Programmation back-end

Fabien Coelho, Claire Medrala

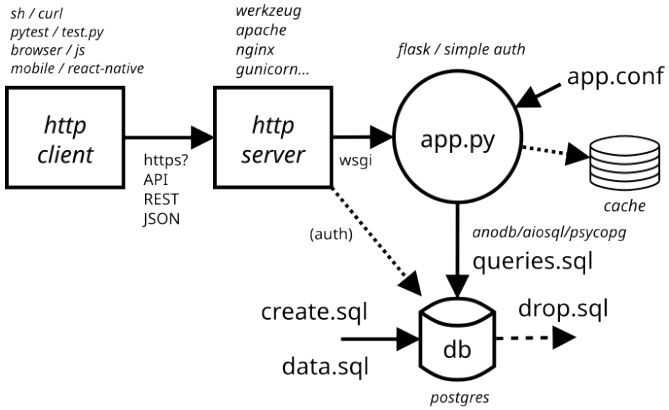
Mines Paris – PSL, MESR

Janvier 2025

Menu

Architecture back-end

Architecture back-end *très simplifiée*



Architecture back-end — Technologies

HTTP	*HyperText Transport Protocol*	RFC 2616
WSGI	*Python Web Server Gateway Interface*	PEP 333
ASGI	*Python Asynchronous Server Gateway Interface*	
Flask	requête HTTP : conf, params, auth, réponses, json...	
app	*votre application !*	
AnoDB	surcouche de AioSQL qui gère les connexions	
AioSQL	gestion de requêtes à une base de donnée relationnelle	
PsycoPg	interface base de données	DB API, PEP 249
Postgres	base de donnée relationnelle (ou tests avec SQLite)	
Redis	cache de données (clef-valeur), partagé	

Architecture back-end — Sécurité – AAA

Préoccupation globale !

Authentification — *Qui ?*

- utilisateur, mot de passe...
- HTTP Basic, Digest, paramètres, *token*

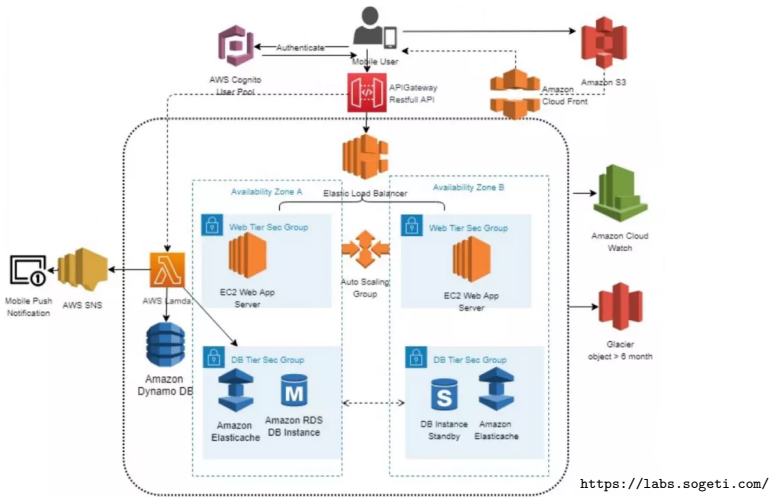
Authorization — *Quoi ?*

- utilisateur, rôle (groupe)...
- par route ou selon les données

Audit — *Quand ?*

- qui fait quoi : traces, données, historique

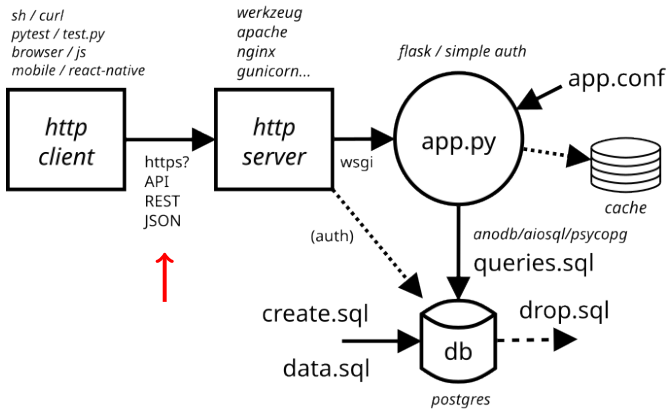
Architecture back-end... — Exemple AWS



https://labs.sogeti.com/

REST

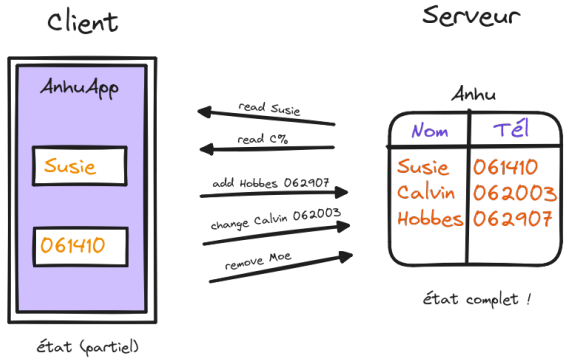
API REST



REST – REpresentational State Transfer

Architecture logicielle — Interface

- transfert d'états — identification et manipulation de ressources



REST – REpresentational State Transfer

Architecture

- proposé par Roy Fielding (HTTP, Fondation Apache)
  PhD à Irvine (Ca) en 2000 sur Architecture du Web...
- s'appuie sur le protocole HTTP

Interfaces avec 5 contraintes principales — API

client-serveur	asymétrique, requête-réponse
sans état	requêtes auto-contenues
uniforme	forme URI, paramètres et résultats en JSON (ou XML)
cachable	informations gardées par le client
en couche	scalability avec proxy, équilibrage de charge...

HTTP – protocole client-serveur

`https://calvin:mdp@kiva.mobapp.minesparis.psl.eu/api/store?flt=c%`

Requête HTTP

```
GET /api/store HTTP/1.1
Host: kiva.mobapp.minesparis.psl.eu
Authorization: Basic Y2FsdmluOm1kcA==
Content-Length: 16
Content-Type: application/json

{"flt":"c%"}
```

méthode	GET POST...
chemin	/api/store
entêtes	eg authentification, ...
corps	données, eg JSON

Réponse HTTP

```
HTTP/1.1 200 OK
Server: Apache/2.4.52 (Ubuntu)
Content-Length: 34
Content-Type: application/json

[{"key":"calvin","val":"hobbes"}]
```

code	2xx 4xx 5xx...
état	OK, Error...
entêtes	eg serveur, date, ...
corps	données, eg JSON

REST / HTTP – concrètement

Appel de fonction, avec effet de bord, à travers HTTP

chemin	identification d'une ressource *(point d'entrée, path, endpoint)*
méthode	GET POST PATCH PUT DELETE idempotence GET PUT DELETE, cachabilité GET POST
paramètres	HTTP ou JSON, retour JSON
authn	authentification HTTP basic/param, token...

Requête

```
GET /students/5432 HTTP/1.1
Host: api.minesparis.psl.eu
Authorization: Basic Zm9vOmJsYT8h
```

Réponse

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 31

{"nom":"Calvin","classe":"CE1"}
```


REST / HTTP – conventions

Méthodes — *SQL*

GET	consultation	SELECT
POST	ajout (retour *id* ?)	INSERT
PATCH/PUT	modification (partielle/totale)	UPDATE
DELETE	effacement	DELETE ?

Chemin – **identification** d'une ressource — *tuple(s)*

/students	liste des étudiants
/students/42	l'étudiant numéro 42
/students/42/courses	liste des cours de l'étudiant 42
/courses/53	le cours 53
/courses/53/students	liste des étudiants du cours 53

REST / HTTP – paramètres

Localisation

implicite	dans le chemin lui-même	/courses/*42*
explicite	en HTTP ou JSON ou XML	nom=...
valeur	texte, éventuellement convertie	...=2020-07-29

Usage

identification de la ressource concernée	chemin implicite
valeurs (données, critères de recherche)...	explicite

GET /students	nom=*C%*	*dont le nom commence par C*
POST /students	nom=*Hobbes* ne=...	*création de Hobbes*
PATCH /students/*42*	ne=*2001-02-03*	*modification date de naissance*
PUT /students/*42*	ne=*2001-02-02* nom=...	*modification complète*
DELETE /students/*42*		*efface cet étudiant*
DELETE /students		*efface tous les étudiants*

REST / HTTP – exemples de requêtes et réponses

Paramètres JSON

```
GET /api/store HTTP/1.1
Host: kiva.mobapp.minesparis.psl.eu
Authorization: Basic Y2FsdmluOm1kcA==
Content-Length: 16
Content-Type: application/json

{"flt":"c%"}
```

Paramètres HTTP (bof)

```
GET /api/store HTTP/1.1
Host: kiva.mobapp.minesparis.psl.eu
Authorization: Basic QXJlIHlvdSBqb2tpbmc/
Content-Length: 11
Content-Type: application/x-www-form-urlencoded

flt=c%25
```

Réponse JSON liste de tuples

```
HTTP/1.1 200 OK
Server: Apache/2.4.52 (Ubuntu)
Content-Length: 22
Content-Type: application/json

[["calvin","hobbes"]]
```

Réponse JSON liste de dicts

```
HTTP/1.1 200 OK
Server: Apache/2.4.52 (Ubuntu)
Content-Length: 34
Content-Type: application/json

[{"key":"calvin","val":"hobbes"}]
```

REST / HTTP – retours

état et valeurs

HTTP Status Codes

200	OK	GET
201	Created	POST
204	No Content	DELETE PATCH PUT
400	Bad Request	eg pb paramètres
401	Unauthorized	authentification
403	Forbidden	autorisation
404	Not Found	ressource inexistante
405	Method Not Allowed	pas implémenté
409	Conflict	POST ?

Valeurs

- JSON de préférence ! (vs texte, XML...) — compatible JS
- tableaux vs objets vs ...

JSON – format

JavaScript Object Notation

JSON – STD 90 / RFC 8259 (2017)

- serialisation d'un objet sous forme de texte unicode
- JSON : objet, tableau, valeur num, chaîne, `true false null`
  Python : `dict list int/float str True False None`
- échappement avec *backslash*
- **mais** pas de commentaires, dictionnaires de *string*...

