k="" h="SF" hnn="49ers" hs="25" v="BAL" vnn="ravens" vs="20" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101811" gsis="56592" d="Sun" t="8:30" q="F" k="" h="IND" hnn="colts" hs="27" v="NE" vnn="patriots" vs="34" p="" rz="" ga="" gt="REG"/>
    <g eid="2015101900" gsis="56593" d="Mon" t="8:30" q="P" k="" h="PHI" hnn="eagles" hs="" v="NYG" vnn="giants" vs="" p="" rz="" ga="" gt="REG"/>
  </gms>
 </ss>
```

Step 1. Flask base application

```
from flask import Flask
from . import cache
from .data import to_game_data

app = Flask(__name__)

@app.route('/api/season/<int:season_year>/<week_slug>')
def retrieve_week(season_year, week_slug):
    is_preseason = week_slug[:4] == 'pre-'
    week = int(week_slug[4:] if is_preseason else week_slug)

    cache_key = cache.build_key(season_year, week_slug)
    week_data = cache.ensure_data(cache_key)

    if week_data is None:
        week_obj = fetch_week(season_year, week, is_preseason)
        week_data = [to_game_data(game) for game in week_obj.iterfind('gms/g')]
        cache.store_data(cache_key, week_data)

    return jsonify(week_data)
```

Step 2. Running Flask application

```
$ gunicorn -k sync -w 5 -t 60 app:app
```

```
$ gunicorn -k eventlet -w 5 -t 60 app:app
```

Step 3. aiohttp.web base application

```
import ujson
from aiohttp import web

async def aio_retrieve_week(request):
    season_year = int(request.match_info['season_year'])
    week_slug = request.match_info['week_slug']

    is_preseason = week_slug[:4] == 'pre-'
    week = int(week_slug[4:] if is_preseason else week_slug)

    cache_key = cache.build_key(season_year, week, is_preseason)
    week_data = await cache.aio_ensure_data(cache_key)

    if week_data is None:
        week_obj = await aio_fetch_week(season_year, week, is_preseason)
        week_data = [to_game_data(game) for game in week_obj.iterfind('gms/g')]
        await cache.store_data(cache_key, week_data)

    return web.Response(ujson.dumps(week_data),
                        content_type='application/json')
```

Step 3. aiohttp.web base application (continue)

```
aio_app = web.Application()
aio_app.router.add_route('GET',
                         '/api/season/{season_year:\d{4}}/{week_slug}',
                         aio_retrieve_week)
```

aiohttp.web basics

- Each view function should be a `coroutine/async def` and return response or raise an exc
- Each view function receives only one arg: `request`
- Query params data contains in `request.GET` multidict
- Body data contains in `request.POST` multidict and **should be** awaited with `await request.post()`
- Matched data contains in `request.match_info` dict
- `request` also contains an `app` instance

Step 4. Runinng aiohttp.web application

```
$ gunicorn -k aiohttp.worker.GunicornWebWorker -w 5 -t 60 app:aio_app
```

```
- games: [
    - {
        id: 2015101500,
      - home: [
            "no",
            31
        ],
        is_ended: true,
        scheduled_at: "2015-10-16T00:25:00+00:00",
      - away: [
            "atl",
            21
        ],
        game_time: "FT"
    },
    - {
        id: 2015101800,
      - home: [
            "det",
            37
        ],
        is_ended: true,
        scheduled_at: "2015-10-18T17:00:00+00:00",
      - away: [
            "chi",
            34
        ],
        game_time: "FOT"
    },
```

async/await in real life

Server-Sent Events

Task. Livescore stream

- When there are live games, update the scores automagically for the user
- Do not reconnect each X seconds for new scores
- When new scores available - show them to user
- This is the case for Server-Sent Events (SSE)

Server-Sent Events. Message Format

```
data: Hello, PyCon Finland!
<blank line>
```

```
event: pyconfi
data: Hello, Pycon Finland!
data: I'm glad to be here
<blank link>
```

```
id: 42
event: uapycon
data: PyCon Ukraine 2016 will held a place in Lviv at April 2016
<blank link>
```

Server-Sent Events. EventSource

```
const element = document.getElementById("notifications-container");
const source = new EventSource("/api/notifications");
source.onerror = err => {
  ...
};
source.addEventListener("pyconfi", evt => {
  element.innerHTML += `<code>#pyconfi</code> ${evt.data}<br>`;
});
source.addEventListener("uapycon", evt => {
  element.innerHTML += `<code>#uapycon</code> ${evt.data}<br>`;
});
...
source.close();
```

How to implement?

1. Run script to fetch live scores and publish new scores to Redis PUBSUB
2. Subscribe to this channel in livescore stream view, receive message and push it to client
3. Subscribe to livescore stream in client and update scores when new message received
4. …
5. PROFIT

Step 1. Fetching & publishing scores change

