PROJET Flask
RAPPORT

Licence ingénierie de développement d'applications informatique IDAI

Présenté par :
Aya Zoujari Idrissi

Encadré par :
Mr. OTHMAN BAKKALI

I N T R O D U C T I O N

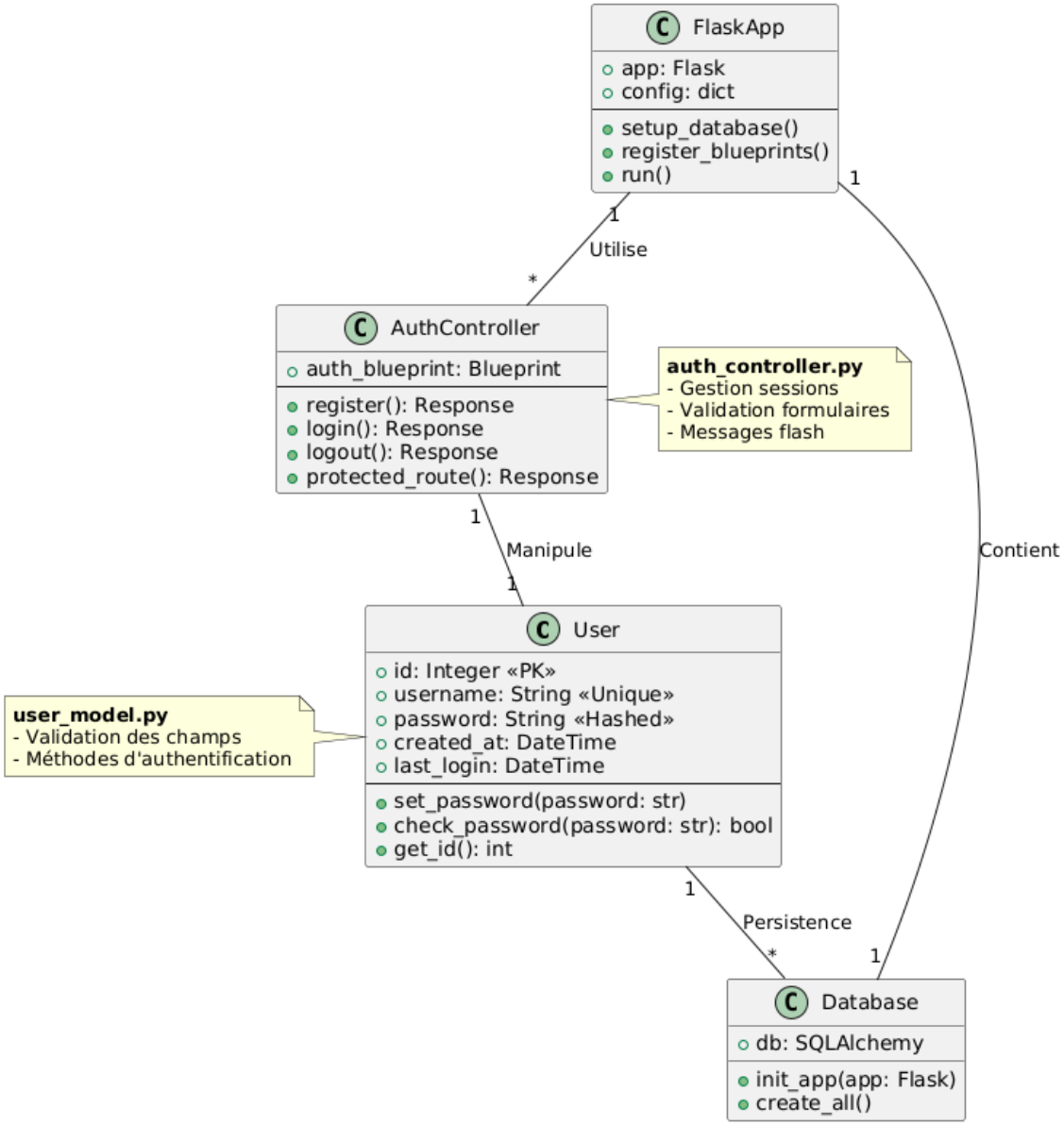
Ce projet vise à concevoir et développer une application web en utilisant le framework **Flask**, en suivant une architecture **MVC** (Modèle-Vue-Contrôleur) pour une structure claire et modulaire. L'objectif est de créer une application dynamique et interactive, intégrant une base de données **SQLite** pour la persistance des données, tout en exploitant les mécanismes de **sessions** pour gérer l'authentification et les préférences utilisateur.
Enfin, l'application sera containerisée avec **Docker** pour assurer une portabilité optimale et un déploiement simplifié dans différents environnements.