```
{
  "id": 42,
  "nom": "PostgreSQL",
  "version": 17.2,
  "liste": [ "hello world!\n", "Susie\tCalvin\tHobbes\n" ],
  "updated": false,
  "none": null,
  "tags": { "one": 1, "two": 2.0 }
}
```


REST / HTTP – exemples

`geo.api.gouv.fr`

Région *01*

```
GET /regions/01 HTTP/1.1
Host: geo.api.gouv.fr
```

Réponse JSON

```
HTTP/1.1 200 OK
Server: nginx/1.10.3 (Ubuntu)
Content-Type: application/json; charset=utf-8
Content-Length: 32

{"nom":"Guadeloupe","code":"01"}
```

Département *46*

```
GET /departements/46 HTTP/1.1
Host: geo.api.gouv.fr
```

Réponse JSON

```
HTTP/1.1 200 OK
Server: nginx/1.10.3 (Ubuntu)
Content-Type: application/json; charset=utf-8
Content-Length: 43

{"nom":"Lot","code":"46","codeRegion":"76"}
```


REST / HTTP – exemples

`api-adresse.data.gouv.fr`

```
curl -s "https://api-adresse.data.gouv.fr/reverse/?lat=48.845&lon=2.339" | \
  jq .features[0].properties.label
```

```
GET /reverse/?lat=48.845&lon=2.339 HTTP/1.1
Host: api-adresse.data.gouv.fr
```

```
HTTP/1.1 200 OK
Server: nginx/1.23.3
Date: Wed, 10 Jan 2024 08:15:30 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 2585
Connection: keep-alive
Vary: Origin
ETag: W/"a19-s7ifV//ge9CBj7Euq+QViFCDCw4"
X-Cache-Status: MISS
Access-Control-Allow-Headers: X-Requested-With,Content-Type

{
  "label":"60b Boulevard Saint-Michel 75006 Paris", ...
}
```

Philosophie

Cycle de vie des données – CRUD(S)

C	Create	INSERT
R	Read	SELECT
U	Update	UPDATE
D	Delete	DELETE
S	Search	SELECT

Types d'interfaces

CRUDS	élémentaires	*souple, logique côté client, latence...*
haut niveau	métier	*rigide, logique côté serveur, latence réduite*

Niveaux de maturité REST — *Leonard Richardson*

0 swamp **1** ressources **2** + verbes **3** + retour de liens (HATEOAS)

Conseils

Préliminaires

- modèle de données bien défini — *concepts*
- user stories — *fonctionalités*

Règles sur la partie chemin — homogénéité

caractères	ASCII, minuscules, - **pas** _
style	court, pluriel ok, **pas** d'extensions, de types, de méthodes...
valeurs	uniquement pour identifier une ressource (clé primaire)
hiérarchie	avec /, **pas** à la fin

Oui `/eleves` `/eleves/5432` `/eleves/5432/cours`

Non `/Get_liste_des_élèves.json/`
`/Élève_par_numéro?n=5432`
`/Cours_d_un_élève_par_numéro?n=5432`

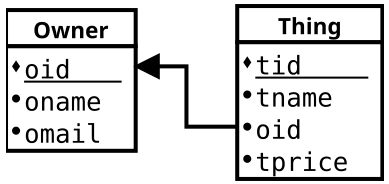
Exercice — Définition d'une API REST

Interface REST

- chemin — préfix et paramètres intégrés
- méthodes pertinentes — selon les besoins !
- gestion des permissions — qui peut le faire ?
- paramètres requis/facultatifs ? — types, forme...
- retours attendus (ok et erreurs) — status, sortie (exemple)

path	method	who	in	stat	out	comment
`/users`	`GET`	*admin*	`flt?`	200	*array*	*list users*
	`POST`	*admin*	`login pass admin`	201	*int*	*create user*
				409	–	*user exists*
`/users/<uid>`	`GET`	*admin*		200	obj	*get user*
				404	–	*no such user*
	`PATCH`	*admin*	`pass? admin?`	204	–	*update user*
				404	–	*no such user*

Exercice — *Interface REST de type CRUD*



Owner — propriétaire

oid	clé primaire
oname	nom de la personne, unique
omail	adresse de messagerie

Thing — chose

tid	clé primaire
tname	nom de la chose
oid	clé étrangère vers le propriétaire
tprice	valeur de l'objet

Web vs REST

Les 7 différences...

Web	vs	REST
méthodes	GET/POST	GET/POST/PUT/PATCH/DELETE
user-agent	navigateur	application
format	document HTML	données JSON
auth	page login/logout	par requête, persistent
session	serveur	application
http	grosses requêtes	petites requêtes nombreuses
back-end	plus complexe (templates, sessions)	plus simple

Exemples d'API REST

OpenAPI

`https://www.postman.com/explore` `https://swagger.io`

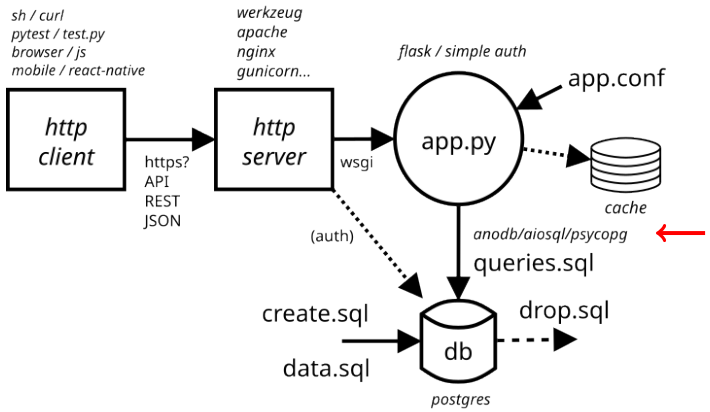
- nombreuses API, y compris publiques
- problèmes de sécurité : limitation du flux de requêtes
- authentifications par clés, abonnements...
- ne suivent pas souvent les règles !
- Postman : spec, doc, tests...

Conseils

modéliser	bien définir les concepts manipulés par l'API
documenter	pour dev *front end* et *back end*
JSON	privilégier pour les paramètres et les retours (JS)
latence	HTTP coûte cher, aggréger les transferts !
asynchrone	si traîtements trop longs, files d'attentes à traîter...

Python DB API

DB API



Connexion Python – DB

Modèle similaire dans tous les langages

- identification de la base de données
- établissement d'une connexion authentifiée
- exécution de divers types de requêtes
- passage de paramètres vs *SQL injection*
- conversion types langages et SQL
- consultation des métadonnées
- gestion des erreurs...

Python DB API, Java JDBC, Perl DBI, C ODBC, R DBI, Rust SQLx...

Connexion Python – DB

Cycle de vie

1. établissement d'une connexion
2. création d'un ou plusieurs curseurs
3. exécution d'une requête
4. récupération des résultats...
5. fermetures...

```python
import psycopg as db
conn = db.connect("")
curs = conn.cursor()
curs.execute("SELECT 1, 'un' UNION SELECT 2, 'deux'")
for i in range(curs.rowcount):
    print(curs.fetchone())
curs.close()
conn.close()
```


Portabilité ? DB API 2.0 – PEP249

NON !

- parties facultatives de la spécifications...
- variantes de passages de paramètres %s vs ?...
- extensions diverses et variées
- chargement explicite du package, pas d'abstraction *(driver)*

```python
# NE FONCTIONNE PAS
import sqlite3 as db
conn = db.connect(":memory:")
curs = conn.cursor()
curs.execute("SELECT 1, 'un' UNION SELECT 2, 'deux'")
for i in range(curs.rowcount):
    print(curs.fetchone())
curs.close()
conn.close()
```


Types de connexions et d'implémentations

- connexion directe vs réseau
- implémentation du protocole vs utilisation d'une librairie
- variantes asynchrones (non standard) `async await`...

SQLite

`sqlite3` connexion directe via librairie

Postgres

`psycopg` connexion réseau via librairie `libpq`
`pg8000` connexion réseau via TCP/IP
`pygresql` connexion réseau via librairie `libpq`
`aiopg` version asynchrone au dessus de `psycopg2`

Connexion

`conn = db.connect(...)`

- identification de la base de donnée
- authentification...
- options diverses
- support des transactions `commit` `rollback`
- cher ! partager si possible, persistance...

