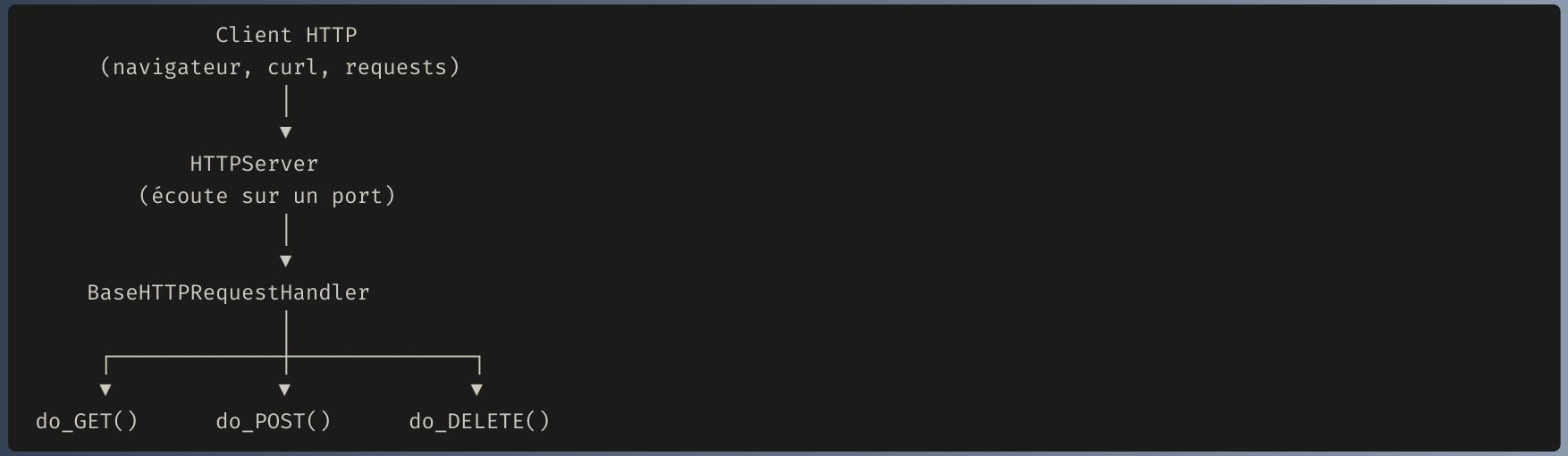
Le module http.server

Le module `http.server` fait partie de la bibliothèque standard de Python. Il permet de créer un serveur HTTP sans installer de framework externe. Les deux classes principales sont :

- `HTTPServer` : le serveur HTTP qui écoute sur un port.
- `BaseHTTPRequestHandler` : la classe que l'on hérite pour traiter les requêtes HTTP.

Ce module est très utile pour comprendre le fonctionnement d'une API REST, mais il n'est **pas recommandé en production**. Il est principalement utilisé à des fins pédagogiques, pour des prototypes ou des outils internes.

Architecture générale



Le serveur reçoit une requête

- puis appelle automatiquement la méthode correspondant au verbe HTTP.

Par exemple :

Requête	Méthode appelée
GET	do_GET()
POST	do_POST()
PUT	do_PUT()
DELETE	do_DELETE()
PATCH	do_PATCH()
HEAD	do_HEAD()

Premier serveur

```python
from http.server import HTTPServer, BaseHTTPRequestHandler

class MonAPI(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-Type", "text/plain")
        self.end_headers()
        self.wfile.write(b"Bonjour")

server = HTTPServer(("localhost", 8000), MonAPI)
print("Serveur démarré")
server.serve_forever()
```

```
http://localhost:8000
```

renvoie

```
Bonjour
```

Rôle de HTTPServer

```
server = HTTPServer(("localhost", 8000), MonAPI)
```

Le constructeur reçoit :

```
(adresse, classe_handler)
```

Ici :

```
localhost:8000
```

est l'adresse, et

```
MonAPI
```

la classe qui traitera chaque requête.

Boucle principale

```
server.serve_forever()
```

Cette méthode :

- écoute le port
- accepte les connexions
- crée un objet `MonAPI`
- appelle automatiquement `do_GET()` ou `do_POST()`.

Hériter de BaseHTTPRequestHandler

```
class MonAPI(BaseHTTPRequestHandler):
```

Toute la logique REST est écrite dans cette classe.

do_GET()

```
def do_GET(self):
```

Cette méthode est exécutée lorsqu'un client envoie :

```
GET / ...
```

Exemple avec curl :

```
curl http://localhost:8000
```

Envoyer une réponse

- Trois étapes sont nécessaires.