O B J E C T I F S

- Organisation du code pour une maintenance simplifiée.
- Stockage et gestion des données via SQLAlchemy.
- Sécurisation des données utilisateur et personnalisation de l'expérience.
- Isolation des dépendances et déploiement reproductible.

CONCEPTION : DIAGRAMME DE CLASSE

Ce diagramme de classes illustre l'architecture MVC (Modèle-Vue-Contrôleur) de l'application Flask, centrée autour de trois composants principaux. La classe User (modèle) représente l'entité utilisateur en base de données, avec des méthodes dédiées à la gestion sécurisée des mots de passe (hachage et vérification).
Le AuthController (contrôleur) orchestre les opérations d'authentification (inscription, connexion, déconnexion) en s'appuyant sur le modèle User et les fonctionnalités de routage de Flask. La classe FlaskApp initialise et configure l'application, intégrant à la fois l'instance SQLAlchemy pour la persistance des données et les blueprints des contrôleurs



S T R U C T U R E D U P R O J E T

```
Flask_Project/
│
├── .venv/                  # Environnement virtuel Python (ignoré par Git)
│
├── controllers/            # Contrôleurs (Blueprints)
│   ├── __pycache__/        # Cache Python (ignoré)
│   ├── auth_controller.py  # Gestion authentification (login/register/logout)
│   └── main_controller.py  # Routes principales (optionnel)
│
├── instance/               # Configurations sensibles (ignoré par Git)
│   └── users.db            # Base de données SQLite (dev)
│
├── models/                 # Modèles de données
│   ├── __pycache__/
│   ├── database.py         # Initialisation de SQLAlchemy (db = SQLAlchemy())
│   └── user_model.py       # Modèle User (table utilisateurs)
│
├── static/                 # Assets statiques
│   └── css/
│       └── style.css       # Feuilles de style
│
├── templates/              # Vues (templates Jinja2)
│   ├── dashboard.html      # Page après connexion
│   ├── login.html          # Formulaire de connexion
│   └── register.html       # Formulaire d'inscription
│
├── .dockerignore           # Fichiers ignorés par Docker
├── .gitignore              # Fichiers ignorés par Git
├── app.py                  # Point d'entrée principal
├── docker-compose.yml      # Configuration multi-conteneurs
├── Dockerfile              # Configuration de l'image Docker
└── requirements.txt        # Dépendances Python
```

POURQUOI ON UTILISE MODELE MVC ?

Séparation des préoccupations : MVC divise l'application en trois parties distinctes :

- Modèle : Gère la logique des données, l'accès à la base de données et les opérations CRUD (Create, Read, Update, Delete).
- Vue : Responsable de l'interface utilisateur, de la présentation des données et de la génération du code HTML.
- Contrôleur : Agit comme intermédiaire entre le Modèle et la Vue. Il reçoit les requêtes de l'utilisateur, interagit avec le Modèle pour obtenir ou modifier les données, et transmet ces données à la Vue pour l'affichage.

Cette séparation améliore l'organisation, la lisibilité et la maintenabilité du code. Il est plus facile de modifier ou de remplacer une partie de l'application sans affecter les autres.

Réutilisation du code : Les Modèles peuvent être réutilisés dans différentes Vues et Contrôleurs, réduisant ainsi la duplication du code.

