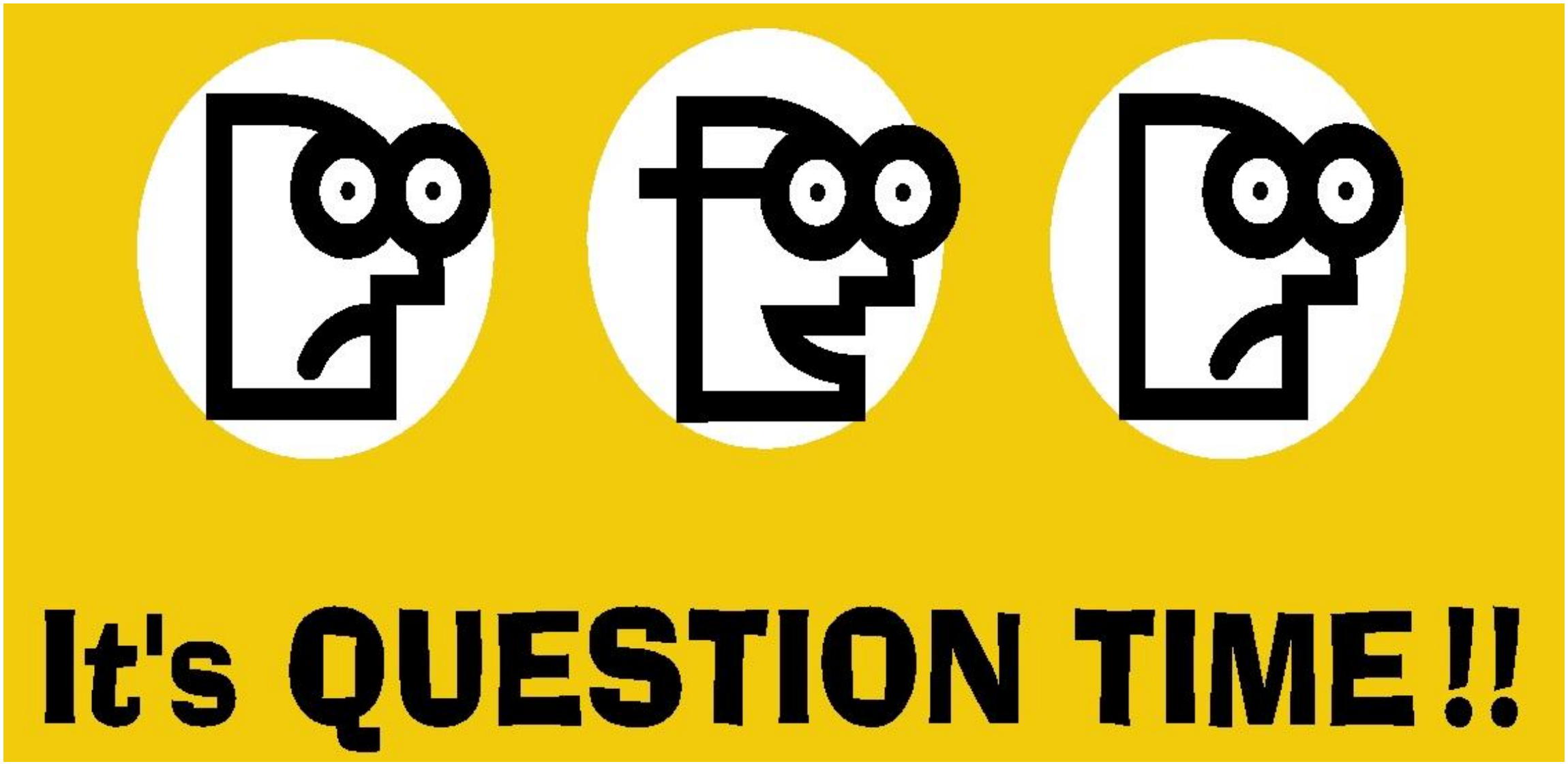
Advanced Software Engineering (LAB)

Stefano Forti

name.surname@di.unipi.it

Department of Computer Science, University of Pisa


It's QUESTION TIME !!