```
import psycopg as db
conn = db.connect("host=pagode dbname=comics")
conn = db.connect(host="pagode", dbname="comics", user="calvin", password="5ecret")
conn = db.connect(...)
conn.commit()
conn.close()
```


Curseur

`curs = conn.cursor()`

- exécution d'une requête ou d'une commande — `execute`
- parcours de la relation résultat
  - automatique — `for ... in ...`
  - manuel : tuples / `None` — `fetchone` `fetchmany` `fetchall`
- fermeture — `close`
- fonctionne aussi avec un contexte — `with`

```
# version portable...
with conn.cursor() as curs:
    curs.execute("SELECT * FROM Auteur")
    for row in curs:
        print(row)
```


Curseur – Retour dictionnaire

- résultats par défaut : liste de tuples
- modification avec `row_factory` : dict, objets...

```
import psycopg as pg
with pg.connect(row_factory=pg.rows.dict_row) as conn:
    with conn.cursor() as curs:
        curs.execute("SELECT i AS i, i^3 AS i³ FROM generate_series(2, 8) AS i")
        for row in curs:
            print("row:", row)
    conn.commit()
# row: {'i': 2, 'i³': 8.0}
# row: {'i': 3, 'i³': 27.0}
# row: {'i': 4, 'i³': 64.0}
# row: {'i': 5, 'i³': 125.0}
# row: {'i': 6, 'i³': 216.0}
# row: {'i': 7, 'i³': 343.0}
# row: {'i': 8, 'i³': 512.0}
```


Curseur – Passage de paramètres

5 styles	*qmark numeric format named pyformat*
paramètres	tuple ou dictionnaire
portabilité	faible, aucun n'est imposé... *merci PEP249 !*
	psycopg *pyformat format*
	sqlite3 *qmark numeric*

```
curs.execute("SELECT * FROM Auteur WHERE nom LIKE ?", ("A%",))              # qmark
curs.execute("SELECT * FROM Auteur WHERE nom LIKE :1", ("A%",))             # numeric
curs.execute("SELECT * FROM Auteur WHERE nom LIKE %s", ("A%",))             # format
curs.execute("SELECT * FROM Auteur WHERE nom LIKE :pat", { "pat":"A%" })    # named
curs.execute("SELECT * FROM Auteur WHERE nom LIKE %(pat)s", { "pat":"A%" }) # pyformat
```

Injection de SQL

Cause erreur d'échappement de valeurs fournies par l'utilisateur

Outil `sqlmap` analyse et exploitation (*e.g.* à l'aveugle)...

```
# NE JAMAIS FAIRE ÇA !
curs.execute("SELECT * FROM Friend WHERE login='"+ who + "'")
curs.execute("SELECT * FROM Friend WHERE login='%s'" % who)
curs.execute("SELECT * FROM Friend WHERE login='%s'".format(who))
curs.execute(f"SELECT * FROM Friend WHERE login='{who}'")

# mais plutôt ça
curs.execute("SELECT * FROM Friend WHERE login=%s", (who, ))
```

Démonstration ! `https://xkcd.com/327/`

- accès aux données... aux méta-données... modification des données...

Requêtes dynamiques

Composition de SQL en évitant les injections

- driver spécifique, par exemple avec `psycopg.sql`
- échappements selon l'utilisation
  - `Identifier` identifiant *table, colonne*
  - `Litteral` valeur *string, int, float*
  - `Placeholder` paramètre de requête à fournir

```
from psycopg import sql

def update_something(conn, tab, col, val):
    with conn.cursor() as curs:
        query = sql.SQL("UPDATE {tab} SET {col} = {val}").format(
            tab=sql.Identifier(tab),
            col=sql.Identifier(col),
            val=sql.Placeholder("val"))
        curs.execute(query, {"val": val})
```


Conclusion

Python DB API

- connexion Python - DB relationnelles
- compléments et extensions : copy, 2PC, BLOB...
- standard sans être portable ! sécurité !

Abstractions de plus haut niveau pour dissimuler DB API

ORM *Object Relational Mapper* `SQLAlchemy Django PeeWee`

- cache SQL dans une syntaxe Python (DDL, DML...)
- plus complexe, portable, SQL parfois moyen, un peu plus lent...

YeSQL encapsulation de SQL `AioSQL AnoDB`

- cache l'interaction SQL/Python dans des fonctions
- très simple, connaissance et puissance de SQL, plus statique...

AnoDB

SQL en Python — *YeSQL!*

AnoDB / AioSQL / DB API

- connexion/déconnexion à la base de données
- définition de fonctions par requêtes `-- name: add-stuff`
  - passage de paramètres (valeurs) nommés `(key, val)`
  - utilisation des paramètres format `named` `:key :val`
  - contrôle des retours : valeur vs tuple vs dict vs list `^ $ !`
- support SQLite Postgres MySQL MariaDB SQL Server...
- `cursor` disponible pour prendre la main si nécessaire

```python
from anodb import DB
db = DB("sqlite3", "things.db", "things.sql")    # create connection and load queries
print("things:", db.get_all_things())            # [(1, "Watch"), (2, "Table"), ...]
print("thing #42:", db.get_a_thing(tid=42))      # ("Car", "Dad")
print("#things:", db.count_things())             # 5432
deleted = db.rm_all_things()                     # deleted = 5432
```

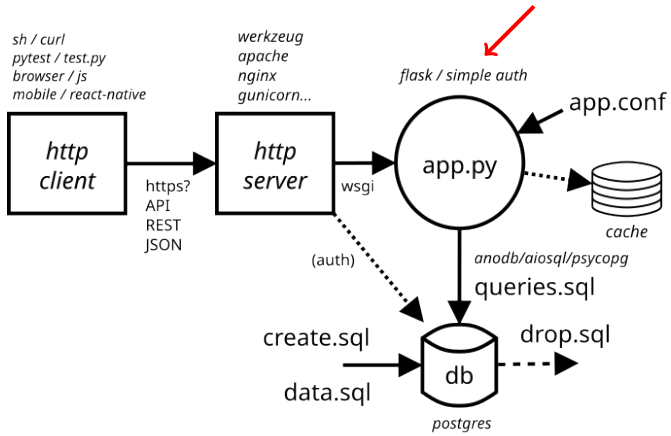

SQL en Python — définition des requêtes, suffix

Requêtes *fixées*		paramètres `:param` pour des valeurs
	SELECT	liste de tuples
^	SELECT	un seul tuple
$	SELECT	une seule valeur
!	INSERT UPDATE DELETE (sans RETURNING)	nombre de lignes

```sql
-- name: get_all_things()
SELECT tid, tname FROM Thing ORDER BY 1;

-- name: get_a_thing(tid)^
SELECT tname, oname FROM Thing JOIN Owner USING (oid) WHERE tid = :tid;

-- name: count_things()$
SELECT COUNT(*) FROM Thing;

-- name: rm_all_things()!
DELETE FROM Thing WHERE TRUE;
```


Flask et FlaskSimpleAuth

HTTP en Python — WSGI



Frameworks Python REST ou Web

popularité décroissante		*téléchargements par mois, janvier 2025*
Flask	micro	93.4 M/mo
FastAPI	micro async	72.4 M/mo
Tornado	micro	42.2 M/mo
Django	web, ORM	14.9 M/mo
Bottle	micro	3.5 M/mo
Pyramid	web	1.9 M/mo
Sanic	micro	0.9 M/mo
Falcon	micro	0.5 M/mo

HTTP en Python — `Flask` et `FlaskSimpleAuth`

`Flask` — framework WSGI

- configuration de l'application via chargement de code python
- routage : méthode + chemin → fonction
- récupération des paramètres (HTTP/JSON, chemin)
- génération de réponses JSON, HTML (via template) ; status HTTP
- *event hooks* : avant, après une requête

`FlaskSimpleAuth` — extension `Flask`

- gestion automatique des paramètres typés (HTTP/JSON, chemin)
- authentification très configurable
- autorisations explicites sur les routes
- utilitaires : caches, références...

HTTP en Python — Usage général

- initialisation de la classe — `Flask`
- divers options de configuration — `app.config`
- décorateurs méthode/route → fonction — `route get post`
- autorisations — `authz`

```python
from FlaskSimpleAuth import Flask
app = Flask("demo")
app.config.from_envvar("APP_CONFIG")  # fichier de conf EXTERNE

