

Architecture Logicielle

État d'esprit et méthodologie

Michel VEDRINE

2024

IUT de Vélizy

Format de ce cours

Objectifs

- développer une application de bout en bout ;
- mettre en oeuvre une architecture classique.

Un séance c'est ...

- ~ 30min de cours sur le thème du jour ;
- 2h de développement de l'application.

Site web

<https://kathode.neocities.org>

Généralités

Définition

[...] décrit d'une manière abstraite et schématique les différents éléments d'un système informatique, leurs interrelations et leurs interactions.

Les fondements d'une application



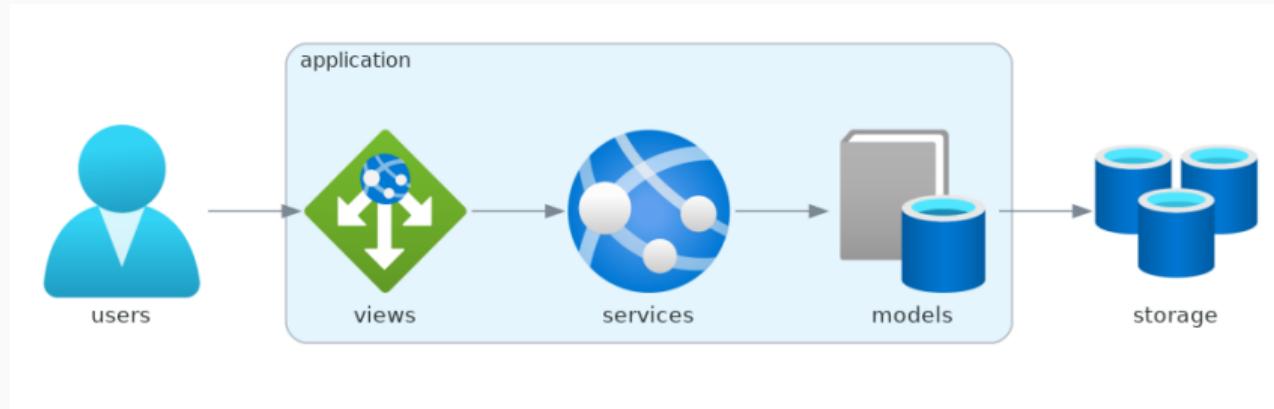
- des espaces de stockage ;
- des modèles ;
- des services ;
- des interfaces pour les clients.

Cas d'une bibliothèque



- stockage : fichiers, base de données
- modèles : Abonné, Livre, Emprunt
- services : s'inscrire, emprunter un livre
- interfaces : bibliothécaire

Architecture cible

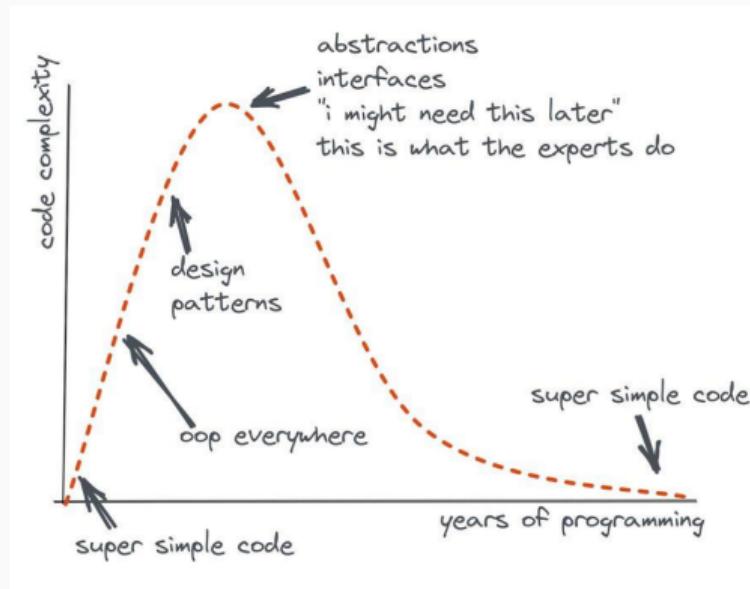


Le cas des designs patterns



- Decorator ;
- Factory ;
- Publish/Subscribe ;
- ...

État d'esprit



- SOC : Separation Of Concern ;
- KISS : Keep It Simple, Stupid ;
- YAGNI : You Ain't Gonna Need It.