Checklist

A. Ubuntu/MacOS installed locally.
B. Ubuntu running in a VM (e.g., using VirtualBox)

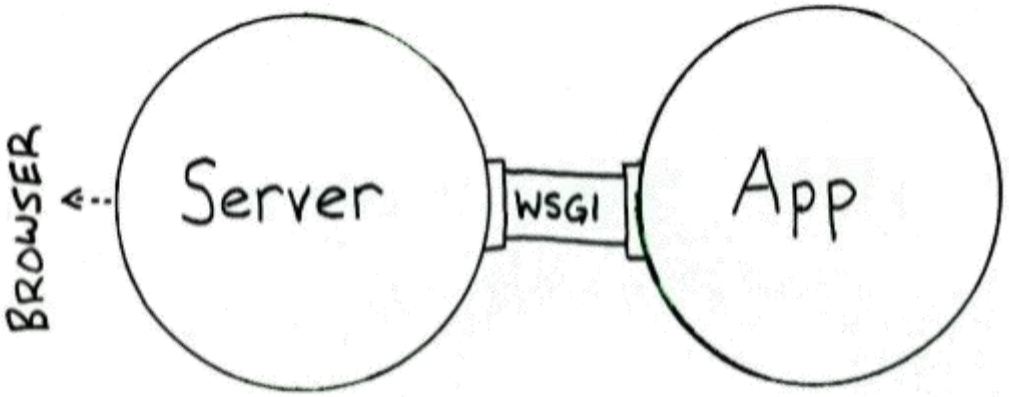
C. curl properly installed.

We'll need $(A \vee B) \wedge C$

WSGI

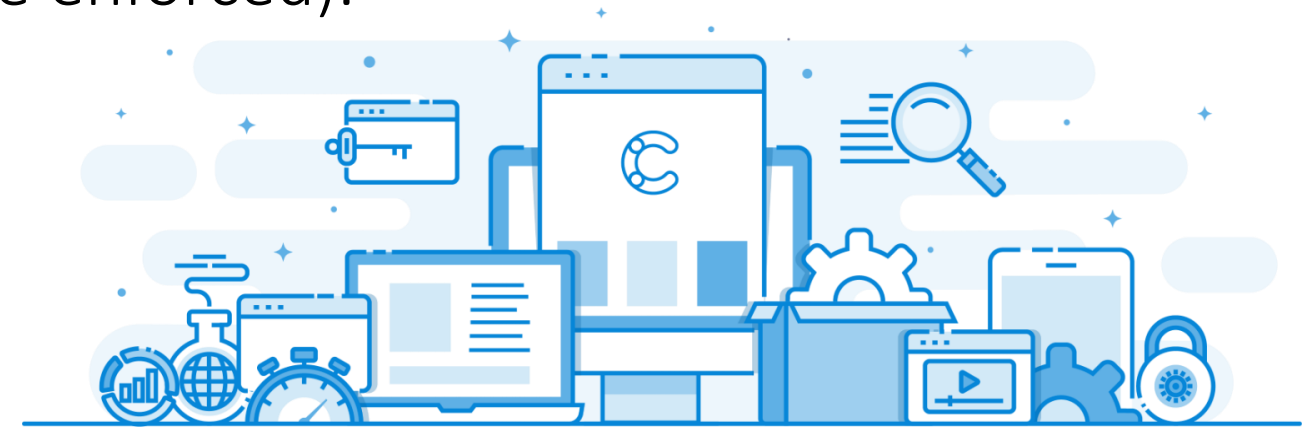
- With Python it is easy to get web applications up and running.
- The Python Web Community created the Web Server Gateway Interface (WSGI) to simplify serving HTTP requests.
- WSGI can be executed on standard Web servers (e.g., Apache, nginx).