@app.get("/some/path", authz="OPEN")
def get_some_path():
    return {"msg": "some path"}, 200

@app.get("/other/path", authz="OPEN")
def get_other_path():
    return {"hello": "other path"}, 200
```


HTTP en Python — chemin, méthode, paramètres

- tronçons typés `int float str path uuid...` — `name i`
- paramètres **obligatoires** si pas de valeur par défaut — `j`
- paramètres **facultatifs** si valeur par défaut — `k`
- typage des paramètres **obligatoire** pour les conversions automatiques

```python
@app.post("/users/<name>", authz="OPEN")
def post_users_name(name: str):
    return {"name": name, "msg": f"bonjour {name} !"}, 201

@app.get("/add/<i>", authz="OPEN")
def get_add_i(i: int, j: int, k: int = 0):
    return str(i+j+k), 200

@app.get("/users", authz="OPEN")
def get_users(flt: str|None = None):
    return users_filtered(flt) if flt else users_all()
```

HTTP en Python — Retours

- résultat *et* statut HTTP — 200 par défaut
- fonction `jsonify` : *automatique* pour `dict`... mais nécessaire pour `list`
- voir aussi la classe `Response`

```python
from FlaskSimpleAuth import jsonify, Response

@app.get("/msg/<p>", authz="OPEN")
def get_msg(p: path):
    return jsonify(f"hello {p} !"), 200

@app.route("/bad", methods=["BAD"], authz="OPEN")
def bad_bad():
    return Response("bad bad bad!", status=500)
```


HTTP en Python — Hooks

- enregistrement de fonctions exécutées
- avant une requête — `before_request`
- ou après — `after_request`

```python
import logging
log = logging.getLogger("app")

def audit_trail(res: Response) -> Response:
    log.info(f"result code {res.code}")
    return res

app.after_request(audit_trail)
```


HTTP en Python — Configuration

Authentification — directives `FSA_*`

FSA_AUTH	none httpd basic digest param token...
FSA_REALM	authentication realm
FSA_TOKEN_*	directives pour l'authentification par token
FSA_PASSWORD_*	password management option (algo, params)

```python
# exemple de configuration pour FlaskSimpleAuth
FSA_AUTH = "basic"
FSA_REALM = "comics"
FSA_TOKEN_CARRIER = "bearer"

import logging
FSA_LOGGING_LEVEL = logging.DEBUG
```


HTTP en Python — Authentification

Param Authentication — *login mdp en paramètres*

```
DELETE /users/hobbes HTTP/1.1
Host: www.comics.net
Content-Type: application/json
Content-Length: 33

{"USER":"calvin","PASS":"hobbes"}
```

Basic Authentication — *login mdp en base64*

```
DELETE /users/hobbes HTTP/1.1
Host: www.comics.net
Authorization: Basic Y2FsdmluOmhvYmJlcw==
```

Bearer Authentication — *token signé*

```
DELETE /users/hobbes HTTP/1.1
Host: www.comics.net
Authorization: Bearer comics:calvin:20380119031407:1b098331931e6059
```

HTTP en Python — Démarrage/arrêt

Pour des tests

- commande principale : `flask`
- application via option : `--app=...`
- sous-commandes : `run routes shell`
- autres options ... : `--host=...`
- parfois variables d'environnement
- arrêt brutal : `pkill flask`

```
# fichier de configuration via l'environnement
export APP_CONFIG="./app.conf"
# démarrage du processus avec redirection des traces dans "app.log"
flask --debug --app="app.py" run --host="0.0.0.0" >> app.log 2>&1 &
# dont on garde le numéro du processus pour envoyer des signaux
echo $! > app.pid
```


Tests avec pytest

Tests internes

- objet `test_client()` de `Flask`
- méthodes `get post put patch delete...`
- entêtes un peu à la main...

Tests externes

- lancer le serveur et lui envoyer des requêtes !
- utiliser l'objet `Session` du module `requests`
- permet de tester un serveur déployé

Authentification ?

Tests internes

Fichier `test_interne.py`

- utilisation de `app.test_client()`

```
import pytest
from app import app

@pytest.fixture
def client():
    with app.test_client() as c:
        yield c

def test_stuff(client):
    r = client.post("/stuff", data={"stuff": "Book"})
    assert r.status_code == 201
    sid = r.json["sid"]
    r = client.get("/stuff")
    assert r.status_code == 200 and b'"Book"' in r.data
    r = client.delete(f"/stuff/{sid}")
    assert r.status_code == 204
```


Tests externes

Fichier `test_externe.py`

- utilisation du module `requests`

```
from requests import Session
requests = Session()  # use persistant connections!

import os
URL = os.environ["APP_URL"]

def check_api(method: str, path: str, status: int, **kwargs):
    r = requests.request(method, URL + path, **kwargs)
    assert r.status_code == status
    return r

def test_stuff():
    r = check_api("POST", "/stuff", 201, data={"stuff": "Table"})
    sid = r.json["sid"]
    assert r'"Table"' in check_api("GET", "/stuff", 200).text
    check_api("DELETE", f"/stuff/{sid}", 204)
```

Tests mixtes

Fichier `test_mix.py`

- utilisation du module `FlaskTester` — env `FLASK_TESTER_APP`
  - si URL : tests externes — `http://localhost:5000`
  - si package python : tests internes — `app`
- gestion de l'authentification, des cookies...

```python
import pytest
from FlaskTester import ft_client, ft_authenticator

def test_stuff(ft_client):
    res = ft_client.post("/stuff", 201, data={"stuff": "Book"})
    sid = res.json["sid"]
    ft_client.get("/stuff", 200, b'"Book"')
    ft_client.delete(f"/stuff/{sid}", 204)
```


Conseils

Design de tests pour une API REST

Propriétés

systématiques	combinaisons méthodes × routes
complets	code *et* résultat attendus
échecs	vérifier les erreurs ! oublis, typage...
droits	authentification, autorisations...
indépendants	création/destruction des données de tests
nettoyage	possible des données de tests
simples	à lancer...
rapides	à exécuter...
automatiques	CI/CD...

Conseils

Conception — DB, API

- besoins réels vs imaginaires
- modélisation des données, données de test
- API REST (y compris erreurs), AAA

Répartition des tâches — client vs serveur

- client : tri pour l'affichage
- serveur : requête déterministes

Séparation API et logique applicative

- fonctions et classes pour les aspects *métiers*
- routes focalisées sur les aspects API : HTTP, JSON

Conseils

Tests systématiques

- méthodes, chemins, paramètres...
- 2xx, 4xx (droits !) pas de 5xx en fonctionnement normal ?

Performance

- génération de données, scenarii
- différents niveaux : app, API, DB
  dénormalisation, factorisation, cache, index...

Automatisation

locale scripts, outils `make docker`...

CI *Continuous integration*, test automatiques sur `git push` ou PR

CD *Continuous delivery*, déploiement de l'application, annulations ?

CRUDS

Exemple CRUDS — Fichiers

Fichiers applicatifs — *Python, config, SQL*

app.py	implémentation de l'API REST
database.py	gestion de la base de données
model.py	définition des types
app.conf	configuration de l'application (Python) spécifique à l'instance lancée, chargée par Flask
create.sql	création du schéma la base de données
data.sql	données initiales (pour les tests par exemple)
drop.sql	efface les tables
queries.sql	requêtes pour `AnoDB`

Exemple CRUDS — Python Virtual Environment

Modules python

flasksimpleauth	extension flask
anodb psycopg	base de données Postgres
bcrypt	authentification par mot de passe
pytest flasktester	tests automatiques

```
# create a flask venv
python -m venv venv
source venv/bin/activate
pip install \
    flasksimpleauth anodb psycopg bcrypt \
    flasktester pytest
```

Exemple CRUDS — Fichiers SQL

Administration

- initialisation du schéma de la base de données

```
createdb stuff
psql -1 -f create.sql -f data.sql stuff
```

- nettoyage (version très rapide...)

```
dropdb stuff
```

create.sql

```sql
CREATE TABLE IF NOT EXISTS
  Stuff(sid SERIAL PRIMARY KEY, stuff TEXT UNIQUE NOT NULL);
```

data.sql

```sql
INSERT INTO Stuff(stuff) VALUES ('Chair'), ('Desk');
```

drop.sql

```sql
DROP TABLE IF EXISTS Stuff;
```


Exemple CRUDS — Fichier *app.conf*

Contenu — *Python*

- variables de l'application : initialisation, connexion...