L'application

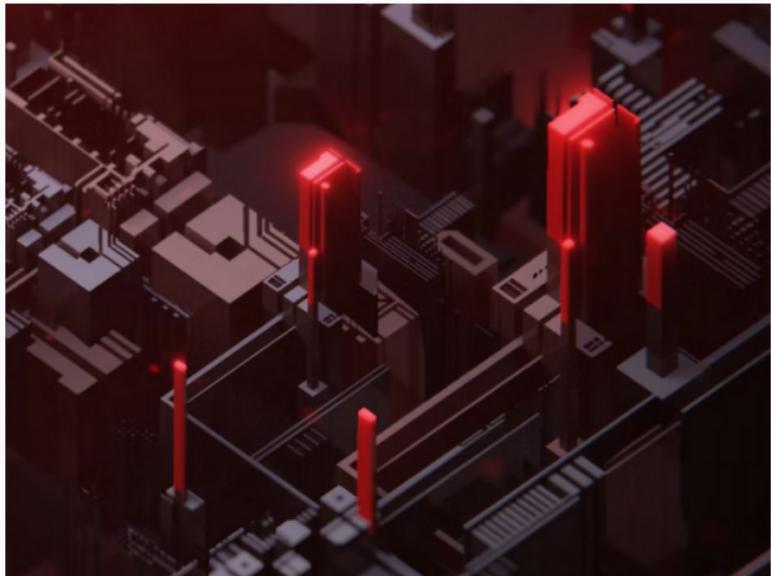
Gestion de TODO

- opérations **CRUD** sur des entités *Todo* ;
- une entité *Todo* comprend à minima :
 - un champ de **texte** pour prendre des notes
 - un champ de **date** de fin de tâche, optionnel ;
 - un champ de **booléen** pour savoir si la tâche est réalisée ou non ;
- **export** et **import** des tâches au format **CSV**.

Planning

1. Interface en ligne de commande
2. Séance libre
 - *template de remise à niveau*
3. Gestion des données
4. Interface web
 - **rendu intermédiaire**
5. Industrialisation
6. Validation & traçabilité
7. Authentification & Autorisation
8. API REST
 - **rendu final**

Technologies



- Python
- Click
- SQLite
- SQLAlchemy
- Flask

Interface en ligne de commande

Click

```
import click

@click.command()
@click.option("--count", default=1)
@click.option("--name", prompt="Your name")
def hello(count, name):
    for x in range(count):
        click.echo(f"Hello {name}!")
```

Click

```
$ python hello.py --count=3
```

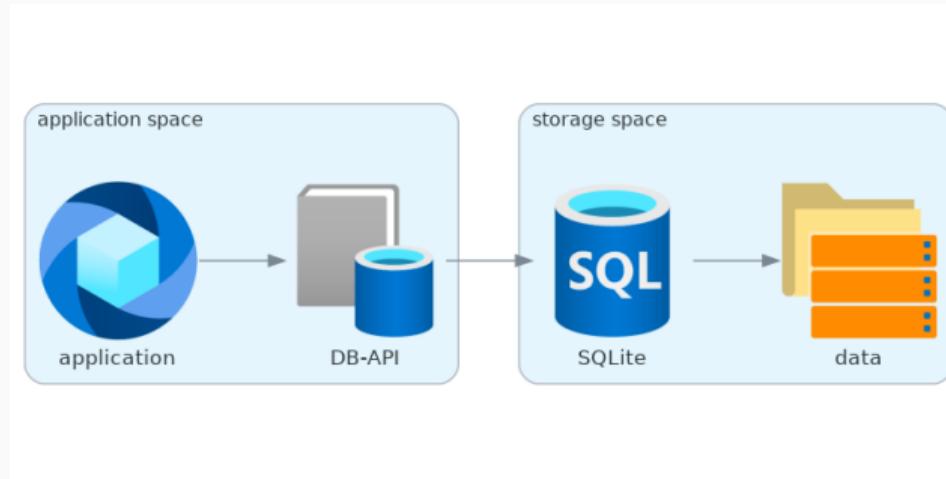
```
  Your name: John
```

```
  Hello John!
```

```
  Hello John!
```

```
  Hello John!
```

SQLite



SQLite

```
import sqlite3

db = sqlite3.connect("tutorial.db")
db.execute("CREATE TABLE movie(title, year, score)")

result = db.execute("SELECT title FROM movie").fetchone()
```

Documentation

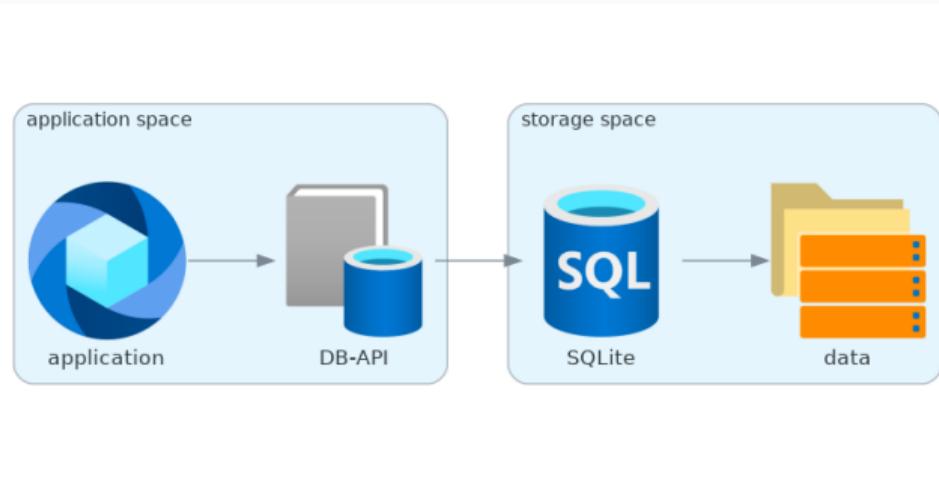
- <https://click.palletsprojects.com>
- <https://docs.python.org/3/library/sqlite3.html>