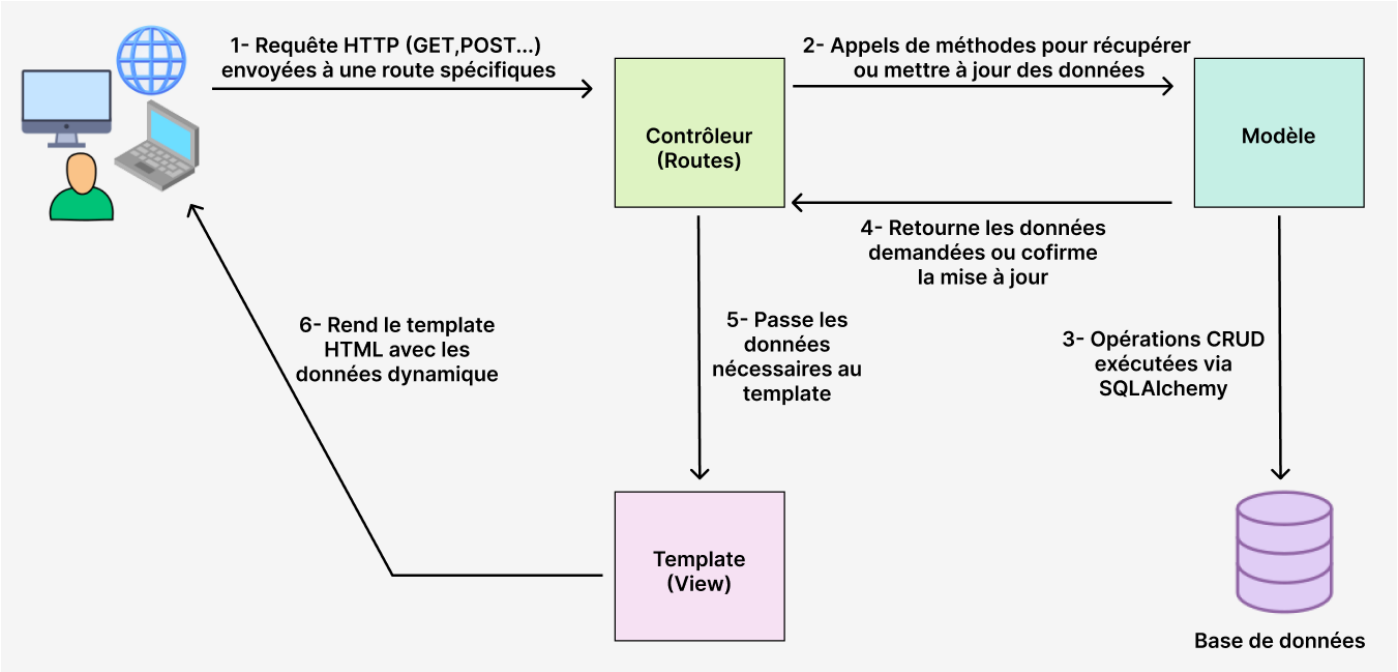
Testabilité : Chaque composant (Modèle, Vue, Contrôleur) peut être testé de manière isolée, ce qui facilite la détection et la correction des erreurs.

Collaboration : MVC permet à différents développeurs de travailler simultanément sur différentes parties de l'application.

COMMENT FONCTIONNE MVC AVEC FLASK ?

Le cycle de vie d'une requête dans une application Flask architecturée selon le modèle MVC débute avec une requête HTTP envoyée par l'utilisateur à une route spécifique.
Le contrôleur, orchestré par les routes définies dans Flask, reçoit cette requête et détermine l'action à exécuter. Pour obtenir ou modifier les données requises, il sollicite le modèle, qui interagit avec la base de données au travers d'opérations CRUD (Create, Read, Update, Delete), souvent facilitées par un ORM tel que SQLAlchemy.
Une fois les données obtenues ou mises à jour, le modèle les renvoie au contrôleur.
Le contrôleur prépare ensuite ces données pour l'affichage et les transmet au template, la vue dans l'architecture MVC.
Le template, généralement géré par un moteur comme Jinja2, utilise ces données pour générer dynamiquement le code HTML, qui est finalement renvoyé au navigateur de l'utilisateur, affichant la page web mise à jour.

Le schéma suivant résume tout le cycle :