```python
import psycopg
DATABASE = { # connection AnoDB / AioSQL / PsycoPg
    "db": "psycopg",
    "conn": "dbname=stuff application_name=stuff-backend",
    "queries": "queries.sql",
    "row_factory": psycopg.rows.dict_row,
}

# FlaskSimpleAuth
FSA_MODE = "prod"
FSA_ERROR_RESPONSE = "json:error"
FSA_AUTH = "basic"
FSA_DEFAULT_CONTENT_TYPE = "application/json"

import logging
FSA_LOGGING_LEVEL = logging.DEBUG
```


Exemple CRUDS — Fichier *app.py*

Initialisations

- importations et création de l'application — `app`
- chargement de la configuration via l'environnement — `APP_CONFIG`
- initialisation des modules — `init_app`

```python
import logging
logging.basicConfig()
log = logging.getLogger("stuff")
log.setLevel(logging.DEBUG)

from FlaskSimpleAuth import Flask, jsonify, CurrentUser, err as error
app = Flask("stuff")
app.config.from_envvar("APP_CONFIG")

import database
database.init_app(app)
from database import db
```


Exemple CRUDS — Fichier *database.py*

Connexion base de données

- utilisation de AnoDB et Reference — `db`
- commit automatique via un *hook* — `db_commit`

```python
import FlaskSimpleAuth as fsa  # type: ignore
import anodb  # type: ignore

db = fsa.Reference()

def db_commit(res: fsa.Response) -> fsa.Response:
    if res.status_code < 400:
        db.commit()
    else:  # 4xx user errors, 5xx server errors
        db.rollback()
    return res

def init_app(app: fsa.Flask):
    db.set(anodb.DB(**app.config["DATABASE"]))
    app.after_request(db_commit)
```

Exemple CRUDS — start/stop

Fonctionnement de Flask

- configuration via environnement et fichier
- commande `flask` pour tests locaux
  démarrage/arrêt du processus, serveur `werkzeug`

start.sh

```
# fichier de configuration via l'environnement
export APP_CONFIG="./app.conf"
# démarrage du processus avec redirection des traces dans "app.log"
flask --debug --app="app.py" run --host="0.0.0.0" >> app.log 2>&1 &
# dont on garde le numéro du processus pour envoyer des signaux
echo $! > app.pid
```

stop.sh

```
# arrêt du processus flask
kill $(cat app.pid)
rm -f app.pid
```


Exemple CRUDS — version

GET /info

- teste le fonctionnement de base : HTTP, requête SQL

queries.sql

```
-- name: now()$
SELECT CURRENT_TIMESTAMP;
```

app.py

```
@app.get("/info", authz="OPEN")
def get_info():
    # NOTE json automatique pour les dictionnaires
    return {"app": app.name, "user": app.current_user(), "now": db.now()}, 200
```


Exemple CRUDS — test version

test

```
# récupération des informations de l'application
curl -si -X GET http://0.0.0.0:5000/info
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 69

{"app":"stuff","now":"2025-03-12 10:35:21.279865+01:00","user":null}
```


Exemple CRUDS — S

GET /stuff

- liste des trucs, éventuellement filtrée

queries.sql

```
-- name: get_stuff_all()
SELECT * FROM Stuff ORDER BY 1;

-- name: get_stuff_like(flt)
SELECT * FROM Stuff WHERE stuff LIKE :flt ORDER BY 1;
```

app.py

```
@app.get("/stuff", authz="OPEN")
def get_stuff(flt: str|None = None):
    if flt:
        res = db.get_stuff_like(flt=flt)
    else:
        res = db.get_stuff_all()
    return jsonify(res), 200
```

Exemple CRUDS — test S

test

```
# tous les trucs
curl -si -X GET http://0.0.0.0:5000/stuff
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

[{"sid": 1, "stuff": "Chair"},{"sid": 2, "stuff": "Desk"}]
```

---

test

```
# tous les trucs, mais avec un filtre
curl -si -X GET -d flt=% http://0.0.0.0:5000/stuff
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

[{"sid": 1, "stuff": "Chair"},{"sid": 2, "stuff": "Desk"}]
```


Exemple CRUDS — test S

test

```
# les trucs qui commencent pas C
curl -si -X GET -d flt=C% http://0.0.0.0:5000/stuff
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

[{"sid": 1, "stuff": "Chair"}]
```

---

test

```
# les trucs qui commencent par A
curl -si -X GET -d flt=A% http://0.0.0.0:5000/stuff
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

[]
```


Exemple CRUDS — R

GET /stuff/<sid>

- récupération d'un truc, retour 200 ou 404

queries.sql

```
-- name: get_stuff_sid(sid)^
SELECT * FROM Stuff WHERE sid = :sid;
```

app.py

```
@app.get("/stuff/<sid>", authz="OPEN")
def get_stuff_sid(sid: int):
    res = db.get_stuff_sid(sid=sid)
    # _ = res or error(...)
    if res is None:
        error(f"no such sid: {sid}", 404)
    return res, 200
```


Exemple CRUDS — test R

test

```
# le truc 1
curl -si -X GET http://0.0.0.0:5000/stuff/1
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 26

{"sid":1,"stuff":"Chair"}
```

---

test

```
# le truc 5432, qui n'existe pas !
curl -si -X GET http://0.0.0.0:5000/stuff/5432
```

résultat

```
HTTP/1.1 404 NOT FOUND
Content-Type: application/json
Content-Length: 30

{"error": "no such sid: 5432"}
```

Exemple CRUDS C

POST /stuff

- création d'un truc, 201 ou 400

queries.sql

```
-- name: add_stuff(stuff)$
INSERT INTO Stuff(stuff) VALUES(:stuff) RETURNING sid;
```

app.py

```
@app.post("/stuff", authz="OPEN")
def post_stuff(stuff: str):
    sid = db.add_stuff(stuff=stuff)
    return {"sid": sid}, 201
```


Exemple CRUDS test C

test

```
# ajout d'une horloge
curl -si -X POST -d stuff=Clock http://0.0.0.0:5000/stuff
```

résultat

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 10

{"sid":3}
```

---

test

```
# il manque le paramètre "stuff"
curl -si -X POST http://0.0.0.0:5000/stuff
```

résultat

```
HTTP/1.1 400 BAD REQUEST
Content-Type: application/json
Content-Length: 43

{"error": "parameter \"stuff\" is missing"}
```


Exemple CRUDS test C

test

```
# attention, il y a déjà une horloge !
# erreur interne 500 sur vérification des contraintes...
# laisser ? détecter à l'avance et retour 409 Conflict ?
# le plus simple : utiliser ON CONFLIT + RETURNING...
curl -si -X POST -d stuff=Clock http://0.0.0.0:5000/stuff
```

résultat

```
HTTP/1.1 500 INTERNAL SERVER ERROR
Content-Type: application/json
Content-Length: 58

{"error": "internal error caught at parameters on /stuff"}
```


Exemple CRUDS D

DELETE /stuff/<sid>

- destruction d'un truc, 204 ou 404

queries.sql

```
-- name: del_stuff_sid(sid)!
DELETE FROM Stuff WHERE sid = :sid;
```

app.py

```
@app.delete("/stuff/<sid>", authz="OPEN")
def delete_stuff_sid(sid: int):
    # res is the number of deleted rows
    res = db.del_stuff_sid(sid=sid)
    if not res:
        error(f"no such sid: {sid}", 404)
    return "", 204
```

Exemple CRUDS — test D

test

```
# efface le truc 2
curl -si -X DELETE http://0.0.0.0:5000/stuff/2
```

résultat

```
HTTP/1.1 204 NO CONTENT
Content-Type: text/plain
```

---

test

```
# plus de truc 2...
curl -si -X GET http://0.0.0.0:5000/stuff/2
```

résultat

```
HTTP/1.1 404 NOT FOUND
Content-Type: application/json
Content-Length: 27

{"error": "no such sid: 2"}
```


Exemple CRUDS — test D

test

```
# attention, déjà effacé
curl -si -X DELETE http://0.0.0.0:5000/stuff/2
```

résultat

```
HTTP/1.1 404 NOT FOUND
Content-Type: application/json
Content-Length: 27

{"error": "no such sid: 2"}
```


Exemple CRUDS — U

PATCH /stuff/<sid>

- modification d'un truc, 204 ou 400 ou 404

queries.sql

```
-- name: upd_stuff_sid(sid, stuff)!
UPDATE Stuff SET stuff = :stuff WHERE sid = :sid;
```

app.py

```
@app.patch("/stuff/<sid>", authz="OPEN")
def patch_stuff(sid: int, stuff: str):
    res = db.upd_stuff_sid(sid=sid, stuff=stuff)
    if not res:  # no row changed!
        error(f"no such sid: {sid}", 404)
    return "", 204
```


Exemple CRUDS — test U

test

```
# modification du truc 3
curl -si -X PATCH -d stuff=Bed http://0.0.0.0:5000/stuff/3
```

résultat

```
HTTP/1.1 204 NO CONTENT
Content-Type: text/plain
```

---

test

```
# le truc 3 a bien été modifié...
curl -si -X GET http://0.0.0.0:5000/stuff/3
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 24

{"sid":3,"stuff":"Bed"}
```

Exemple CRUDS — test U

test

```
# modification du truc 5432, mais il n'exise pas...
curl -si -X PATCH -d stuff=Wardrobe http://0.0.0.0:5000/stuff/5432
```

résultat

```
HTTP/1.1 404 NOT FOUND
Content-Type: application/json
Content-Length: 30

{"error": "no such sid: 5432"}
```


AAA

Authentication – Authorization – Audit

Exemple AAA

AAA

Authentication vérification identité — QUI ?
- serveur web ou framework ou application...
- *login/mdp*, *token*, *certificat*...
- coût de la vérification...