Objectifs

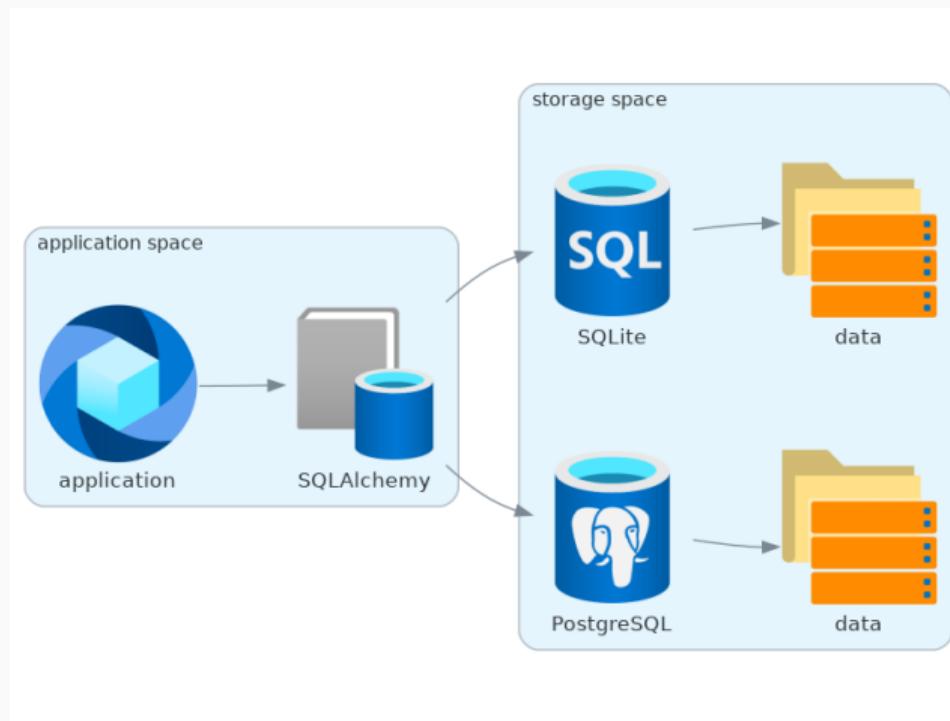
- faire une application de gestion de tâches répondant aux besoins ;
- utiliser un stockage des données avec SQLite ;
- utiliser une interface en ligne de commande avec click.

Gestion des données

Modèle actuel



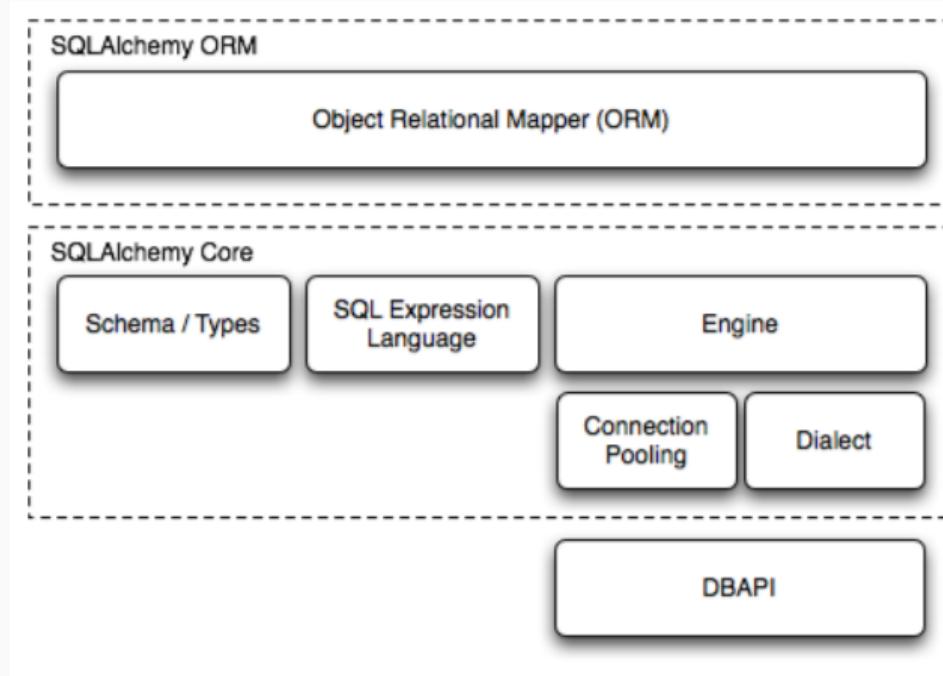
Modèle cible



SQLAlchemy

SQLAlchemy est un toolkit open source SQL et un mapping objet-relationnel (ORM) écrit en Python et publié sous licence MIT.