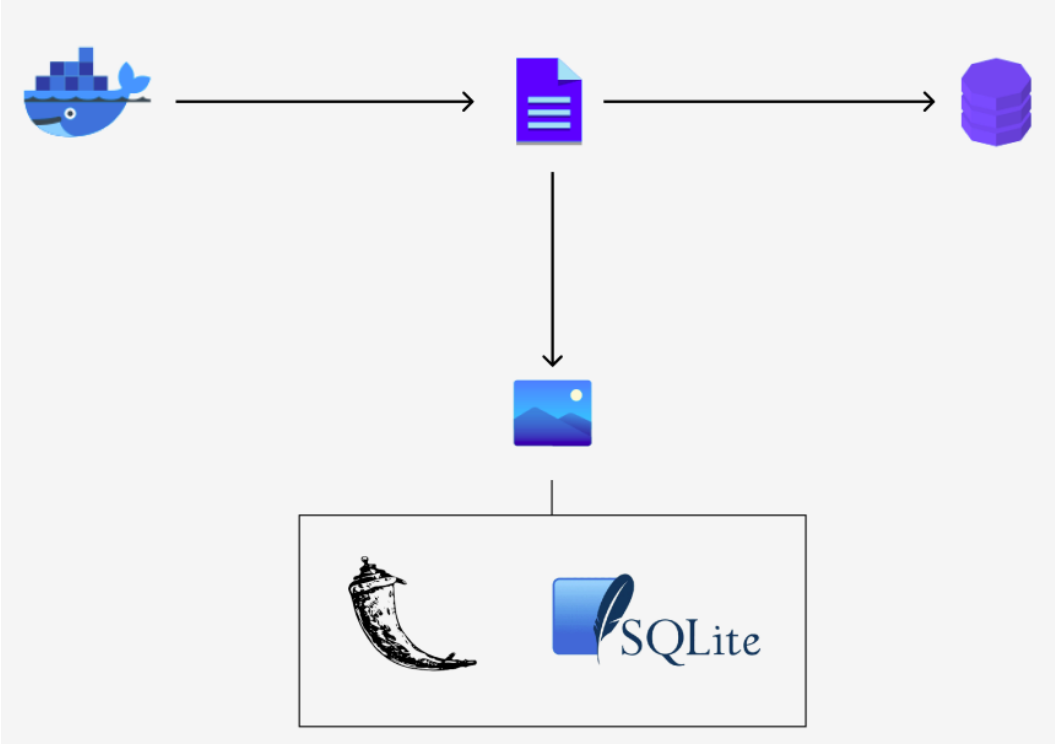


COMMENT DOCKER INTERAGIT AVEC FLASK ?

D'abord Docker est une plateforme permettant de containeriser une application, c'est-à-dire de l'exécuter dans un environnement isolé et portable appelé conteneur.

Parmi les principaux composants qu'on va discuter, il y'a :

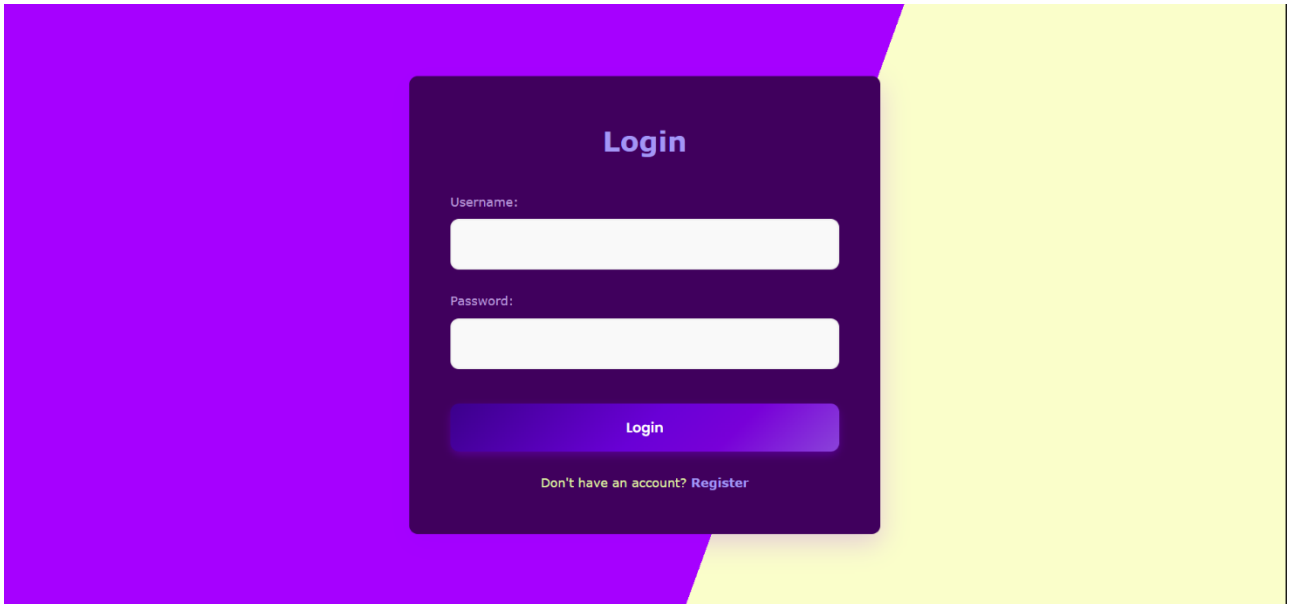
- **Dockerfile** : un fichier qui décrit comment construire une image Docker, il permet de donner des instructions pour installer les dépendances nécessaires, ajouter le code, définir les ports d'écoute et le point d'entrée de l'application.
- **Image Docker** : cette image est construite à partir dockerfile, elle peut être utilisé pout démarrer un conteneur, on peut trouver et construire d'autre image à partir d'une image existante dans docker hub.
- **Docker-compose** : outil permettant de définir et de gérer plusieurs conteneurs Docker via un fichier docker-compose.yml , il est utilisé pour faciliter la gestion de l'environnement.
- **Volumes** : Les volumes Docker sont utilisés pour stocker les données de façon persistante, par exemple pour SQLite qui stocke les fichiers localement le volume assure que les données persistent même si le conteneur est supprimé ou redémarré.