Authorization droit de faire une opération — QUOI ?
- rôles dans l'application – *groupes*
- logique applicative – *données de l'utilisateur*

Audit génération de traces — QUAND ?
- serveur web
- application WSGI
- base de données...

Exemple AAA — *Philosophie*

Sécurité déclarative

- modes d'authentification : `basic` `token`...
- authorizations : conditions requises sur une route
  - auth : authentification simple
  - groupes : applicatifs
  - oauth : délégation des droits, par opérations
  - perms : permissions liées aux objets et opérations
- code applicatif plus simple, conditions garanties

Exemple AAA — données

Authentification avec FlaskSimpleAuth

mode none httpd fake basic param digest token

passe $h(s.p)$ sel s, mdp p

fonctions conçues pour être coûteuses... *400 ms*

create.sql

```
CREATE TABLE IF NOT EXISTS Utilisateur(
  uid SERIAL PRIMARY KEY,
  login TEXT UNIQUE NOT NULL,                -- utilisateur
  pword TEXT NOT NULL,                       -- authentification
  isAdmin BOOL NOT NULL DEFAULT FALSE,       -- autorisation
  created TIMESTAMP DEFAULT CURRENT_TIMESTAMP  -- audit...
);
```

queries.sql

```
-- name: user_perms(login)^
SELECT pword, isAdmin FROM Utilisateur WHERE login = :login;
```


Exemple AAA — *callbacks*

Fonctions de support de FlaskSimpleAuth

- mot de passe ou None — get_user_pass
- appartenance à un groupe — group_check user_in_group

app.py

```
@app.get_user_pass
def get_user_pass(user):
    res = db.user_perms(login=user)
    return res["pword"] if res else None

@app.group_check("ADMIN")
def user_is_admin(user):
    res = db.user_perms(login=user)
    return res["isadmin"] if res else False
```


Exemple AAA — mots de passe

Stockage des mots de passe

- module python passlib — *plaintext bcrypt ldap...*
- fonction de hash bcrypt avec 2^4 *rounds* — *quelques ms*

pass2csv.py

```
import re
import fileinput
import bcrypt

for line in fileinput.input():
    if not re.match(r"^\s*[#$]", line):  # skip comments and blank lines
        w = re.split(r"[ \t]", line.strip(), 2)
        p = bcrypt.hashpw(w[1].encode("UTF8"), bcrypt.gensalt(rounds=4, prefix=b"2b"))
        print(f'"{w[0]}","{p.decode("ascii")}",{w[2] if len(w) > 2 else None}')
```


Exemple AAA — chargement mdp

users.in

```
# login pass raw,list,of,csv,stuff
calvin hobbes TRUE
hobbes susie FALSE
susie derkins FALSE
```

shell

```
python ./pass2csv.py < users.in > users.csv
```

users.csv

```
"calvin","$2b$04$3xpV/IJ7J4tvaL0UDFPiH.VJl.8G1LRj7NHSJTjJJaGbF1Vzog4H2",TRUE
"hobbes","$2b$04$7a5KJkerw/2kKUlLL1FnPehY/Pc9EDem.vV2zq9em61911qDQsbDe",FALSE
"susie","$2b$04$0leBZKmsF.doJYHcOH/2IOnKHrLt/gyKDLx7yj4nbpFzibBoB0pwa",FALSE
```

load.sql

```
\copy Utilisateur(login, pword, isAdmin) from './users.csv' (format csv)
SELECT SETVAL('utilisateur_uid_seq', MAX(uid)) FROM Utilisateur; -- sync seq
```

Exemple AAA — *authorization*

Autorisation via paramètre `authz` du décorateur `route`

- CLOSE valeur par défaut, 403 !
- OPEN aucune authentification requise
- AUTH tous les utilisateurs *authentifiés*
- ... groupes applicatifs

app.py

```python
@app.get("/hello", authz="AUTH")
def get_hello(user: CurrentUser):
    return jsonify(f"hello {user}!"), 200

@app.get("/admin", authz="ADMIN")
def get_admin(user: CurrentUser):
    return jsonify(f"hello admin {user}!"), 200
```


Exemple AAA — test hello

test

```
# pas d'authentification
curl -si -X GET http://0.0.0.0:5000/hello
```

résultat

```
HTTP/1.1 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 41
WWW-Authenticate: Basic realm="stuff", Bearer realm="stuff"

{"error": "missing authorization header"}
```

---

test

```
# utilisateur inexistant
curl -si -X GET -u hacker:secret http://0.0.0.0:5000/hello
```

résultat

```
HTTP/1.1 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 33
WWW-Authenticate: Basic realm="stuff", Bearer realm="stuff"

{"error": "no such user: hacker"}
```


Exemple AAA — test hello

test

```
# mauvais mot de passe
curl -si -X GET -u hobbes:pas-le-bon http://0.0.0.0:5000/hello
```

résultat

```
HTTP/1.1 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 40
WWW-Authenticate: Basic realm="stuff", Bearer realm="stuff"

{"error": "invalid password for hobbes"}
```

---

test

```
# ok, hobbes est un utilisateur
curl -si -X GET -u hobbes:susie http://0.0.0.0:5000/hello
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 16

"hello hobbes!"
```


Exemple AAA — test admin

test

```
# non, hobbes n'est pas admin
curl -si -X GET -u hobbes:susie http://0.0.0.0:5000/admin
```

résultat

```
HTTP/1.1 403 FORBIDDEN
Content-Type: application/json
Content-Length: 35

{"error": "not in group \"ADMIN\""}
```

---

test

```
# ok, calvin est admin
curl -si -X GET -u calvin:hobbes http://0.0.0.0:5000/admin
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 22

"hello admin calvin!"
```

Exemple AAA — register

Enregistrement d'un nouvel utilisateur

- route ouverte à tous, mot de passe, droits par défaut...
- vérification éventuelle ?

queries.sql

```
-- name: add_new_user(login, pword)!
INSERT INTO Utilisateur(login, pword) VALUES (:login, :pword);
```

app.py

```
@app.post("/register", authz="OPEN")
def post_register(login: str, pword: str):
    # FIXME check whether user already exists...
    db.add_new_user(login=login, pword=app.hash_password(pword))
    return "", 201
```


Exemple AAA — test register

test

```
# add new user "moe" with password "secret"
curl -si -X POST -d login=moe -d pword=secret http://0.0.0.0:5000/register
```

résultat

```
HTTP/1.1 201 CREATED
Content-Type: text/plain
Content-Length: 0
```

---

test

```
# calvin already exists
curl -si -X POST -d login=calvin -d pword=comics http://0.0.0.0:5000/register
```

résultat

```
HTTP/1.1 500 INTERNAL SERVER ERROR
Content-Type: application/json
Content-Length: 61

{"error": "internal error caught at parameters on /register"}
```


Exemple AAA — whoami

app.py

```
# affiche l'utilisateur authentifié
@app.get("/whoami", authz="AUTH")
def get_whoami(user: CurrentUser):
    return jsonify(user), 200
```

---

test

```
# route acceptant une authentification "token" ou "basic"
curl -si -X GET -u moe:secret http://0.0.0.0:5000/whoami
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 6

"moe"
```


Exemple AAA — token

Authentification par token

- beaucoup moins coûteux qu'une vérification de mot de passe
  connection avec mdp → token, puis utilisation jusqu'à expiration
- format FSA `realm:user:timestamp:signature`
  exemple *kiva:calvin:20380119031407:bdc82cef86b08aa9607fd16e6a03985f*
- standard JWT RFC 7519
- transport : *bearer*, *cookie*, paramètre...

app.py

```
# création d'un token pour l'utilisateur authentifié
@app.get("/login", authz="AUTH", authn="basic")
def get_login(user: CurrentUser):
    return jsonify(app.create_token(user)), 200
```

Exemple AAA — test token

test

```
# récupération d'un token...
curl -si -X GET -u moe:secret http://0.0.0.0:5000/login
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 44

"stuff:moe:20250312103521:b1972eab4305fe02"
```


Exemple AAA — test token valide

test

```
curl -si -X GET \
  -H "Authorization: Bearer stuff:moe:20250312103521:b1972eab4305fe02" \
  http://0.0.0.0:5000/whoami
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 6

"moe"
```


Exemple AAA — test token invalide

test

```
curl -si -X GET \
  -H "Authorization: Bearer stuff:moe:30250312103521:b1972eab4305fe02" \
  http://0.0.0.0:5000/whoami
```

résultat

```
HTTP/1.1 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 44
WWW-Authenticate: Basic realm="stuff", Bearer realm="stuff"

{"error": "unexpected Authorization header"}
```


