WHO AM I?
Setting up expectations

- This session IS a mind opener on how to provide added value with minimal effort.
- This session IS NOT about saying X is a bad technology and Y is a good technology.
- Assuming knowledge in web development and REST APIs in particular.
Motivation

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return {'hello': 'world'}

if __name__ == '__main__':
    app.run()
```

```python
from sanic import Sanic
from sanic.response import json

app = Sanic(__name__)

@app.route("/")
async def hello(_):
    return json({'hello': 'world'})

if __name__ == '__main__':
    app.run()
```
Motivation - cont.

Concurrency Level: 100
Time taken for tests: 13.247 seconds
Complete requests: 10000
Failed requests: 0
Total transferred: 1630000 bytes
HTML transferred: 180000 bytes
Requests per second: 754.88 [#/sec] (mean)
Time per request: 132.471 [ms] (mean)
Time per request: 1.325 [ms] (mean, across all
Transfer rate: 120.16 [Kbytes/sec] received

Concurrency Level: 100
Time taken for tests: 4.000 seconds
Complete requests: 10000
Failed requests: 0
Total transferred: 1070000 bytes
HTML transferred: 170000 bytes
Requests per second: 2499.90 [#/sec] (mean)
Time per request: 40.002 [ms] (mean)
Time per request: 0.400 [ms] (mean, across all
Transfer rate: 261.22 [Kbytes/sec] received

3-4x
Notes on the motivation experiment

- simplejson is installed as an optional dependency for flask.
- ab is used for benchmarking.
Flask

A micro web framework that revolutionized how web is developed with python.
Asyncio

Library to write concurrent IO-bound* code using the async/await syntax.

* Example for IO-bound: http requests ; example for CPU-bound: compression.
Asyncio - cont.

Why asyncio? what’s wrong with thread / process per request?

Currently, we consume more HTTP based services than ever.

=> We easily reach 10k connections concurrently on a single server (AKA c10k problem).

=> cooperative tasks that can better utilize a CPU can save a lot of $$$$.
Sanic

Python 3.6+ web server & web framework that's written to go fast using the async/await syntax.
Introducing pyaday

- CRUD for python packages metadata used by content curators.
- Get your daily random python package metadata for fun and profit.
Introducing pyaday - cont.

```python
> http http://127.0.0.1:5000/rand
HTTP/1.0 200 OK
Content-Length: 97
Content-Type: application/json
Date: Thu, 09 Jul 2020 11:52:28 GMT
Server: Werkzeug/1.0.1 Python/3.8.2

{
    "name": "poetry",
    "short_desc": "Python dependency management and packaging made easy."
}
```
from flask import Flask

from pyaday.api.packages import packages_bp
from pyaday.api.rand import rand_bp

app = Flask(__name__)
app.register_blueprint(packages_bp, url_prefix="/packages")
app.register_blueprint(rand_bp, url_prefix="/rand")

if __name__ == "__main__":
    app.run()
Introducing pyaday - cont.

```python
import json

from flask import Blueprint, Response, request
from werkzeug.datastructures import Headers

packages_bp = Blueprint(name="packages_bp", import_name=__name__)

python_packages = [
    {
        "name": "flask",
        "short_desc": "A simple framework for building complex web applications.",
    },
    {
        "name": "sanic",
        "short_desc": "A web server and web framework that's written to go fast. Build fast. Run fast.",
    },
    {
        "name": "poetry",
        "short_desc": "Python dependency management and packaging made easy.",
    },
]```
@packages_bp.route("", methods=['POST'])
def create_package():
    req_data = request.json
    python_packages.append(
        {
            "name": req_data.get("name"),
            "short_desc": req_data.get("short_desc")
        }
    )

    return Response(
        response=None,
        status=201,
        headers={"Location": f'/packages/{req_data.get('name')}"},
    )
Introducing pyaday - cont.

```python
@packages_bp.route("/\<package_name\>", methods=["GET"])
def read_package(package_name):
    for python_package in python_packages:
        if python_package["name"] == package_name:
            return jsonify(python_package)
    return Response(
        response=json.dumps({"title": "Could not find a package"}),
        status=404,
        content_type="application/problem+json",
    )
```
Introducing pyaday - cont.

```python
def update_package(package_name):
    req_data = request.json
    for python_package in python_packages:
        if python_package['name'] == package_name:
            python_package['short_desc'] = req_data.get('short_desc')
            return Response(response=None, status=204)
    return Response(
        response={'title': "Could not find a package"},
        status=404,
        content_type="application/problem+json",
    )
```
Introducing pyaday - cont.

```python
@packages_bp.route("/<package_name>", methods=["DELETE"])
def delete_package(package_name):
    for idx, python_package in enumerate(python_packages):
        if python_package["name"] == package_name:
            del python_packages[idx]
            return Response(response=None, status=204)
    return Response(
        response=json.dumps({"title": "Could not find a package"}),
        status=404,
        content_type="application/problem+json",
    )
```
Introducing pyaday - cont.

```python
import random

from flask import Blueprint, jsonify

from pyaday.api.packages import python_packages

rand_bp = Blueprint(name="rand_bp",
                   import_name=__name__)

@rand_bp.route("", methods=['GET'])
def read_random_package():
    return jsonify(random.choice(python_packages))
```
Why convert?