```
def main(*args):
    season, week = args

    loop = asyncio.get_event_loop()

    week_obj = loop.run_until_complete(aio_fetch_week(season, week, False))
    week_data = [to_game_data(game) for game in week_obj.iterfind('gms/g')]

    redis = loop.run_until_complete(create_redis(('localhost', 6379)))
    loop.run_until_complete(redis.publish('CHANNEL', ujson.dumps(week_data)))

    loop.close()
    return False

if __name__ == '__main__':
    sys.exit(int(main(*sys.argv[1:])))
```

Step 2. Implementing livescore stream

```
async def livescore(request):
    if request.headers['Accept'] != 'text/event-stream':
        raise web.HTTPFound('/api/')

    response = web.StreamResponse(status=200, headers={
        'Content-Type': 'text/event-stream',
        'Cache-Control': 'no-cache',
        'Connection': 'keep-alive',
    })
    response.start(request)  # This will be changed in asyncio==0.18.0

    ...

    return response
```

Step 2. Subscribing to scores update

```
...

redis = await create_redis(('localhost', 6379))
channel = (await redis.subscribe('CHANNEL'))[0]

while (await channel.wait_message()):
    message = await channel.get()
    response.write(b'event: update\r\n')
    response.write(b'data: ' + message + b'\r\n\r\n')

...
```

Step 2. Registering stream to the app

```
aio_app.router.add_route('GET', '/api/livescore', livescore)
```

Step 3. Subscribing to livescore stream on client

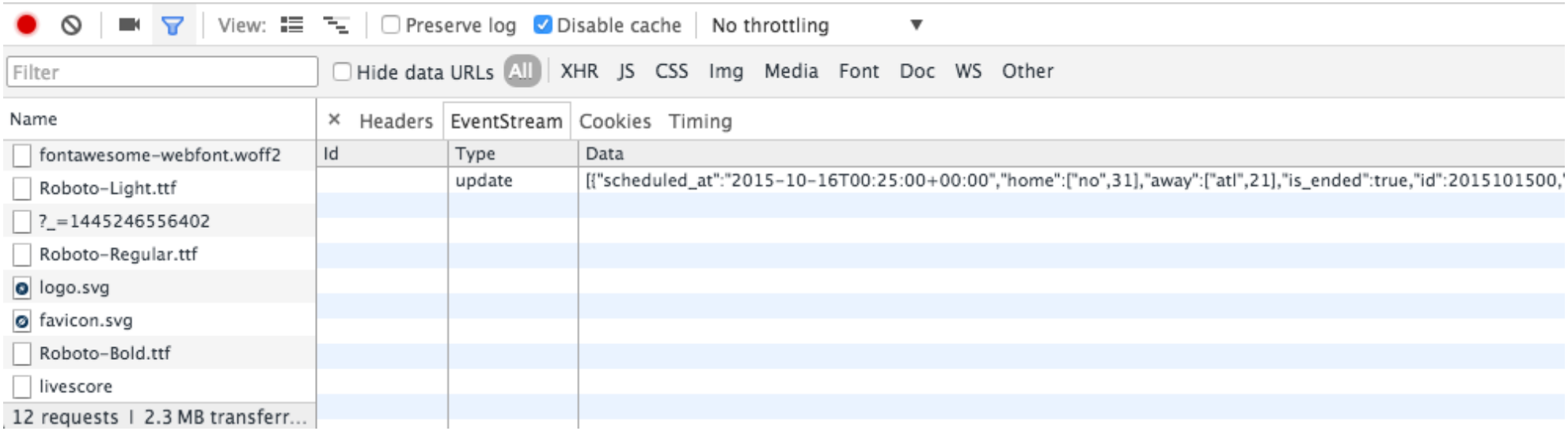
```
import React, {Component} from "react";

export default App extends Component {
  liveScore = null;
  constructor(props) {
    super(props);
    this.state = {games: props.games};
  }
  componentDidMount() {
    this.liveScore = new EventSource("/api/");
    this.liveScore.addEventListener("update", evt => {
        this.setState({games: JSON.parse(evt.data)});
    });
  }
  componentWillUnmount() {
    this.liveScore.close();
  }
  render() {
    return <Games data={this.state.games} />;
  }
}
```

Step 4. Setting up the nginx

```
location /api/livescore {
    chunked_transfer_encoding off;
    proxy_buffering off;
    proxy_cache off;
    proxy_http_version 1.1;
    proxy_pass http://127.0.0.1:8000;
    proxy_set_header Connection '';
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
```

Step 5. It works



async/await era

Conclusion

The Future is Here

- Python 3.5 released and ready to be used in production
- Python 3.5 contains new `async/await` statements
- `asyncio` stack got great addition, code now is simple to read and understand
- **asyncio stack** (for web applications):
  - `aiohttp` – interact with remote API
  - `aiohttp.web` – create API backends
  - `aiopg` – interact with PostgreSQL (DB)
  - `aioredis` – interact with Redis (cache)

The Future is Here

- `asyncio` stack is a great reason why you'll finally need to switch to Python 3
- **`asyncio` is good fit for:**
  - Database-less API
  - Requesting remote API
  - Async code execution
  - Predictable async I/O
- And I'm not telling you about WebSockets, which supported by `aiohttp` out of the box

And the real reason of using asyncio

Cause it cool and trendy!

Questions?

Bonus. PyCon Ukraine 2016