Exemple AAA — qui suis-je ?

app.py

```
# utilisateur authentifié par token uniquement
@app.get("/qui-suis-je", authz="AUTH", authn="token")
def get_qui_suis_je(user: CurrentUser):
    return jsonify(user), 200
```

---

test

```
# cette route requiert une authentification "token"
curl -si -X GET -u moe:secret http://0.0.0.0:5000/qui-suis-je
```

résultat

```
HTTP/1.1 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 26
WWW-Authenticate: Bearer realm="stuff"

{"error": "missing token"}
```

Exemple AAA — test token

test

```
curl -si -X GET \
  -H "Authorization: Bearer stuff:moe:20250312103521:b1972eab4305fe02" \
  http://0.0.0.0:5000/qui-suis-je
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 6

"moe"
```


Exemple AAA — permissions

Objets avec un propriétaire

- restriction d'accès aux propriétaires des objets

create.sql

```
CREATE TABLE IF NOT EXISTS Owned(
  oid SERIAL PRIMARY KEY,
  object TEXT NOT NULL,
  owner INTEGER NOT NULL REFERENCES Utilisateur,
  UNIQUE(object, owner)
);

CREATE INDEX IF NOT EXISTS owned_owner ON Owned(owner);
```


Exemple AAA — permissions

app.py

```
@app.get("/owned", authz="ADMIN")
def get_owned():
    res = db.get_owned()
    return jsonify(res), 200


@app.get("/object", authz="ADMIN")
def get_object():
    res = db.get_owned()
    return jsonify(res), 200
```

queries.sql

```
-- name: get_owned()
SELECT oid, object, owner
FROM Owned
ORDER BY 1;
```


Exemple AAA — permissions

test

```
# calvin is an admin
curl -si -X GET -u calvin:hobbes http://0.0.0.0:5000/owned
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

[{"oid": 1, "object": "Box", "owner": 1},{"oid": 2, "object": "Box", "owner": 2},{"oid": 3, "c
```

---

test

```
# susie is not an admin
curl -si -X GET -u susie:derkins http://0.0.0.0:5000/owned
```

résultat

```
HTTP/1.1 403 FORBIDDEN
Content-Type: application/json
Content-Length: 35

{"error": "not in group \"ADMIN\""}
```

Exemple AAA — permissions

test

```
# calvin is an admin
curl -si -X GET -u calvin:hobbes http://0.0.0.0:5000/object
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

[{"oid": 1, "object": "Box", "owner": 1},{"oid": 2, "object": "Box", "owner": 2},{"oid": 3, "c
```


Exemple AAA — permissions

Enregistrement de permissions

- l'utilisatrice *login* peut-elle accéder à l'object *oid* pour l'opération *mode* ?
- `True` *Ok* `False` *403 Forbidden* `None` *404 Not Found*

app.py

```
# a user can access an object they own
@app.object_perms("owned")
def check_owned_perms(login: str, oid: int, _mode):
    res = db.can_access_owned(login=login, oid=oid)
    return res["owned"] if res else None
```

queries.sql

```
-- name: can_access_owned(login, oid)^
SELECT u.login = :login AS owned
FROM Owned AS o JOIN Utilisateur AS u ON (o.owner = u.uid)
WHERE o.oid = :oid;
```


Exemple AAA — permissions

Accès à un object particulier

- dont on est propriétaire. . .

app.py

```
@app.get("/object/<oid>", authz=("owned", "oid"))
def get_object_oid(oid: int):
    res = db.get_object_oid(oid=oid)
    return res, 200
```

queries.sql

```
-- name: get_object_oid(oid)^
SELECT oid, object
FROM Owned
WHERE oid = :oid;
```


Exemple AAA — permissions

test

```
# calvin can see his Box
curl -si -X GET -u calvin:hobbes http://0.0.0.0:5000/object/1
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 25

{"object":"Box","oid":1}
```

---

test

```
# hobbes cannot see calvin's Box
curl -si -X GET -u hobbes:susie http://0.0.0.0:5000/object/1
```

résultat

```
HTTP/1.1 403 FORBIDDEN
Content-Type: application/json
Content-Length: 50

{"error": "permission denied on owned:[1] (None)"}
```

Enregistrement de permissions

- l'utilisateur *login* peut-il accéder à l'utilisateur *uid* pour l'opération *mode* ?

app.py

```python
# a user can access data related to them
@app.object_perms("user")
def check_user_perms(login: str, uid: int, _mode):
    res = db.can_access_user(login=login, uid=uid)
    return res["self"] if res else None
```

queries.sql

```sql
-- name: can_access_user(login, uid)^
SELECT u.login = :login AS self
FROM Utilisateur AS u
WHERE u.uid = :uid;
```


Accès aux objets d'un utilisateur

- soi-même...

app.py

```python
@app.get("/owned/<uid>", authz=("user",))
def get_owned_uid(uid: int):
    res = db.get_owned_uid(uid=uid)
    # NOTE jsonify pas indispensable...
    return jsonify(res), 200
```

queries.sql

```sql
-- name: get_owned_uid(uid)
SELECT oid, object, owner
FROM Owned
WHERE owner = :uid;
```


test

```
# calvin can see his own objects
curl -si -X GET -u calvin:hobbes http://0.0.0.0:5000/owned/1
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

[{"oid": 1, "object": "Box", "owner": 1},{"oid": 4, "object": "Tiger", "owner": 1},{"oid": 5,
```

---

test

```
# hobbes cannot access calvin's objects
curl -si -X GET -u hobbes:susie http://0.0.0.0:5000/owned/1
```

résultat

```
HTTP/1.1 403 FORBIDDEN
Content-Type: application/json
Content-Length: 49

{"error": "permission denied on user:[1] (None)"}
```


test

```
# there is no user 42
curl -si -X GET -u hobbes:susie http://0.0.0.0:5000/owned/42
```

résultat

```
HTTP/1.1 404 NOT FOUND
Content-Type: application/json
Content-Length: 29

{"error": "object not found"}
```

AAA avec FlaskSimpleAuth

Simple à mettre en place !

Authentification utiliser `basic` et `token` sur TLS

- forcer un login et l'utilisation de `token`
- durée de vie des tokens selon application ?
- renouvellement ? multi-facteur ?

Authorisations modèle déclaratif et complet `authz`

- modèle par rôle très vite limité
- permissions par objets...

Performances cache avec TTL automatique

Extensions possibles : *recovery code*

Pydantic

dataclass, pydantic...

- définition de structures de données (types)
- éventuellement avec des contraintes de validation

`model.py`

```python
import pydantic

class User(pydantic.BaseModel):
    login: str
    password: str
    isAdmin: bool
    uid: int|None = None  # undefined on POST
```


- passage d'un objet complet de JS à SQL
- étape suivante ? stocker du JSON dans la base...

`queries.sql`

```sql
-- name: new_user_with_object(u)$
INSERT INTO Utilisateur(login, pword, isAdmin)
  VALUES (:u.login, :u.password, :u.isAdmin)
  ON CONFLICT DO NOTHING
  RETURNING uid;
```

`app.py`

```python
@app.post("/users", authz="OPEN")  # FIXME open!
def post_users(user: model.User):
    user.password = app.hash_password(user.password)  # salted hash!
    uid = db.new_user_with_object(u=user)
    if uid is None:
        error(f"conflict on user: {user.login}", 409)
    return {"uid": uid}, 201  # OR jsonify(uid), 201
```

Pydantic — test JSON

test

```
# automatic conversion of string parameters (through JSON)
curl -si -X POST \
  -d 'user={"login":"sis","password":"sis-pass","isAdmin":false}' \
  http://0.0.0.0:5000/users
```

résultat

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 10

{"uid":8}
```


Pydantic — U

queries.sql

```
-- name: change_user_with_object(u)!
UPDATE Utilisateur
  SET login = :u.login,
      pword = :u.password,
      isAdmin = :u.isAdmin
  WHERE uid = :u.uid;
```

app.py

```
@app.put("/users/<uid>", authz="OPEN")  # FIXME close!
def put_users(uid: int, user: model.User):
    uid == user.uid or error("inconsistent user id", 400)
    user.password = app.hash_password(user.password)  # salted hash!
    res = db.change_user_with_object(u=user)
    res or error(f"no such user: {uid}", 404)
    return "", 204
```


Pydantic — test JSON

test

```
# change sis with JSON data
curl -si -X PUT \
  -H "Content-Type: application/json" \
  -d '{"user":{"uid":8,"login":"sis","password":"sis-PASS","isAdmin":true}}' \
  http://0.0.0.0:5000/users/8
```

résultat

```
HTTP/1.1 204 NO CONTENT
Content-Type: text/plain
```


Pydantic — test JSON

test