Architecture de SQLAlchemy



SQLAlchemy Core

```
engine = create_engine("sqlite:///todos.db", echo=True)
metadata = MetaData()

todos_table = Table(
    "todos",
    metadata,
    Column("id", Uuid, primary_key=True, default=uuid.uuid4),
    Column("task", String, nullable=False),
    Column("complete", Boolean, nullable=False),
    Column("due", DateTime, nullable=True)
)
```

SQLAlchemy Core

```
stmt = todos_table.insert().values(  
    task=task,  
    complete=complete,  
    due=due  
)  
  
with engine.begin() as conn:  
    result = conn.execute(stmt)
```

Documentation

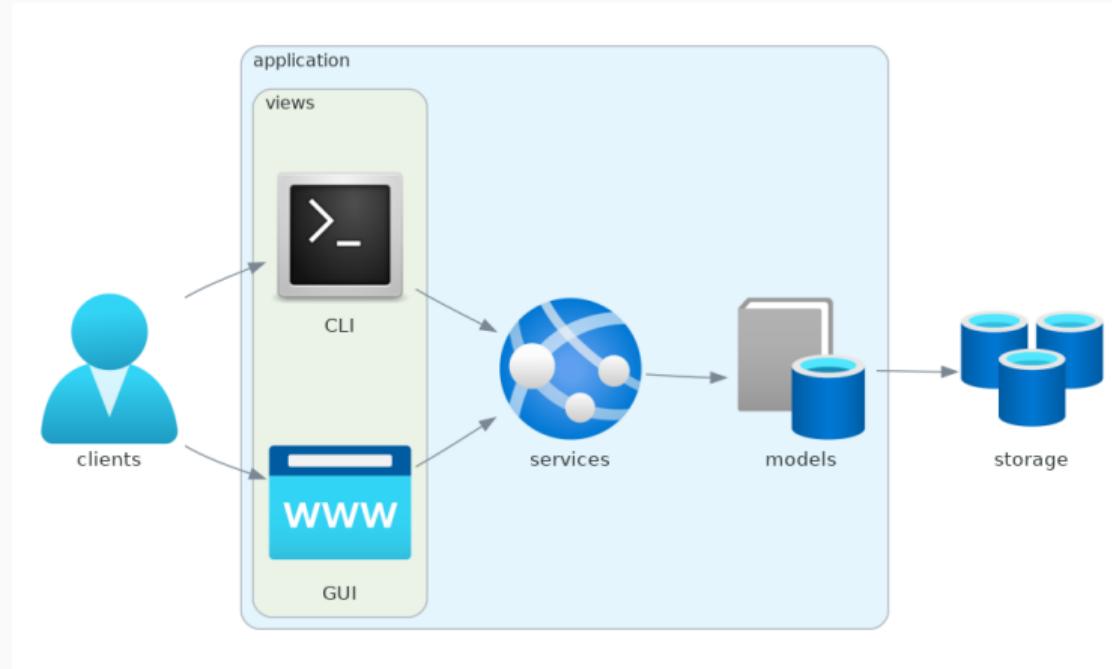
- <https://docs.sqlalchemy.org/>
- <https://docs.sqlalchemy.org/en/20/tutorial/index.html>

Objectifs

- utiliser SQLAlchemy Core ;
- ne **PAS** utiliser SQLAlchemy ORM.

Interface web

Architecture cible



Flask

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications.

Application simple

```
from flask import Flask

app = Flask(__name__)

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

Lancement

```
$ flask --app hello run
 * Serving Flask app 'hello'
 * Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
```

Template HTML

```
from flask import render_template

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

Template HTML avec Flask & Jinja2

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
    <h1>Hello {{ name }}!</h1>
{% else %}
    <h1>Hello, World!</h1>
{% endif %}
```

Arborescence des fichiers

```
.
├── pyproject.toml
├── README.md
└── src
    └── toudou
        ├── __init__.py
        ├── models.py
        ├── services.py
        ├── templates
        │   └── home.html
        └── views.py
```

Lancement de votre application

Après avoir configuré une application Flask :

