UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Flask

Sommaire

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Présentation

Qu'est-ce-que Flask

- Flask est un framework web léger créé par Armin Ronacher sorti le 1er avril 2010
- Il est conçu pour permettre un démarrage rapide et facile, avec la possibilité d'évoluer vers des applications complexes
- Flask propose des suggestions, mais n'impose aucune dépendance ni aucune structure de projet. C'est au développeur de choisir les outils et les bibliothèques qu'il souhaite utiliser

Werkzeug et Jinja2

Flask est basé sur deux outils :

1. **Jinja**
   Jinja est un moteur de template populaire pour Python. Celui-ci permet de faire des rendu de page dynamique
2. **Werkzeug WSGI**
   Werkzeug permet d'implémenter les requêtes, les objets de réponse et d'autres fonctions utilitaires

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Installation

Installation

- Flask nécessite une version de python > 3.7

1. Créer un venv

```
$ mkdir myproject
$ cd myproject
$ python -m venv venv
$ . venv/bin/activate
```

2. Installer Flask `pip install Flask-SQLAlchemy`

Première application Flask

1. Créer un fichier hello.py
2. Ajouter le code qui suit
3. Exécuter la commande `flask --app hello run --debug`

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Concepts

Exemple d'architecture d'une app

L'application est découpée de sorte à faciliter sa maintenance

```
.
├──monblog           : package du code de l'application
│   ├──auth          : package avec la logique d'authentification
│   ├──blog          : package avec la logique du blog
│   ├──static        : fichiers statiques (js, css)
│   └──templates     : templates html
│       ├──auth
│       └──blog
├──.venv
└──tests
```

Configuration de l'application

- La manière la plus propre de créer une application Flask est d'utiliser une fonction. Ce procédé s'appelle ***application factory***.
- Pour cela il faut créer un répertoire et ajouter un fichier `__init__.py`

```
import os

from flask import Flask

def create_app(test_config=None):
    # Ajouter du code pour la configuration de l'application ici
```

Blueprint

Les blueprint permettent de conserver et réutiliser des fonctionnalités du code et contribuent à de meilleures pratiques de développement

Les blueprint servent à :

1. Faciliter l'organisation de projet à grande échelle
2. Augmenter la réutilisabilité du code
3. Utiliser un même blueprint avec des règles d'URL différentes

Création d'un blueprint

- Flask associe des fonctions de vue à des blueprints lors de l'envoi de requêtes et de la génération d'URL d'un point d'extrémité à un autre

```
from flask import Blueprint, render_template

bp = Blueprint('blog', __name__, url_prefix='/blogs')

@bp.route("/")
def index():
    return render_template("blog/index.html")
```

Enregistrer un blueprint

- Dans la fonction de construction de l'application

```
from <app>.blog import blog

app.register_blueprint(blog)
```

- Pour faire référence à l'URL d'un blueprint depuis un autre il suffit d'utiliser `url_for('blog.index')`

Les vues

- Les vues sont des méthodes généralement définies dans une fichier `views.py`
- Les vues sont annotées d'un décorateur route() qui permet d'indiquer l'URL, les paramètres ainsi que la/les méthode(s) http que la vue accepte

Paramètres d'url

- Pour ajouter des paramètres dans une URL il suffit d'ajouter le paramètre entre chevrons
- Les paramètres peuvent être typés (string, int, float, path, uuid)

```
@app.route('/user/<username>')
def show_user_profile(username):
    return f'User {escape(username)}'

@app.route('/post/<int:post_id>')
def show_post(post_id):
    return f'Post {post_id}'
```

Méthode HTTP

- Une vue peut accepter plusieurs méthodes HTTP
- Pour récupérer la méthode il suffit de regarder la valeur présente dans la l'attribut `method` de la request

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()
```

Mise en forme d'un template

- Les vues peuvent rendre des templates à l'aide de la fonction `render_template` dans flask
- La fonction prend en paramètre le nom du template ainsi que des key arguments nécessaires au template

```
@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

Moteur de template : Jinja

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>My Webpage</title>
</head>
<body>
    <ul id="navigation">
    {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
    {% endfor %}
    </ul>