```
# change with JSON data
curl -si -X PUT \
  -H "Content-Type: application/json" \
  -d '{"user":{"uid":42,"login":"dad","password":"dad-pass","isAdmin":true}}' \
  http://0.0.0.0:5000/users/42
```

résultat

```
HTTP/1.1 404 NOT FOUND
Content-Type: application/json
Content-Length: 29

{"error": "no such user: 42"}
```

Special Parameter — principe

Paramètres spéciaux générés selon leur type

- passe le nom du paramètre, génération d'un objet
- prédéfinis : `CurrentUser FileStorage Cookie Header...`
- ajout avec `special_parameter`, accès à `request`, `current_app`...

model.py

```python
from pydantic import dataclasses

@dataclasses.dataclass
class CurrentAuth:
    uid: int
    login: str
```

queries.sql

```
-- name: get_current_auth(login)$
SELECT uid FROM Utilisateur WHERE login = :login;
```


Special Parameter — utilisation

app.py

```python
from FlaskSimpleAuth import current_app

@app.special_parameter(model.CurrentAuth)
def current_auth(_: str):
    login = current_app.current_user()
    uid = db.get_current_auth(login=login)
    return model.CurrentAuth(uid=uid, login=login)

@app.get("/hello-you", authz="AUTH")
def get_hello_you(auth: model.CurrentAuth):
    return jsonify(auth), 200
```

test

```
curl -si -X GET -u calvin:hobbes http://localhost:5000/hello-you
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 27

{"login":"calvin","uid":1}
```


Blueprint

Exemple Blueprint — importation

Blueprint

- organisation en plusieurs fichiers d'une application flask
- enregistrement différé des routes dans l'application

app.py

```python
from compute import comp
app.register_blueprint(comp, url_prefix="/cmp")
```

Exemple Blueprint — définition

Authentification avec `FlaskSimpleAuth`

- attention au partage de variables `current_app db`

compute.py

```python
from flask import Blueprint, current_app as app, jsonify
comp = Blueprint("compute", __name__)

@comp.get("/add", authz="AUTH")
def get_add(i: int, j: int):
    return jsonify(i+j), 200

@comp.get("/hash", authz="AUTH")
def get_hash(apass: str):
    return jsonify(app.hash_password(apass)), 200
```


Exemple Blueprint — tests

test

```
# gestion de paramètres
curl -si -X GET -u moe:secret -d i=30 -d j=12 http://0.0.0.0:5000/cmp/add
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 3

42
```

---

test

```
# gestion de paramètres
curl -si -X GET -u moe:secret -d apass="Wow!" http://0.0.0.0:5000/cmp/hash
```

résultat

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 63

"$2b$04$xfGeTwcuo.RYorDbI1RBZudp9o/8JIfHjMaSXDuByLx/Swu4k0du2"
```


Déploiement

(Mise en production)

Cycle de vie d'un *back-end*

Development

- tests en *local* (si possible) : machines de dev assez puissantes, docker...
- tests automatiques à distance : *Continuous Integration/Delivery* (CI/CD)

Production

- mise en ligne du service, base avec données initiales, caches...
- éventuellement environnement de *pre*-production

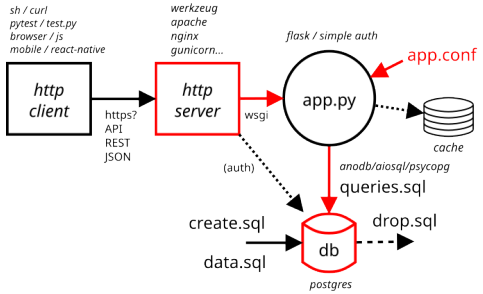
Maintenance et sécurité

- corrections de bugs, nouveaux services, modifications des données...
- problématique : gestion des versions, des secrets...

Fin de vie (Retirement)

- transfert/archivage des données ?

Déploiement d'une application WSGI



base de données	configuration, initialisation, droits
application	fichiers, droits, accès DB. . .
serveur web/WSGI	site, interface WSGI, droits. . .

Postgres — pagode

Compte et base de données applicative — psql

- utilisateur `kiva`, base de donnée `kiva`, droits ?

```
CREATE ROLE kiva WITH
  LOGIN ENCRYPTED PASSWORD 'EfTLPJBijAlPZTOtuk8nAo'
  CONNECTION LIMIT 3
  USER susie, calvin, hobbes, moe;

CREATE DATABASE kiva WITH
  OWNER kiva
  CONNECTION LIMIT 5;

\c kiva
ALTER DEFAULT PRIVILEGES IN SCHEMA PUBLIC
  GRANT ALL PRIVILEGES ON TABLES TO pg_database_owner;
ALTER DEFAULT PRIVILEGES IN SCHEMA PUBLIC
  GRANT ALL PRIVILEGES ON ROUTINES TO pg_database_owner;
ALTER DEFAULT PRIVILEGES IN SCHEMA PUBLIC
  GRANT ALL PRIVILEGES ON SEQUENCES TO pg_database_owner;
```


Postgres — pagode

Accès à la base de données — pg_hba.conf

- autorisation d'accès à partir du serveur
- chiffrement ? impact sur les performances ?

```
# access to database kiva with role kiva from wsgi server with password authn
hostssl kiva kiva 10.101.1.100/32 scram-sha-256
# group access with identd authn
hostssl kiva +kiva 10.201.8.88/32 ident
```


Application — mobapp-srv

Compte utilisateur — kiva@mobapp-srv

- compte utilisateur avec accès par SSH — `kiva`
- répertoire(s) de l'application — `api conf`
- authentification pour la base de données — `.pgpass`

```
sudo adduser --disabled-password kiva
# ~kiva/.ssh/authorized_keys: ...
# ~kiva/.pgpass: pagode:5432:kiva:kiva:EfTLPJBijAlPZTOtuk8nAo
# ~kiva/api: répertoire des l'application, (conf www static venv...)
chmod a+x ~kiva ~kiva/api ~kiva/conf ~kiva/www ...
```

Transfert des fichiers *dev* vers *(pre-)prod* — rsync

```
rsync -av *.py *.sql ... kiva@mobapp-srv.minesparis.psl.eu:api/
```

Création des données initiales (une seule fois en *prod* !) — psql

```
psql -1 -f create.sql -f data.sql "host=pagode user=kiva dbname=kiva"
```

Application — `mobapp-srv`

Lancement de l'application — `app/kiva.wsgi`

```python
# app configuration
from os import environ
environ["APP_NAME"] = "kiva"
environ["APP_CONFIG"] = "/home/kiva/conf/server.conf"
environ["APP_SECRET"] = "..."

# import flask application: app.py / app → application
from app import app as application
```

Configuration de l'application — `conf/server.conf`

```python
import psycopg

DATABASE = {
    "db": "postgres",
    "conn": "host=pagode user=kiva dbname=kiva application_name=kiva-app",
    "queries": "queries.sql",
    "row_factory": psycopg.rows.dict_row,
}
```


Server web : apache, wsgi, venv — `kiva-httpd.conf`


```
WSGIPythonHome "/home/kiva/venv"
WSGIPythonPath "/home/kiva/venv/lib/python3.12/site-packages"
WSGIPythonOptimize 2

<Directory /home/kiva/api>
  WSGIProcessGroup kiva
  WSGIApplicationGroup kiva
  Require all granted
</Directory>

<VirtualHost *:443>
  ServerName kiva.mobapp.minesparis.psl.eu
  # SSL, logs, document root...

  WSGIDaemonProcess kiva \
    lang=C.UTF-8 locale=C.UTF-8 user=kiva group=kiva \
    processes=1 threads=1 display-name=kiva home="/home/kiva/api"

  WSGIScriptAlias "/api" "/home/kiva/api/kiva.wsgi"
  WSGIPassAuthorization On
</VirtualHost>
```


Déploiement en pratique

Équipe *prod* (Claire, Fabien...)

OS	création des comptes bases de données et unix, accès SSH...
Platform	conf apache, venv, WSGI, application...

Équipe *dev* (vous)

- déploiement semi-automatique `make deploy`

```
make deploy
# rsync ssh psql...

curl -si -X GET https://kiva.mobapp.minesparis.psl.eu/api/info
# TADA...
```


Au delà... — `https://www.fullstackpython.com/`

Problématiques — *prod/dev*

Version	de l'application, du schéma, des données...
Performance	monitoring, load balancer, caches, async, index, pool...
Sûreté	monitoring, (haute) disponibilité, sauvegardes, PCA, PRA...
Sécurité	parefeu, logs, surveillance, proxy SSL, secrets...
Maintenance	mises à jours (de sécurité), obsolescence OS/applications

Compétences — *prod*

- installation et maintenance de l'infrastructure matérielle (ou *Cloud*)
  accès, réseau, serveurs, baies de disques, alimentation, refroidissement, ...
- installation et administration de machines (ou services)
  comptes et gestion des accès, audit/sécurité, sauvegardes, maj, obsolescence, ...
- installation et administration des services (interdépendants) nécessaires
  Apache, Python/Flask/..., Postgres, Redis...