```
$ pdm add flask  
$ pdm run flask --app toudou.views --debug run
```

Puis se rendre sur <http://localhost:5000>.

Documentation

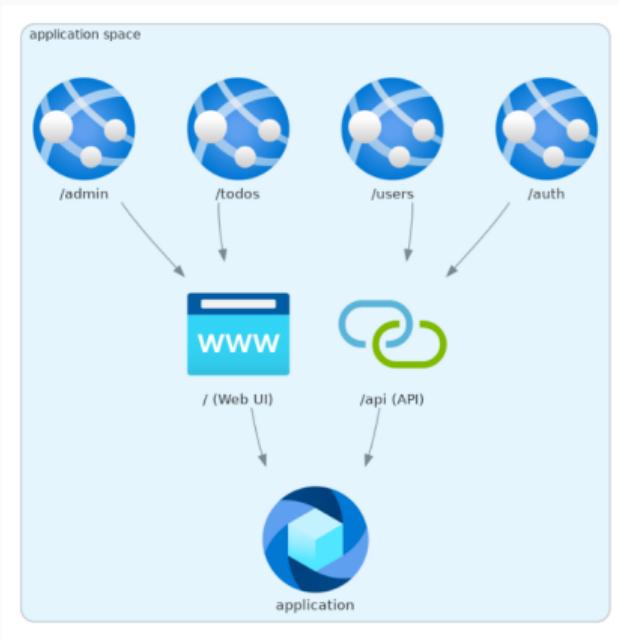
- <https://flask.palletsprojects.com>
- <https://flask.palletsprojects.com/en/3.0.x/quickstart/>

Objectifs

- ajouter une nouvelle vue web ;
- utiliser Flask et Jinja2, **aucun Javascript nécessaire.**

Industrialisation

Modularité : Blueprints



Modularité : Blueprints

```
from flask import Blueprint, render_template

web_ui = Blueprint("web_ui", __name__, url_prefix="/")

@web_ui.route("/<page>")
def show(page):
    return render_template(f"pages/{page}.html")
```

Modularité : Application factory

```
def create_app():
    app = Flask(__name__)

    from toudou.views import api, web_ui
    app.register_blueprint(api)
    app.register_blueprint(web_ui)

    return app
```

Modularité : Application factory

Rien ne change, Flask cherche une fonction `create_app` :

```
$ pdm run flask --app toudou.views --debug run
```

Configuration centralisée

```
engine = create_engine("sqlite:///todos.db", echo=True)
metadata = MetaData()
```

Configuration centralisée

```
from toudou import config

engine = create_engine(config["DATABASE_URL"], echo=config["DEBUG"])
metadata = MetaData()
```

Configuration centralisée

Dans le fichier `src/toudou/__init__.py` :

```
config = dict(  
    DATABASE_URL="sqlite:///todos.db",  
    DEBUG=True  
)
```

dotenv

Collection de fichiers contenant la configuration pour chaque environnement :

- dev.env
- testing.env
- demo.env
- prod.env
- ...

dotenv : format

Dans le fichier dev.env :

```
TOUDOU_DATABASE_URL=sqlite:///todos.db
TOUDOU_DEBUG=True
TOUDOU_FLASK_SECRET_KEY=secret!
```

dotenv : utilisation manuelle

```
$ env $(cat dev.env | xargs) pdm run flask --app ...
```

dotenv : utilisation avec PDM

Dans le fichier pyproject.toml :

```
[tool.pdm.scripts]
_.env_file = "dev.env"
start = "flask --app toudou.views --debug run"
```

dotenv : utilisation dans l'application

Dans le fichier src/toudou/__init__.py :

```
import os

config = dict(
    DATABASE_URL=os.getenv("TOUDOU_DATABASE_URL", ""),
    DEBUG=os.getenv("TOUDOU_DEBUG", "False") == "True"
)
```

dotenv : utilisation avec Flask

```
app.config.from_prefixed_env(prefix="TOUDOU_FLASK")
```

Gestion des erreurs

```
@app.errorhandler(500)
def handle_internal_error(error):
    flash("Erreur interne du serveur", "error")
    return redirect(url_for(".home"))
```

Gestion des erreurs

```
from flask import abort, render_template

@app.get("/todos/create")
def todo_create_form():
    abort(500)
    return render_template("todo_create_form.html")
```

Documentation

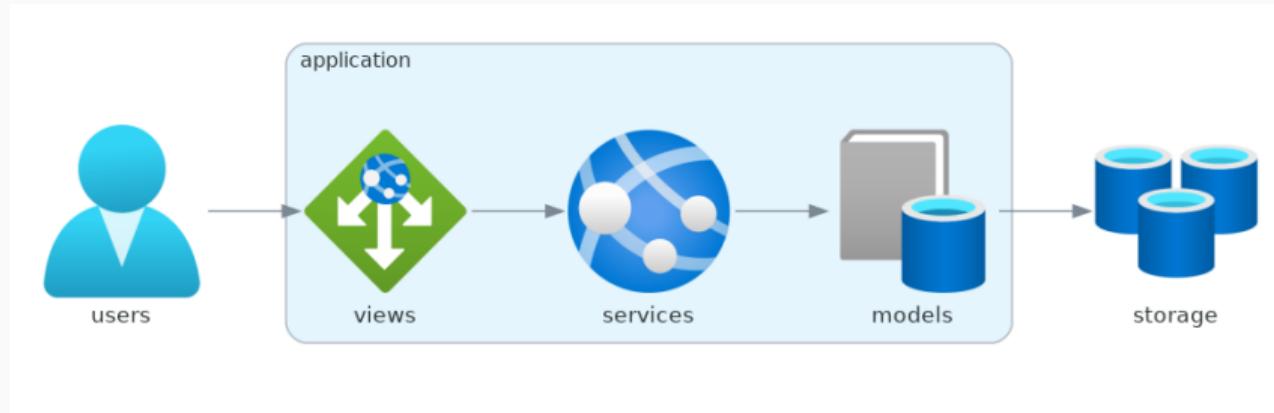
- <https://flask.palletsprojects.com/en/3.0.x/blueprints/>
- <https://flask.palletsprojects.com/en/3.0.x/patterns/appfactories/>
- <https://pdm.fming.dev/latest/usage/scripts/>
- <https://flask.palletsprojects.com/en/3.0.x/errorhandling/#error-handlers>

Objectifs

- basculer sur un blueprint et une application factory
- centraliser la configuration et utiliser dotenv
- gérer les erreurs avec des handlers Flask

Validation & traçabilité

Concepts



Types de validation

Syntaxique : le type de la donnée est respecté (date, entier, texte, etc).

Sémantique : les données ont du sens (date de début inférieur à la date de fin, prix supérieur à 0, etc).

WTForms

WTForms is a flexible forms validation and rendering library for Python web development.

WTForms : déclaration