- The sole problem of WSGI is its synchronous nature: the application stays idle until it gets a response from the invoked service.



Microframeworks

- Flask was started in 2010, leveraging the Werkzeug WSGI toolkit.
- Together with Bottle and a handful of other projects, they constitute the Python **microframeworks** ecosystem.
- Microframeworks are a set of tools designed to build Web apps faster.
- Micro- here means that the framework attempts to take as few decisions as possible for the programmer (no particular paradigm or design choice enforced).



- Which Python?
- How Flask handles requests
- Flask built-in features
- A microservice skeleton

Which Python?

Python 3 Support

Flask, its dependencies, and most Flask extensions support Python 3. You should start using Python 3 for your next project, but there are a few things to be aware of.

You need to use Python 3.3 or higher. 3.2 and older are *not* supported.

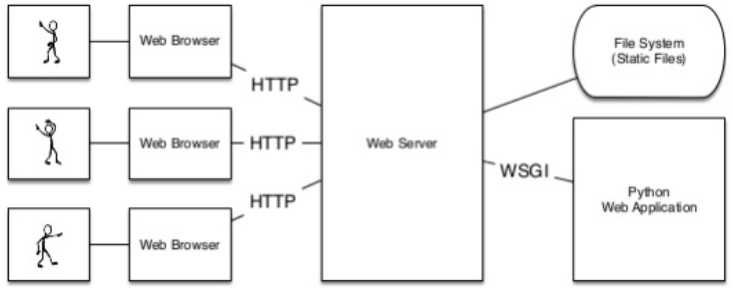
You should use the latest versions of all Flask-related packages. Flask 0.10 and Werkzeug 0.9 were the first versions to introduce Python 3 support.

Python 3 changed how unicode and bytes are handled, which complicates how low level code handles HTTP data. This mainly affects WSGI middleware interacting with the WSGI `environ` data. Werkzeug wraps that information in high-level helpers, so encoding issues should not affect you.

The majority of the upgrade work is in the lower-level libraries like Flask and Werkzeug, not the high-level application code. For example, all of the examples in the Flask repository work on both Python 2 and 3 and did not require a single line of code changed.

Handling requests

- The entry point is the `Flask` class in the `flask.app` module.
- Flask apps run one instance of the `Flask` class, taking care of all incoming WSGI requests by
  1. Dispatching them to the right code, and
  2. Returning a response to the caller.



WSGI is a specification that defines the interface between web servers and Python applications. The incoming request is described in a single mapping, and frameworks such as Flask take care of routing the call to the right callable.

Our first microservice

```
pip install Flask
```

The class offers a **route** method, which can decorate functions. Decorated functions become views in the Werkzeug routing system.

The **__name__** variable is the name of the application package. Flask instantiates a new logger with that name and a suitable directory.

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api')
def my_microservice():
        return jsonify({'Hello': 'World'})

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

```
python <filename>.py
```

```
* Serving Flask app "lecture2" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

https://curl.haxx.se/

- A useful command we will often use is:

```
curl -v http://127.0.0.1:5000/api
```

```
StatusCode        : 200
StatusDescription : OK
Content           : {"Hello":"World"}

RawContent        : HTTP/1.0 200 OK
                    Content-Length: 18
                    Content-Type: application/json
                    Date: Mon, 01 Oct 2018 08:53:28 GMT
                    Server: Werkzeug/0.14.1 Python/3.7.0

                    {"Hello":"World"}

Forms             : {}
Headers           : {[Content-Length, 18], [Content-Type, application/json], [Date, Mon, 01 Oct 2018 08:53:28 GMT],
                    [Server, Werkzeug/0.14.1 Python/3.7.0]}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 18
```

Calling **`/api`** returns a valid JSON with the right headers, thanks to the **`jsonify()`** function, converting the Python **`dict`** into a valid JSON response with the proper **`Content-Type`** header.

The `request` variable

- Flask provides an implicit **`request`** variable, pointing to the current **`Request`** object.
- The **`request`** variable is global, but unique, to each incoming request and it is thread-safe. Let's play with our micro-service.