    <h1>My Webpage</h1>
    {{ a_variable }}

    {# a comment #}
</body>
</html>
```

Syntaxe

Les templates utilisent l'extension `.html` ou `.html.jinja`

- **{% ... %}** pour les déclarations
- **{{ ... }}** pour les expressions à afficher à la sortie du template
- **{# ... #}** pour les commentaires non inclus dans la sortie du template
- Pour en savoir plus : documentation

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

SQLAlchemy

SQLAlchemy

- SQLAlchemy est est un ORM python qui donne aux développeurs d'applications toute la puissance et la flexibilité de SQL
- Il fournit une suite complète de modèles de persistance de données, conçus pour un accès efficace et performant aux bases de données

Flask SQLAlchemy

- Flask-SQLAlchemy est une extension pour Flask qui ajoute la prise en charge de SQLAlchemy dans une application
- Elle simplifie l'utilisation de SQLAlchemy avec Flask en mettant en place des objets communs et des modèles d'utilisation de ces objets, tels qu'une session liée à chaque requête web, des modèles et des moteurs

Configurer SQLAlchemy

1. Dans le fichier `__init__.py` ajouter les éléments suivants pour initialiser SQLAlchemy

```
db = SQLAlchemy()

def create_app(test_config=None):
    # code
    app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///project.db"
    db.init_app(app)
```

L'objet db donne accès à la classe **db.Model** pour définir des modèles et à la classe **db.session** pour exécuter des requêtes

Création d'un modèle

- Pour créer un modèle il suffit de créer une classe dans un fichier `models.py` qui hérite de db.Model
- Il faut ensuite définir chaque champ avec son type et ses contraintes

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String, unique=True, nullable=False)
    password_hash = db.Column(db.String, nullable=False)
```

Générer la BDD avec SQLAlchemy

```
# fichier __init__.py
def create_app(test_config=None):
    # ...
    app.cli.add_command(init_db_command)

def init_db():
    db.drop_all()
    db.create_all()

@click.command('init-db')
def init_db_command():
    init_db()
    click.echo('Initialized the database.')
```

Interroger les données

```
# CREATE
user = User()
db.session.add(user)
db.session.commit()

# UPDATE
user.verified = True
db.session.commit()

# DELETE
db.session.delete(user)
db.session.commit()
```

Requête SELECT

- Les requêtes **select** s'éxecutent via **db.session.execute()**

```
user = db.session.execute(db.select(User).filter_by(username=username)).scalar_one()

users = db.session.execute(db.select(User).order_by(User.username)).scalars()
```

- **scalar_one()** renvoie exactement un résultat scalaire (une instance de User) ou lève une exception
- **scalars()** renvoie une liste d'instances

Requêtes pour les vues

- Flask-SQLAlchemy fournit quelques méthodes de requête supplémentaires pour renvoyer des erreurs 404 en cas d'entrées manquantes
- **SQLAlchemy.get_or_404()** : lève une 404 si l'id spécifié n'existe pas
- **SQLAlchemy.first_or_404()** : lève une 404 si la requête ne renvoie aucun résultat, sinon renvoie le 1er résultat
- **SQLAlchemy.one_or_404()** : lève une 404 si la requête ne retourne pas exactement 1 résultat

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Middlewares

Les middlewares de Flask

- Un middleware WSGI est une application WSGI qui enveloppe une autre application afin d'observer ou de modifier son comportement
- Werkzeug fournit quelques middlewares pour des cas d'utilisation courants

Liste des middlewares de Werkzeug

nom	description
X-Forwarded-For Proxy Fix	Ajuste l'environnement WSGI en fonction des en-têtes X-Forwarded- que les proxies devant une application peuvent définir
Serve Shared Static Files	fournit du contenu statique pour les environnements de développement ou les configurations de serveur simples
Application Dispatcher	crée une application WSGI unique qui s'adresse à plusieurs autres applications WSGI montées sur différents URL
Basic HTTP Proxy	Proxy de requêtes sous un URL vers un serveur externe, acheminant les autres requêtes vers l'application
WSGI Protocol Linter	Ce module fournit un middleware intermédiaire qui vérifie le comportement du serveur WSGI et de l'application
Application Profiler	Ce module fournit un middleware qui établit le profil de chaque requête à l'aide du module cProfile

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Optimisations

Flask-Caching

- Le cache dans Flask est implémentée via une extension
- Il utilise la librairie cachelib et s'intègre à Werkzeug via une API
- Il ést également possible de créer son propre système de cache
- Pour installer l'extension : `pip install Flask-Caching`