```
from flask_wtf import FlaskForm
from wtforms import StringField
from wtforms.validators import DataRequired

class MyForm(FlaskForm):
    name = StringField("name", validators=[DataRequired()])
```

WTForms : validation

```
@app.route("/submit", methods=["GET", "POST"])
def submit():
    form = MyForm()
    if form.validate_on_submit():
        return redirect("/success")
    return render_template("submit.html", form=form)
```

WTForms : génération du formulaire

```
<form method="POST" action="/">
    {{ form.csrf_token }}
    {{ form.name.label }} {{ form.name(size=20) }}
    <input type="submit" value="Go">
</form>
```

Intégration Flask

```
$ pdm add flask-wtf
```

Logging

```
import logging

logging.warning("Watch out!")    # will print to the console
logging.info("I told you so")   # will not print anything
```

Logging : configuration

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format"%(asctime)s [%(levelname)s] %(message)s",
    handlers=[
        logging.FileHandler("toudou.log"),
        logging.StreamHandler()
    ]
)
```

Logging : utilisation

```
import logging

@web_ui.errorhandler(500)
def handle_internal_error(error):
    flash("Erreur interne du serveur", "error")
    logging.exception(error)
    return redirect(url_for(".home"))
```

Documentation

- <https://flask-wtf.readthedocs.io>
- https://wtforms.readthedocs.io/en/3.0.x/crash_course/
- <https://docs.python.org/3/howto/logging.html>

Objectifs

- construire et valider les formulaires avec WTForms ;
- configurer le logging et tracer les erreurs du handler 500 ;
- tracer à la fois dans un fichier et vers stdout.

Authentification & Autorisation

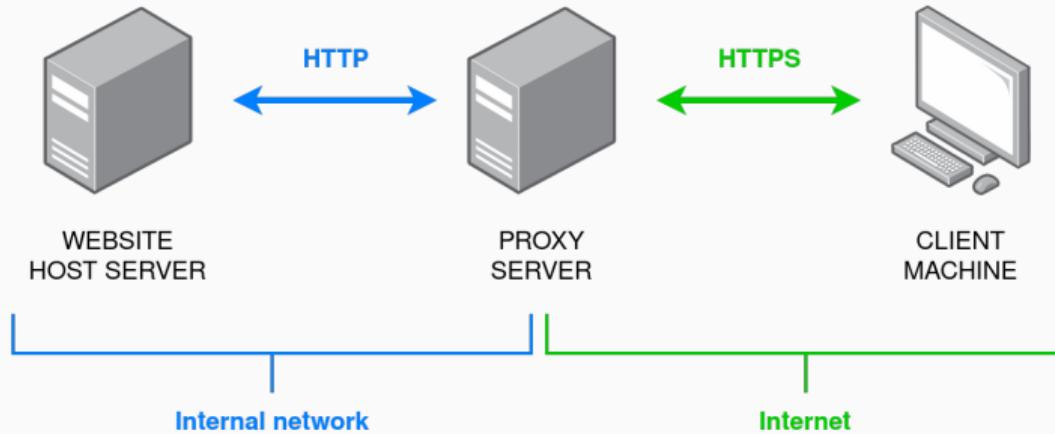
Définitions

- authentification : consiste à vérifier et à valider l'identité des utilisateurs ;
- autorisation : consiste à donner à l'utilisateur la permission d'accéder à une ressource ou à une fonction spécifique.