```
from flask import Flask, jsonify, request
app = Flask(__name__)

@app.route('/api')
def my_microservice():
    print(request)
    response = jsonify({'Hello': 'World'})
    print(response)
    print(response.data)
return response
```

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

```
python <filename>.py
```

curl it!

```
curl -v http://127.0.0.1:5000/api
```

```
127.0.0.1 - - [01/Oct/2018 11:04:00] "GET /api HTTP/1.1" 200 -
<Request 'http://127.0.0.1:5000/api' [GET]>
{'wsgi.version': (1, 0),
'wsgi.url_scheme': 'http',
'wsgi.input': <_io.BufferedReader name=940>,
'wsgi.errors': <_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>,
'wsgi.multithread': True,
'wsgi.multiprocess': False,
'wsgi.run_once': False,
'werkzeug.server.shutdown': <function WSGIRequestHandler.make_environ.<locals>.shutdown_server at 0x03981468>,
'SERVER_SOFTWARE': 'Werkzeug/0.14.1',
'REQUEST_METHOD': 'GET',
'SCRIPT_NAME': '',
'PATH_INFO': '/api',
'QUERY_STRING': '',
'REMOTE_ADDR': '127.0.0.1',
'REMOTE_PORT': 59228,
'SERVER_NAME': '127.0.0.1',
'SERVER_PORT': '5000',
'SERVER_PROTOCOL': 'HTTP/1.1',
'HTTP_USER_AGENT': 'Mozilla/5.0 (Windows NT; Windows NT 10.0; it-IT) WindowsPowerShell/5.1.17134.228',
'HTTP_HOST': '127.0.0.1:5000',
'HTTP_CONNECTION': 'Keep-Alive',
'werkzeug.request': <Request 'http://127.0.0.1:5000/api' [GET]>}
<Response 18 bytes [200 OK]>
b'{"Hello":"World"}\n'
127.0.0.1 - - [01/Oct/2018 11:04:13] "GET /api HTTP/1.1" 200 -
```

Under the hood…

`Routing`

Flask creates the `Map` class

A `Map` class is created to determine if a function decorated by `@app.route` matches the incoming request.
By default, the mapper only accepts `GET, OPTIONS` and `HEAD` calls declared on a route (`405 Method Not Allowed otherwise`).

`Request`

Flask passes a `Request` object to the view

A `Request` object is created, guaranteeing an isolated environment for the thread handling it.

`Response`

A `Response` object is sent back with the response content


```
curl -v -XDELETE http://127.0.0.1:5000/api
```

Supporting other methods

You can try different request types by using the –X flag of the curl command, followed by the request type. E.g.

```
curl -v -X DELETE http://127.0.0.1:5000/api
```

```
stefano@DESKTOP-MCOIMB6:~$ curl -v -XDELETE 127.0.0.1:5000/api
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 5000 (#0)
> DELETE /api HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 127.0.0.1:5000
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 405 METHOD NOT ALLOWED
< Content-Type: text/html
< Allow: GET, HEAD, OPTIONS
< Content-Length: 178
< Server: Werkzeug/0.14.1 Python/3.7.0
< Date: Mon, 01 Oct 2018 09:55:56 GMT
<
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>405 Method Not Allowed</title>
<h1>Method Not Allowed</h1>
<p>The method is not allowed for the requested URL.</p>
* Closing connection 0
stefano@DESKTOP-MCOIMB6:~$
```

```python
@app.route('/api', methods=['POST', 'DELETE', 'GET'])
def my_microservice():
    response = jsonify({'Hello': 'World'})
    return response
```

If you want to support specific methods, you can pass them to the route decorator with the **`methods`** argument.

Variables

- You can use variables using the `<VARIABLE_NAME>` syntax.
- For instance, if you want to create a function that handles all requests to `/person/id`, with `id` being the unique ID of a person, you could use `/person/<person_id>`.