Principaux système de caches intégrés

nom	description
NullCache	Le cache qui ne cache pas
SimpleCache	Utilise un dictionnaire python local pour la mise en cache
FileSystemCache	Utilise le système de fichiers pour stocker les valeurs mises en cache
RedisCache	base de données de cache en mémoire
MemcachedCache	Système de mise en cache d'objets à mémoire distribuée

Configuration du cache

- Dans la fonction de configuration de l'application, ajouter le code suivant :

```
config = {
    'CACHE_TYPE': 'SimpleCache',
    'CACHE_DEFAULT_TIMEOUT': 300
}

cache = Cache(config)
# ...
cache.init_app(app)
```

Mise en cache des vues

Pour mettre en cache les fonctions d'une vue, il suffit d'utiliser le décorateur cached()

Ce décorateur utilise par défaut request.path pour la clé de cache

```
@app.route("/")
@cache.cached(timeout=50)
def index():
    return render_template('index.html')
```

⚠️ toujours placer le décorateur @cache après la route

Mise en cache des autres fonctions

- Pour mettre en cache des fonctions il est nécessaire de remplacer le préfixe key_prefix, sinon il utilisera le chemin request.path cache_key
- Si une clé n'existe pas dans le cache, une nouvelle entrée clé-valeur sera créée dans le cache

```
@cache.cached(timeout=50, key_prefix='all_comments')
def get_all_comments():
    comments = db.get_all_comments()
    return [c.author for c in comments]
```

Mémoïsation

- La mémorisation est que si vous avez une fonction que vous devez appeler plusieurs fois dans une requête, elle ne sera calculée que la première fois que cette fonction est appelée avec ces arguments

```
class Person(db.Model):
    @cache.memoize(50)
    def has_membership(self, role_id):
        return Group.query.filter_by(user=self, role_id=role_id).count() >= 1
```

Mise en cache de templates

- Par défaut la clé du cache est la concaténation du chemin du template et le début de ligne du bloc
- None permet de mettre le template en cache indéfiniment

```
{% cache 60*5 %}
<div>
    <form>
    {% render_form_field(form.username) %}
    {% render_submit() %}
    </form>
</div>
{% endcache %}
```

Profilage de l'application

- Profiler une application Flask se fait à l'aide du middleware de werkzeug
- Cet outil analyse chaque requête reçue par l'application

```
from werkzeug.middleware.profiler import ProfilerMiddleware
app = ProfilerMiddleware(app)
```

- Les données récoltées peuvent être analysées dans un outil comme SnakeViz

Flask DebugToolbar

- Cette extension ajoute une barre d'outils aux applications Flask contenant des informations utiles pour le débogage
- Installation : `pip install flask-debugtoolbar`

```
toolbar = DebugToolbarExtension()
# ...
app = create_app(<config>)
toolbar.init_app(app)
```

