



Tornado

Asbathou Biyalou-Sama
Jérémy Vienne
Franco Davy Irakoze

01 OVERVIEW

02 CONCEPTS

03 CODE WITH
TORNADO

04 PRACTICE

05 FEED BACK

01. OVERVIEW

ORIGINES

- ❑ Created in 2009 by FriendFeed
 - ❑ Open Source (Licence Apache 2), written in Python
 - ❑ Acquired by Facebook



WHICH COMPANIES USE TORNADO?



zalando



uploadcare



Keen

02. CONCEPTS

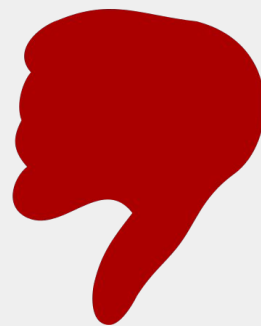
CONCEPTS

Tornado is based on the following major components:

- ❑ A Web Framework
- ❑ Asynchronous library
- ❑ Client / Server HTTP
- ❑ Coroutine library
- ❑ DIY Framework (Do It Yourself)



- ❑ Many simultaneous connections
- ❑ Performances
- ❑ Lightweight
- ❑ Ideal for app needing long time connection



- ❑ Tornado store upload files in memory
- ❑ Smaller community than Django
- ❑ Create all by yourself

PYTHON ALTERNATIVE TO TORNADO ?



- ❑ DIY Framework
- ❑ Back end Framework
- ❑ Lightweight

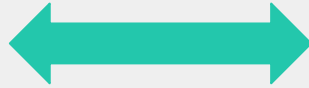


- ❑ DIY Framework
- ❑ Full Stack Framework
- ❑ Long polling and async app



- ❑ Complete Framework
- ❑ Full Stack Framework
- ❑ Big project

SIMILAR FRAMEWORK?



- ❑ Asynchronous Framework
- ❑ Non-blocking I/O
- ❑ Real-time Web app

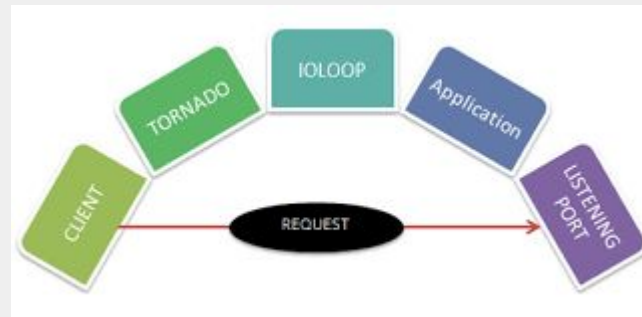
ONE KEY POINT : ASYNCHRONOUS IN TORNADO

- ❑ Python is natively single-threaded language
 - ❑ All resources for One thing at a time
- ❑ In Web, Requests come at any time
 - ❑ How to handle all this !?

03. CODE WITH TORNADO

BASIC IMPLEMENTATION

- ❑ Application
 - ❑ Defines routes
 - ❑ Configures initial state (e.g: connexion to a database, others web services)
- ❑ HTTP Server
 - ❑ enables HTTP access to the application
- ❑ IO Loop
 - ❑ Needs to be started so that the web server remains listening
- ❑ Request Handlers



APPLICATION

```
from tornado.httpserver import HTTPServer
from tornado.ioloop import IOLoop
from tornado.options import define, options
from tornado.web import Application

define(['port', default=8888, help='port to listen on'])
def main():
    """Construct and serve the tornado application."""
    app = Application([
        (r"/", basicRequestHandler),
        (r"/blog", staticRequestHandler),
        (r"/isEven", queryStringRequestHandler),
        (r"/tweet/([0-9]+)", resourceRequestHandler)])
    http_server = HTTPServer(app)
    http_server.listen(options.port)
    print('Listening on http://localhost:%i' % options.port)
    IOLoop.current().start()
```

REQUEST HANDLER

```
from tornado.web import RequestHandler

class basicRequestHandler(RequestHandler):
    """Print 'Hello, world!' as the response body."""

    def get(self):
        """Handle a GET request for saying Hello World!."""
        self.write("Hello, world!")
```

DEFINES ROUTES

`(r"/", basicRequestHandler)`

`class basicRequestHandler(tornado.web.RequestHandler):`
`def get(self):`

`(r"/id", queryStringRequestHandler)`

`class queryStringRequestHandler(tornado.web.RequestHandler):`
`def get(self):`
 `n = int(self.get_argument("n"))`

`(r"/tweet/([0-9]+)", resourceRequestHandler)`

`class resourceRequestHandler(tornado.web.RequestHandler):`
`def get(self, id):`

HOW TO MANAGE VIEWS ?