```
@app.route('/api/person/<person_id>')
def person(person_id):
    response = jsonify({'Hello': person_id})
    return response
```

A **converter** can convert the variable to a particular type. For instance, if you want an integer, use `<int:VARIABLE_NAME>`. Input is then checked against the type. Built-in converters are `string` (the default, a Unicode string), `int`, `float`, `path`, `any`, and `uuid`.

Custom converters

```python
from flask import Flask, jsonify, request
from werkzeug.routing import BaseConverter, ValidationError

_USERS = {'1': 'Fred', '2': 'Barney', '3': 'Wilma'}
_IDS = {val: id for id, val in _USERS.items()}

class RegisteredUser(BaseConverter):
    def to_python(self, value):
        if value in _USERS:
            return _USERS[value]
        raise ValidationError()

    def to_url(self, value):
        return _IDS[value]

app = Flask(__name__)
app.url_map.converters['registered'] = RegisteredUser

@app.route('/api/person/<registered:name>')
def person(name):
    response = jsonify({'Hello': name})
    return response
```

- To create custom converters, we extend the `BaseConverter` class, implementing:
  - `to_python()`, which converts the value to a Python object for the view, and
  - `to_url()`, which converts the Python object to a value.

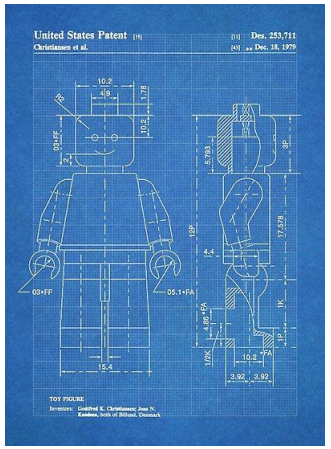
```
curl -v http://127.0.0.1:5000/api/person/1
```

```
curl -v http://127.0.0.1:5000/api/person/5
```

(Other) Flask built-in features

- **The session object**: Cookie-based data
- **Globals**: Storing data in the request context
- **Signals**: Sending and intercepting events
- **Extensions and middlewares**: Adding features
- **Templates**: Building text-based content
- **Configuring**: Grouping your running options in a config file
- **Blueprints**: Organizing your code in namespaces
- **Error handling and debugging**: Dealing with errors in your app

Blueprints



- Microservices typically consist of more than one endpoint, i.e. a handful of Flask-decorated functions.
- Code should be organised according to the rule $1\ module \equiv 1\ view$
  E.g., in a microservice that manages employees and teams of a company you might have 3 modules: `app.py`, `employees.py`, `teams.py`.
- A `Blueprint` is a way to organise a group of related views and other code. Rather than registering views and other code directly with an application, they are registered with a blueprint.

Example: Employees Blueprint

```python
#teams.py
from flask import Blueprint, jsonify

teams = Blueprint('teams', __name__)

_DEVS = ['Tarek', 'Bob']
_OPS = ['Bill']
_TEAMS = {1: _DEVS, 2: _OPS}

@teams.route('/teams')
def get_all():
        return jsonify(_TEAMS)

@teams.route('/teams/<int:team_id>')
def get_team(team_id):
        return jsonify(_TEAMS[team_id])
```

```python
#app.py
from flask import Flask, jsonify, request
from teams import teams

app = Flask(__name__)
app.register_blueprint(teams)