Utiliser des index

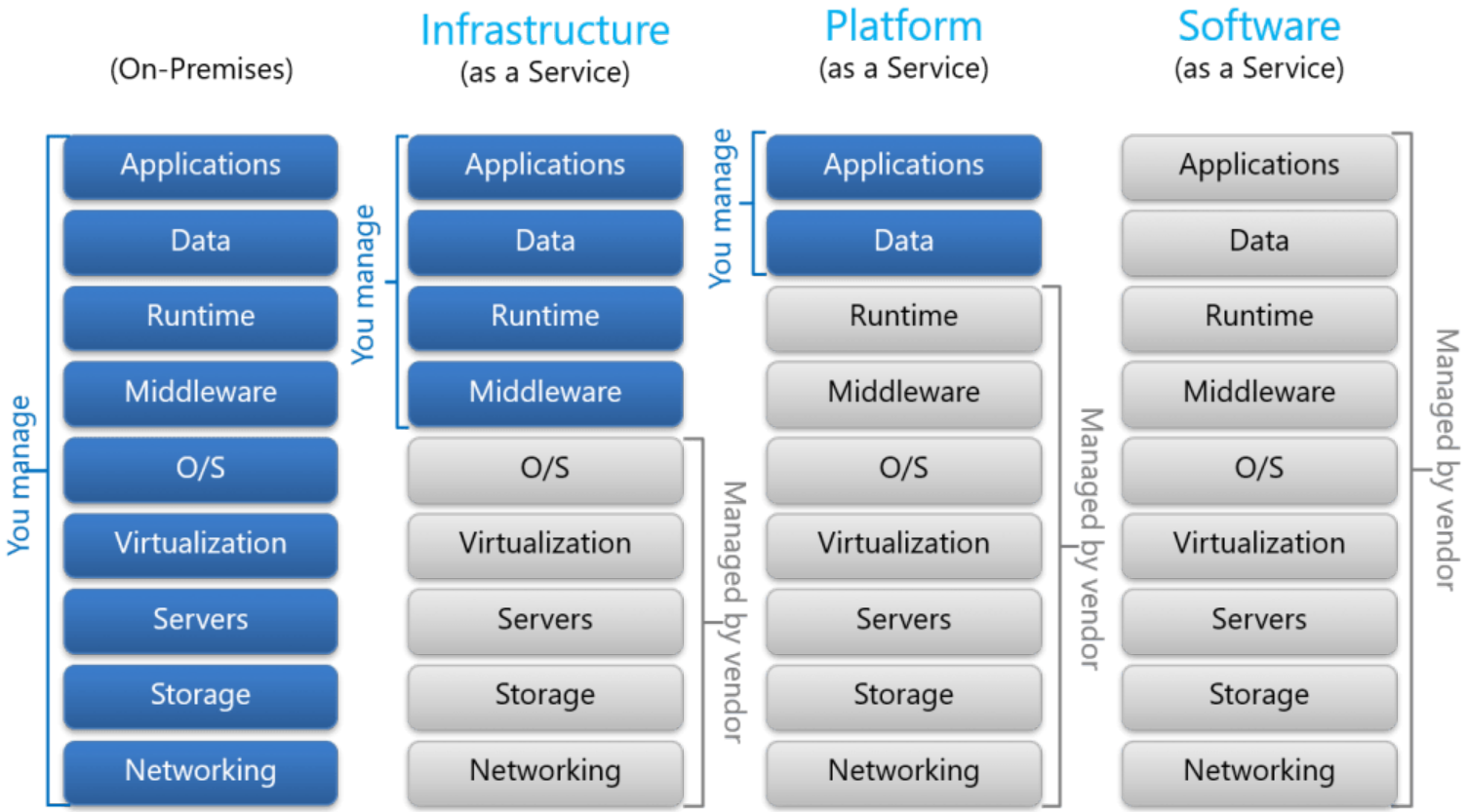
- Un index de base de données est une structure de données qui améliore la vitesse des opérations de recherche de données sur une table de base de données au prix d'écritures et d'espace de stockage supplémentaires pour maintenir la structure de données de l'index

```
class Post(db.Model)
    id = db.Column(db.Integer, primary_key=True)
    author_id = db.Column(db.ForeignKey(User.id), nullable=False)
    created = db.Column(db.DateTime, nullable=False, default=now_utc)
    title = db.Column(db.String, nullable=False, Index=True)
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Déploiement

Les stratégies de déploiement



Flask et WSGI

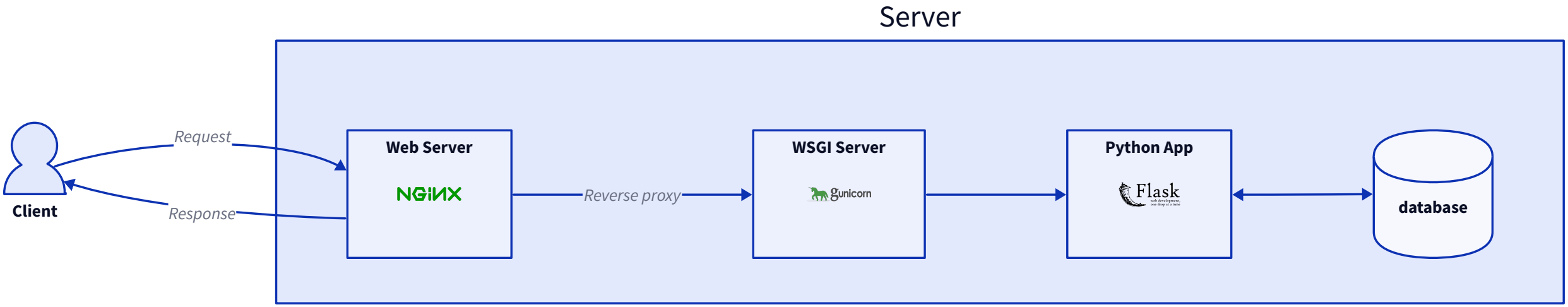
- Un serveur WSGI est utilisé pour exécuter l'application, en convertissant les requêtes HTTP entrantes en WSGI environ et en convertissant les réponses WSGI sortantes en réponses HTTP