Better bang for the buck for a large scale expensive cloud deployment or a limited in resources on premises deployment.

=> Meaning - it will save you $$$

In addition, we’ll try to show the migration is not difficult and the flask knowledge is not wasted.
Let the conversion begin!

Prerequisite:
A project that can benefit from conversion written in python3.6-3.8 (I used 3.8.3).

$ poetry init #not mandatory, my preference
$ poetry add sanic

* Flask v1.1.2 & Sanic v20.3.0 were used, so syntax may vary on different versions.
App constructor

```python
from flask import Flask

app = Flask(__name__)
```

```python
from sanic import Sanic

app = Sanic(__name__)
```
Route

- On Flask request object is globally imported; on Sanic it is the first arg.
- On Sanic, the route is a coroutine (a function that uses the async keyword).
JSON response

Happy path:

```python
return {"hello": "world"}
```

```python
from sanic.response import json
return json({"hello": "world"})
```

```python
from flask import jsonify
return jsonify({"hello": "world"})
```
JSON response - cont.

Error handling (according to RFC7807):

```python
return Response(
    response=json.dumps({'title': 'Could not find a package'}),
    status=404,
    content_type='application/problem+json',
)
```

* There are other Flask options:

```python
response = jsonify(title="...")
response.status =...
return response
```
Auto reload for development

```python
if __name__ == "__main__":
    app.run(debug=True)
```

Same for Flask & Sanic*

* There are other options as well:
  - Flask, from terminal:
    FLASK_ENV=development FLASK_APP=main_flask.py flask run
  - Sanic: app.run(auto_reload=true)
* Used for sub-routing => contains all the exposed methods of a certain route.
* Sanic does not require import_name.
Blueprint - cont.

```python
app.register_blueprint(blueprint=rand_bp, url_prefix="/rand")
```

```python
app.blueprint(blueprint=rand_bp, url_prefix="/rand")
```

* `register_blueprint` can work as well in Sanic, but it is marked as deprecated.
Post conversion diff

```python
from flask import Blueprint, Response, request

packages_bp = Blueprint(name="packages_bp", import_name=__name__)
```

```python
from sanic import Blueprint, response

packages_bp = Blueprint(name="packages_bp")
```
Post conversion diff - cont.
Post conversion diff - cont.

```python
@packages_bp.route("/<package_name>", methods=["GET"],
def read_package(package_name):
    for python_package in python_packages:
        if python_package["name"] == package_name:
            return jsonify(python_package)

return Response(response=json.dumps({'title': "Could not find a package"}),
                 status=404,
                 content_type="application/problem+json",
)
```

```python
async def read_package(_, package_name):
    for python_package in python_packages:
        if python_package["name"] == package_name:
            return response.json(body=python_package)

return response.json(body={'title': "Could not find a package"},
                      status=404,
                      content_type="application/problem+json",
```
Post conversion diff - cont.
Post conversion diff - cont.

```python
@packages_bp.route("/<package_name>"), methods=["DELETE"]

def delete_package(package_name):
    for idx, python_package in enumerate(python_packages):
        if python_package["name"] == package_name:
            del python_packages[idx]
            return Response(response=None, status=204)
    return Response(
        response=json.dumps({"title": "Could not find a package"}),
        status=404,
        content_type="application/problem+json",
    )

async def delete_package(_, package_name):
    for idx, python_package in enumerate(python_packages):
        if python_package["name"] == package_name:
            del python_packages[idx]
            return response.empty(status=204)
    return response.json(
        body={"title": "Could not find a package"},
        status=404,
        content_type="application/problem+json",
    )
```
Testing

```python
with app.test_client() as test_client:
    response = test_client.get("rand")
    assert response.status_code == 200
    assert response.headers["content-type"] == "application/json"
```

```python
test_client = app.test_client
_, response = test_client.get("rand")
assert response.status == 200
assert response.headers["content-type"] == "application/json"
```

* Test client: Flask through a method and a context manager; Sanic - through an attribute.
* Calling routes: Flask - returns `response`; Sanic - returns `request` & `response`.
* Check response status: Flask - `status_code`; Sanic - `status`. 
Testing Diff
Sanic tests can also be async (pytest-sanic package is a requirement for this):

```python
import asyncio

class TestSanic(unittest.TestCase):
    def test_sanic_async(self):
        @pytest.fixture
        def test_client(loop, sanic_client):
            return loop.run_until_complete(sanic_client(app))

        async def test_rand_async(test_client):
            response = await test_client.get("/rand")
            assert response.status == 200
            assert response.headers["content-type"] == "application/json"
```

Testing Diff - cont.
Testing Diff - cont.

* There is only one return value - response, similar to Flask.