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

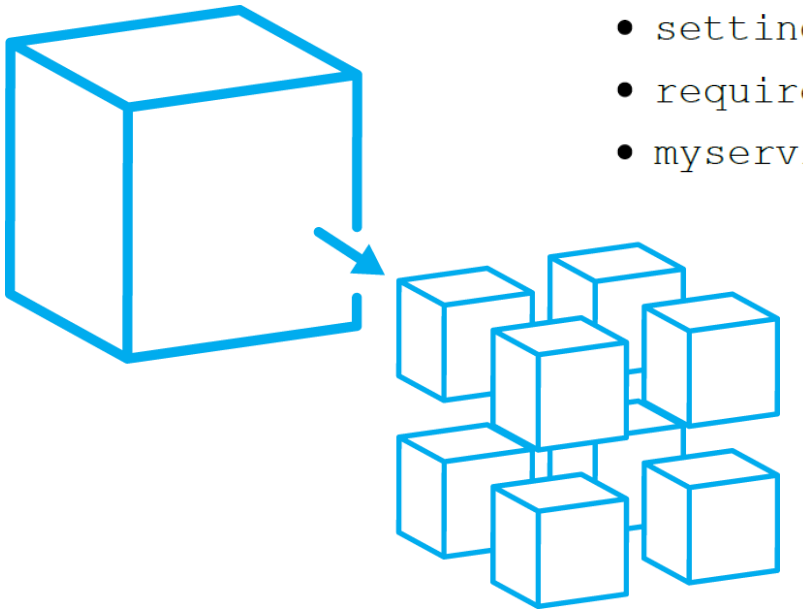
Try to perform a bad request like
`http://127.0.0.1:5000/teams/9`
via your favourite browser!

A microservice skeleton

- Start by running: `pip install Flakon`

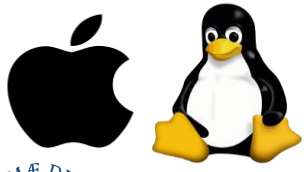
- Our examples used a single module and the **`app.run()`** method call to run the service.
- A microservice skeleton can be found on the Moodle.

Try it!



The `microservice` project skeleton contains the following structure:

- `setup.py`: Distutils' setup file, which is used to install and release the project
- `Makefile`: A Makefile that contains a few useful targets to make, build, and run the project
- `settings.ini`: The application default settings in the INI file
- `requirements.txt`: The project dependencies following the pip format
- `myservices/`: The actual package
  - `__init__.py`
  - `app.py`: The app module, which contains the app itself
  - `views/`: A directory containing the views organized in blueprints
    - `__init__.py`
    - `home.py`: The home blueprint, which serves the root endpoint
  - `tests`: The directory containing all the tests
    - `__init__.py`
    - `test_home.py`: Tests for the home blueprint views



```
$ pip install -r requirements.txt
$ python setup.py develop
$ export FLASK_APP=myservice
$ flask run
```

```
> pip install -r requirements.txt
> python setup.py develop
> $env:FLASK_APP = "myservice"
> flask run
```

In-class Work

```
$ pip install -r requirements.txt
$ python setup.py develop
$ export FLASK_APP=myservice
$ flask run
```

Use the microservice skeleton to implement a calculator, by using the `calculator.py` module from Lab 1.

1. Create a `calc.py` file inside `views` (→).
2. Import it in the `views/__init__.py` file and add `calc` in the `blueprints` list.
3. Implement the other 3 methods.

```python
from flakon import JsonBlueprint
from flask import Flask, request, jsonify

calc = JsonBlueprint('calc', __name__)

@calc.route('/calc/sum', methods=['GET'])
def sum():
    #http://127.0.0.1:5000/calc/sum?m=3&n=5
    m = int(request.args.get('m'))
    n = int(request.args.get('n'))

    result = m

    if n < 0:
        for i in range(abs(n)):
            result -= 1
    else:
        for i in range(n):
            result += 1

    return jsonify({'result':str(result)})
```

Handling JSON

- Flask has great support for JSON, and is a popular choice for building JSON APIs.
- Making requests with JSON data and examining JSON data in responses is very convenient →
- You can easily test JSON APIs with

https://www.getpostman.com/

```
@app.route('/api/auth')
def auth():
    json_data = request.get_json()
    email = json_data['email']
    password = json_data['password']
    return jsonify({'email': email})
```

Homework 1