HTTP Authentication

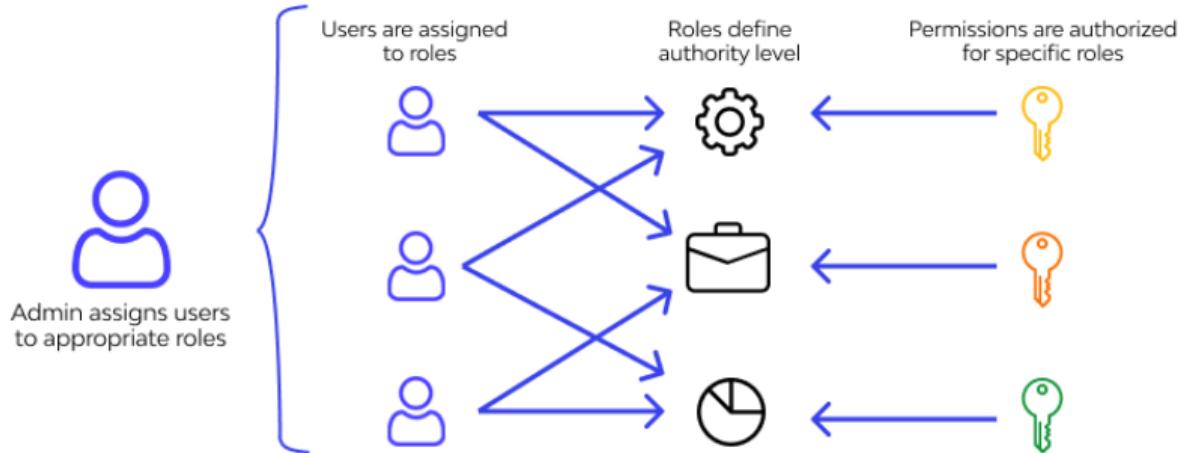


HTTPS



Role-Based Access Control

Role-Based Access Control



Workflow

1. récupération du login + mot de passe
2. comparaison avec les informations du backend
 - les mots de passes sont hashés et salés
3. si OK, récupération des roles associés au login
4. vérification des droits d'accès (role + permission)

flask-httpauth : configuration

```
from flask import Flask
from flask_httpauth import HTTPBasicAuth
from werkzeug.security import generate_password_hash

app = Flask(__name__)
auth = HTTPBasicAuth()

users = {
    "john": generate_password_hash("hello"),
    "susan": generate_password_hash("bye")
}
```

Authentification : configuration

```
from werkzeug.security import check_password_hash

@auth.verify_password
def verify_password(username, password):
    if username in users and \
        check_password_hash(users.get(username), password):
        return username
```

Authentification : utilisation

```
@app.route("/")
@auth.login_required
def index():
    return f"Hello, {auth.current_user()}!"
```

Autorisation : configuration

```
@auth.get_user_roles  
def get_user_roles(user):  
    return user.roles
```

Autorisation : utilisation

```
@app.route("/admin")
@auth.login_required(role="admin")
def admins_only():
    return f"Hello {auth.current_user()}, you are an admin!"
```

flask-httpauth

```
$ pdm add flask-httpauth
```

Documentation

- <https://flask-httpauth.readthedocs.io/en/latest/>

Objectifs

- configurer et utiliser flask-httpauth ;
- configurer et utiliser des rôles :
 - un role `admin` qui accède à tout ;
 - un role `user` qui n'est qu'en lecture seule.

Astuces

Pour se déconnecter avec HTTP Auth :

- écraser avec un mauvais mot de passe : `user:user@localhost:5000`
- nettoyer les données du navigateur : `ctrl+shift+del`

REST

Définition

REST (representational state transfer) est un style d'architecture logicielle définissant un ensemble de contraintes à utiliser pour créer des services web.

Contraintes

- *client-serveur* : communications via HTTP ;
- *stateless* : les informations du client ne sont pas stockées côté serveur ;
- *mise en cache* : les réponses se signalent si elles peuvent être cachées ou non ;
- *interface uniforme* : ressources standardisées ;
- *système en couches* : permet l'extensibilité ;
- *code à la demande* (facultatif) : possibilité d'envoyer du code exécutable depuis le serveur vers le client.

API Web RESTful

Les API RESTful basées sur HTTP sont définies par :

- un URI de base, comme <https://toudou.example.com/todos> ;
- les méthodes HTTP standards : GET, POST, PUT, PATCH et DELETE ;
- un type pour les données : application/xml, application/json, etc.

API Web RESTful : exemple

URI	Verb	Action
/todos	GET	Récupère la collection de ressources <i>Todo</i>
/todos	POST	Crée une ressource dans la collection
/todos	PUT	Remplace toutes les ressources de la collection
/todos	PATCH	Met à jour une partie des ressources de la collection
/todos	DELETE	Supprime la collection

API Web RESTful : exemple

URI	Verb	Action
/todos/13	GET	Récupère la ressource 13 de la collection <i>Todo</i>
/todos/13	POST	Crée une ressource avec l'id 13 dans la collection
/todos/13	PUT	Remplace toutes les données de la ressource 13
/todos/13	PATCH	Met à jour une partie des données de la ressource 13
/todos/13	DELETE	Supprime la ressource 13