1. Code HTTP

```
self.send_response(200)
```

Rappels :

```
200 OK
201 Created
204 No Content
400 Bad Request
401 Unauthorized
404 Not Found
500 Internal Server Error
```

2. Les en-têtes

```
self.send_header(
    "Content-Type",
    "application/json"
)
```

On peut envoyer autant d'en-têtes que souhaité :

```
self.send_header("Server", "Python")
self.send_header("Access-Control-Allow-Origin", "*")
```

3. Fin des en-têtes

```
self.end_headers()
```

Obligatoire.

Après cela on envoie le corps.

Le corps de la réponse

```
self.wfile.write(b"Bonjour")
```

`wfile` est un flux binaire.

Si on possède une chaîne :

```
texte = "Bonjour"

self.wfile.write(
    texte.encode("utf-8")
)
```

Envoyer du JSON

```
import json

data = {
    "nom": "Alice",
    "age": 25
}
json_data = json.dumps(data)
self.send_response(200)
self.send_header(
    "Content-Type", "application/json"
)
self.end_headers()
self.wfile.write(
    json_data.encode()
)
```

Résultat :

```
{
    "nom":"Alice",
    "age":25
}
```

Lire l’URL

La propriété

```
self.path
```

contient :

```
/users
```

ou

```
/users/15
```

Exemple REST

```
class API(BaseHTTPRequestHandler):

    def do_GET(self):
        if self.path == "/users":
            users = [
                {"id":1,"nom":"Alice"},
                {"id":2,"nom":"Bob"}
            ]

            self.send_response(200)
            self.send_header(
                "Content-Type",
                "application/json"
            )

            self.end_headers(
            self.wfile.write(
                json.dumps(users).encode()
            )
        else:
            self.send_error(404)
```

Lire un POST

- Lors d'un POST, le client envoie un corps.
- Il faut connaître sa taille.

```
length = int(
    self.headers["Content-Length"]
)
```

Puis :

```
body = self.rfile.read(length)
```

`body` est de type bytes :

Décoder le JSON

```
import json

body = body.decode()

data = json.loads(body)
```

On obtient :

```
{
    "nom":"Alice"
}
```

Exemple POST

```
def do_POST(self):
    length = int(self.headers["Content-Length"])
    body = self.rfile.read(length)
    user = json.loads(body)
    print(user)
    self.send_response(201)
    self.end_headers()
```

Puis :

```
curl -X POST \
     -H "Content-Type: application/json" \
     -d '{"nom":"Alice"}' \
     http://localhost:8000/users
```

Les en-têtes reçus

Ils sont accessibles via :

```
self.headers
```

Par exemple :

```
print(self.headers["User-Agent"])
print(self.headers["Content-Type"])
print(self.headers["Authorization"])
```

Gestion des paramètres d’URL

Si le client appelle :

```
/users?id=10
```

Alors dans self.path

```
/users?id=10
```

On peut utiliser :

```
from urllib.parse import urlparse, parse_qs
url = urlparse(self.path)
print(url.path)
print(parse_qs(url.query))
```

Résultat :

```
/users
{'id':['10']}
```

Paramètres REST

Pour :

```
GET /users/15
```

on récupère :

```
url = urlparse(self.path)

morceaux = url.path.split("/")
```

Résultat :

```
"", "users", "15"
```

Gérer plusieurs routes

```
def do_GET(self):

    routes = {
        "/users": self.users,
        "/products": self.products
    }

    if self.path in routes:
        routes[self.path]()
    else:
        self.send_error(404)
```

Limites de `http.server`

Bien qu'il permette de construire une API REST fonctionnelle, `http.server` montre rapidement ses limites :

- **Routage manuel** : chaque URL doit être analysée avec `self.path`.
- **Validation des données** : aucune prise en charge intégrée.
- **Gestion des erreurs** : à écrire entièrement à la main.
- **Authentification** : pas de mécanisme intégré (JWT, OAuth, etc.).
- **Sérialisation JSON** : conversion explicite avec `json.dumps()` et `json.loads()`.
- **Concurrence limitée** : `HTTPServer` traite les requêtes de manière séquentielle. Pour gérer plusieurs clients simultanément, on utilise généralement `ThreadingHTTPServer`.
- **Pas de documentation automatique** (comme OpenAPI/Swagger).
- **Pas d'intégration ORM** ni de gestion native des bases de données.

Quand utiliser `http.server` ?

C’est un excellent choix pour :

- comprendre les mécanismes fondamentaux du protocole HTTP ;
- apprendre comment une API REST traite les verbes HTTP, les en-têtes et les corps de requêtes ;
- créer rapidement un prototype ou un petit outil interne.

Pour une application métier ou un service exposé en production, on privilégie généralement des frameworks comme Flask, FastAPI ou Django, qui apportent un routage avancé, une validation des données, une meilleure gestion de la concurrence, la documentation automatique et de nombreuses fonctionnalités prêtes à l’emploi.