- Il existe de nombreux serveurs WSGI et serveurs HTTP, avec de nombreuses possibilités de configuration
- Flask est une application WSGI

Serveur WSGI

- Gunicorn ⭐
- Waitress
- mod_wsgi
- uWSGI

Généralement l'utilisation d'un serveur HTTP dédié (Apache, Nginx) peut s'avérer plus sûr, plus efficace ou plus performant.
Le fait de placer un serveur HTTP devant le serveur WSGI s'appelle un "reverse proxy".

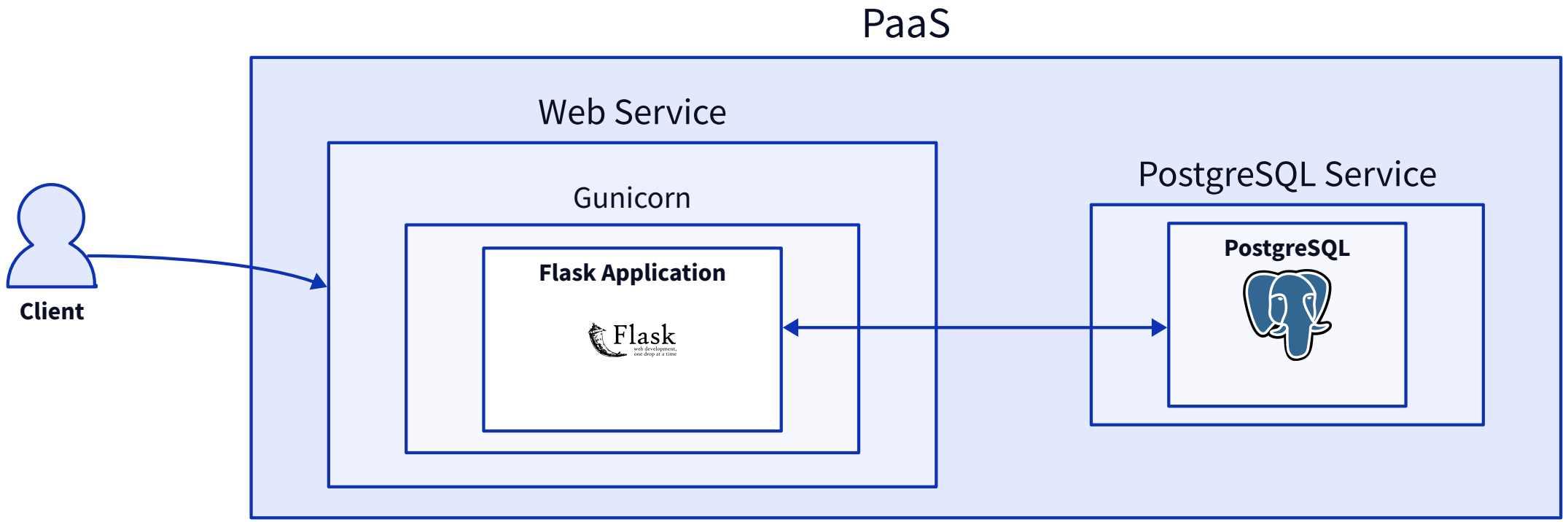
Fonctionnement d'un serveur



Plateforme d'hébergement

- Il existe de nombreux services permettant d'héberger des applications web sans avoir à gérer son propre serveur, son réseau, son domaine, etc
  - PythonAnywhere
  - Google App Engine
  - Google Cloud Run
  - AWS Elastic Beanstalk
  - Microsoft Azure

Fonctionnement d'un PaaS



Dépendances

```
(venv)$ pip install gunicorn
(venv)$ pip install psycopg2-binary

(venv)$ pip freeze > requirements.txt
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention

Des questions ?