API Web RESTful : exemple

URI	Verb	Action
/todos?complete=true	GET	Filtre la collection sur la donnée <i>complete</i>
/users?role=admin	GET	Filtre la collection sur la donnée <i>role</i>

Validation

```
$ pdm add flask-pydantic-spec
```

Validation

```
from flask_pydantic_spec import FlaskPydanticSpec, Request

app = Flask(__name__)
api = FlaskPydanticSpec('flask')

@app.route('/api/user', methods=['POST'])
@api.validate(body=Request(Profile))
def user_profile():
    print(request.context.json)
    return {'text': 'it works'}
```

Validation

```
from pydantic import BaseModel, Field, constr

class Profile(BaseModel):
    name: constr(min_length=2, max_length=40) # Constrained String
    age: int = Field(
        ...,
        gt=0,
        lt=150,
        description='user age(Human)'
    )
```

Authentification

```
from flask import Flask
from flask_httpauth import HTTPTokenAuth

app = Flask(__name__)
auth = HTTPTokenAuth(scheme='Bearer')

tokens = {
    "secret-token-1": "john",
    "secret-token-2": "susan"
}
```

Authentification

```
@auth.verify_token
def verify_token(token):
    if token in tokens:
        return tokens[token]

@app.route('/')
@auth.login_required
def index():
    return "Hello, {}".format(auth.current_user())
```

Tester manuellement

```
$ curl https://localhost:5000/api/todos  
-H "Accept: application/json"  
-H "Authorization: Bearer {token}"
```

Tester avec Redoc ou Swagger

```
app = Flask(__name__)
api = FlaskPydanticSpec('flask')
api.register(app)
```

Puis se rendre sur /apidoc/redoc ou /apidoc/swagger.

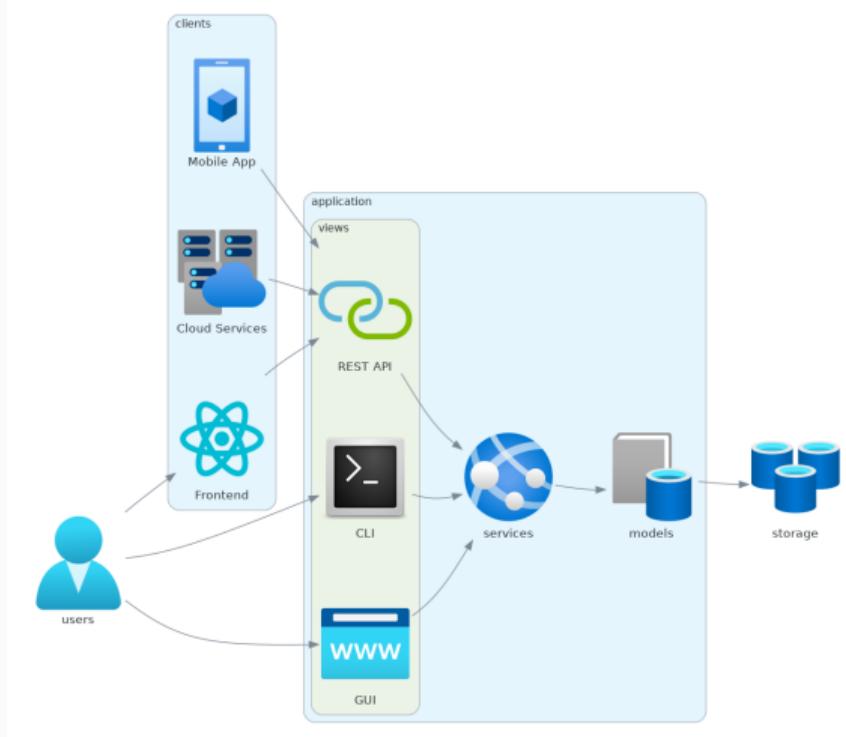
Objectifs

- ajouter un nouveau Blueprint api ;
- ajouter de nouvelles routes respectant les contraintes REST ;
- validation des données avec *flask-pydantic-spec* ;
- authentification par token avec *flask-httpauth*.

Documentation

- <https://developer.mozilla.org/fr/docs/Glossary/REST>
- <https://github.com/turner-townsend/flask-pydantic-spec>
- <https://docs.pydantic.dev/latest/>
- <https://flask-httpauth.readthedocs.io/en/latest/#token-authentication-example>

Architecture finale



Déploiement

Standard WSGI

Créer src/toudou/wsgi.py avec :

```
from toudou.views import create_app
```

```
app = create_app()
```

Serveur d'application

```
$ pdm add gunicorn  
$ pdm run gunicorn toudou.wsgi:app
```

Packaging

```
$ pdm build  
$ ls dist/  
toudou-0.1-py3-none-any.whl  toudou-0.1.tar.gz
```

Dockerfile

```
FROM python:3.10.10-slim-buster

WORKDIR /app

COPY dist/toudou--py3-none-any.whl /app/
RUN pip install *.whl gunicorn

CMD ["gunicorn", "toudou.wsgi"]
```