* Need to “await” every server call as opposed to Flask.
Deployment

```
> gunicorn -w 4 main_flask:app -b 127.0.0.1:5000
[2020-07-09 15:11:07 +0300] [98065] [INFO] Starting gunicorn 20.0.4
[2020-07-09 15:11:07 +0300] [98065] [INFO] Listening at: http://127.0.0.1:5000 (98065)
[2020-07-09 15:11:07 +0300] [98065] [INFO] Using worker: sync

> gunicorn -w 4 -k uvicorn.workers.UvicornWorker main_sanic:app -b 127.0.0.1:8000
[2020-07-09 15:11:53 +0300] [98087] [INFO] Starting gunicorn 20.0.4
[2020-07-09 15:11:53 +0300] [98087] [INFO] Listening at: http://127.0.0.1:8000 (98087)
[2020-07-09 15:11:53 +0300] [98087] [INFO] Using worker: uvicorn.workers.UvicornWorker
```
## Deployment - cont.

### Before Deployment

<table>
<thead>
<tr>
<th>Server Software:</th>
<th>gunicorn/20.0.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Hostname:</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Server Port:</td>
<td>5000</td>
</tr>
<tr>
<td>Document Path:</td>
<td>/rand</td>
</tr>
<tr>
<td>Document Length:</td>
<td>Variable</td>
</tr>
<tr>
<td>Concurrency Level:</td>
<td>100</td>
</tr>
<tr>
<td>Time taken for tests:</td>
<td>9.816 seconds</td>
</tr>
<tr>
<td>Complete requests:</td>
<td>10000</td>
</tr>
<tr>
<td>Failed requests:</td>
<td>0</td>
</tr>
<tr>
<td>Total transferred:</td>
<td>2491241 bytes</td>
</tr>
<tr>
<td>HTML transferred:</td>
<td>967854 bytes</td>
</tr>
<tr>
<td>Requests per second:</td>
<td>1018.78 [/sec] (mean)</td>
</tr>
<tr>
<td>Time per request:</td>
<td>98.156 [ms] (mean)</td>
</tr>
<tr>
<td>Time per request:</td>
<td>0.982 [ms] (mean, across all)</td>
</tr>
<tr>
<td>Transfer rate:</td>
<td>247.85 [Kbytes/sec] received</td>
</tr>
</tbody>
</table>

### After Deployment

<table>
<thead>
<tr>
<th>Server Software:</th>
<th>uvicorn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Hostname:</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Server Port:</td>
<td>8000</td>
</tr>
<tr>
<td>Document Path:</td>
<td>/rand</td>
</tr>
<tr>
<td>Document Length:</td>
<td>Variable</td>
</tr>
<tr>
<td>Concurrency Level:</td>
<td>100</td>
</tr>
<tr>
<td>Time taken for tests:</td>
<td>1.552 seconds</td>
</tr>
<tr>
<td>Complete requests:</td>
<td>10000</td>
</tr>
<tr>
<td>Failed requests:</td>
<td>0</td>
</tr>
<tr>
<td>Total transferred:</td>
<td>2210797 bytes</td>
</tr>
<tr>
<td>HTML transferred:</td>
<td>957432 bytes</td>
</tr>
<tr>
<td>Requests per second:</td>
<td>6445.09 [/sec] (mean)</td>
</tr>
<tr>
<td>Time per request:</td>
<td>15.516 [ms] (mean)</td>
</tr>
<tr>
<td>Time per request:</td>
<td>0.155 [ms] (mean, across all)</td>
</tr>
<tr>
<td>Transfer rate:</td>
<td>1391.48 [Kbytes/sec] received</td>
</tr>
</tbody>
</table>

**5-6x for GET /rand route**
Not always a fairytale

- A cognitive bourdain: for a performant (and an effective) async code the event loop must never be blocked:
  - IO should be await(ed)
  - CPU should run elsewhere (loop.run_in_executor(...))
- Sanic's ecosystem is not as rich as Flask's ecosystem. It is noticeable on Github, on the number of available tutorials and on 3rd party integrations (like okta, auth0 or swagger-codegen).
Not always a fairytale - cont.

- Need to use 3rd party libraries that do not block IO:
  - psycopg2 -> asyncpg / aiopg*
  - requests -> httpx / aiohttp
  - redis -> aioredis / asyncio-redis

... 

* That's why a DB wasn't used for the converted application - to make the comparison simple.
The async web framework landscape

- **Sanic** was chosen for this talk because:
  - It is popular on Github
  - The API it exposes is very similar to the API exposed by Flask. When the API is not the same, it seems like a reasonable evolution that's made possible because there isn't a lot of backward compatibility needed.
  - It is backed by a community run organization.
  - 90s flashback :)

- **Quart** is also a Flask like async web framework.

- **Fastapi** is a hybrid web framework (sync and an async) with dependency injection as a guiding principle.
Summary

- When a Flask app that mostly performs IO becomes resource hungry, it is worthwhile to convert it to Sanic in reasonable effort.
- After converting, the code must be IO & CPU aware in order to not block the event loop.
@DavidBordeynik