- ❑ Templates are managed in the same way as in Django and Flask

Render Integration :

```
class HomeHandler(tornado.web.RequestHandler):  
    def get(self):  
        entries = self.db.query("SELECT * FROM entries ORDER BY date DESC")  
        self.render("home.html", entries=entries)
```

```
app = tornado.web.Application(  
    [  
    ],  
    cookie_secret="__TODO: GENERATE YOUR OWN RANDOM VALUE HERE__",  
    template_path=os.path.join(os.path.dirname(__file__), "templates"),  
    static_path=os.path.join(os.path.dirname(__file__), "static"),  
    debug=options.debug,  
)  
app.listen(options.port)
```

Template view :

```
<html>  
  <head>  
    <title>{{ title }}</title>  
  </head>  
  <body>  
    <ul>  
      {% for item in items %}  
        <li>{{ escape(item) }}</li>  
      {% end %}  
    </ul>  
  </body>  
</html>
```

TEMPLATES POSSIBILITIES

Complex expression translated to Python :

```
{% for student in [p for p in people if p.student and p.age > 23] %}  
  <li>{{ escape(student.name) }}</li>  
{% end %}
```

Use method :

```
### Python code  
def add(x, y):  
    return x + y  
template.execute(add=add)  
  
### The template  
{{ add(1, 2) }}
```

Useful Syntax :

```
{% set *x* = *y* %}
```

```
{% for *var* in *expr* %}...{% end %}
```

```
{% include *filename* %}
```

```
{% extends *filename* %}
```

```
{% block *name* %}...{% end %}
```

SYNC & ASYNC WITH TORNADO

Synchronous method :

```
from tornado.httpclient import HTTPClient

def synchronous_fetch(url):
    http_client = HTTPClient()
    response = http_client.fetch(url)
    return response.body
```

ASynchronous method :

```
from tornado.httpclient import AsyncHTTPClient

async def asynchronous_fetch(url):
    http_client = AsyncHTTPClient()
    response = await http_client.fetch(url)
    return response.body
```

COROUTINES

- ❑ Recommended for asynchronous code in tornado
- ❑ use the keywords : “await” or “yield”
- ❑ as simple as synchronous code
- ❑ easier concurrency : reduce the number of places where a context can switch

Decorated:

```
# Normal function declaration
# with decorator
@gen.coroutine
def a():
    # "yield" all async funcs
    b = yield c()
    # "return" and "yield"
    # cannot be mixed in
    # Python 2, so raise a
    # special exception.
    raise gen.Return(b)
```

Native:

```
# "async def" keywords
async def a():
    # "await" all async
    b = await c()

    # Return normally
    return b
```

HOW TO CALL COROUTINE

Bad way :

```
async def divide(x, y):  
    return x / y  
  
def bad_call():  
    # This should raise a ZeroDivisionError, but it won't because  
    # the coroutine is called incorrectly.  
    divide(1, 0)
```

Good Way :

```
async def good_call():  
    # await will unwrap the object returned by divide() and raise  
    # the exception.  
    await divide(1, 0)
```

Fire and forget method :

```
IOLoop.current().spawn_callback(divide, 1, 0)
```

COROUTINE PATTERNS

Parallelism :

```
from tornado.gen import multi

async def parallel_fetch(url1, url2):
    resp1, resp2 = await multi([http_client.fetch(url1),
                                |
                                |
                                |
                                |
                                |
                                |
                                http_client.fetch(url2)])

async def parallel_fetch_many(urls):
    responses = await multi ([http_client.fetch(url) for url in urls])
    # responses is a list of HTTPResponses in the same order

async def parallel_fetch_dict(urls):
    responses = await multi({url: http_client.fetch(url)
                             |
                             |
                             |
                             |
                             |
                             |
                             for url in urls})
    # responses is a dict {url: HTTPResponse}
```

COROUTINE PATTERNS

CALL BLOCKING METHOD :

```
async def call_blocking():  
    await IOloop.current().run_in_executor(None, blocking_func, args)
```

INTERLEAVING

```
async def get(self):  
    # convert_yielded() starts the native coroutine in the background.  
    # This is equivalent to asyncio.ensure_future() (both work in Tornado).  
    fetch_future = convert_yielded(self.fetch_next_chunk())  
    while True:  
        chunk = yield fetch_future  
        if chunk is None: break  
        self.write(chunk)  
        fetch_future = convert_yielded(self.fetch_next_chunk())  
        yield self.flush()
```

COROUTINE PATTERNS

LOOPING:

```
db = motor.MotorClient().test

@gen.coroutine
def loop_example(collection):
    cursor = db.collection.find()
    while (yield cursor.fetch_next):
        doc = cursor.next_object()
```

RUNNING IN THE BACKGROUND:

```
async def minute_loop():
    while True:
        await do_something()
        await gen.sleep(60)

# Coroutines that loop forever are generally started with
# spawn_callback().
IOloop.current().spawn_callback(minute_loop)
```


<https://gitlab.univ-lille.fr/francodavy.irakoze.etu/tp-tornado>

06. FEEDBACK

HOW TO RUN AND DEPLOY ?



BEGIN YOUR APP

DEPENDENCIES:

```
import tornado.ioloop
import tornado.web
```

RUNNING APP:

```
if __name__ == "__main__":
    app = make_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

CREATE SOME ROUTES

ROUTES:

```
app = tornado.web.Application([
    (r"/", basicRequestHandler),
    (r"/blog", staticRequestHandler),
    (r"/isEven", queryStringRequestHandler),
    (r"/tweet/([0-9]+)", resourceRequestHandler)
])
```

REQUEST HANDLERS:

```
class basicRequestHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world!!!!!!")

class resourceRequestHandler(tornado.web.RequestHandler):
    def get(self, id):
        self.write("Querying tweet with id " + id)

class queryStringRequestHandler(tornado.web.RequestHandler):
    def get(self):
        n = int(self.get_argument("n"))
        r = "odd" if n % 2 else "even"

        self.write("the number " + str(n) + " is " + r)
```