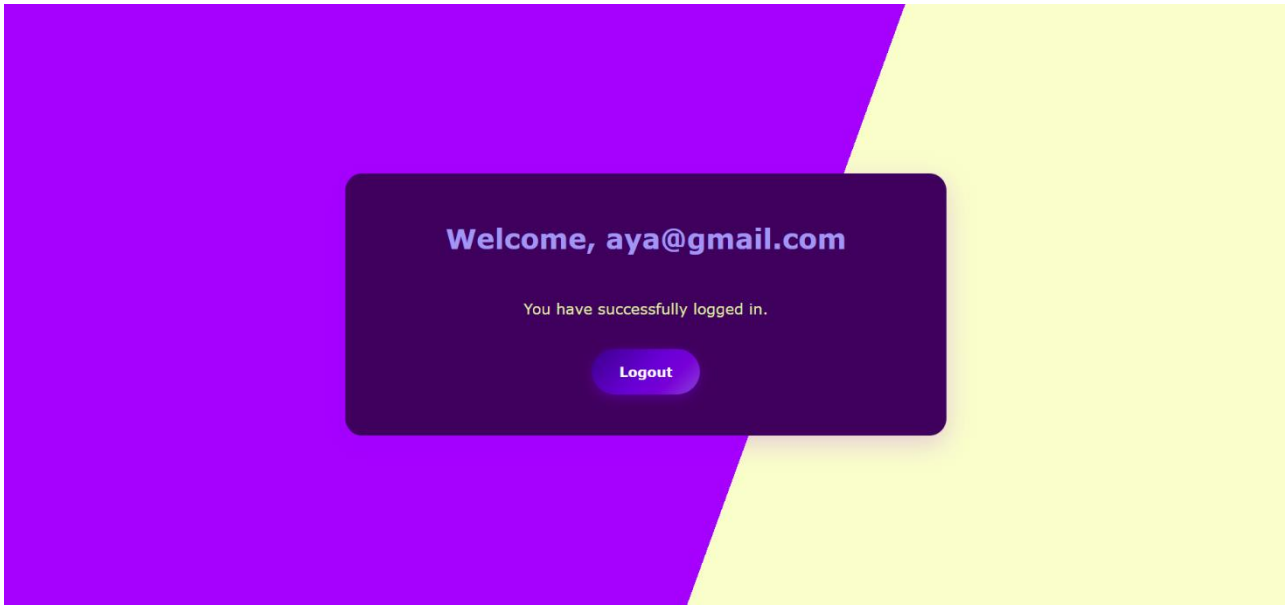


Ce schéma résume les relations entre flask et docker :

1. Docker permet d'exécuter des conteneurs.
2. Le Dockerfile définit comment construire une image contenant Flask et SQLite.
3. L'image est utilisée pour créer un conteneur exécutant l'application Flask.
4. Docker Compose peut être utilisé pour gérer l'exécution et l'interconnexion des conteneurs.
5. Les volumes permettent de stocker les données de SQLite de manière persistante.

U S E R I N T E R F A C E





U T I L I T E D E C H A Q U E D O S S I E R

- **.venv** :

  Ce dossier autrement appelé environnement virtuel est un fichier qu'on fréquente dans développement python. Il sert à crée un espace isolé pour le paquet spécifiée au projet. Comme il contient aussi une copie de l'interpréteur Python, un dossier pour les paquets installés (sans avoir besoin de droits administrateur) par pip et scripts d'activation.

  Pour le crée en utilise la commande : python -m venv .venv
  Il faut aussi l'activer en utilisant : .venv\Scripts\activate

- **__pycache__** :

  Il est crée automatiquement par python lors de l'exécution du scripts. Son rôle est de réduire le temps des chargement des modules puisqu'il évite de recompiler le code à chaque exécution.

- **Controllers** :

  Ce dossier contient la logique métier de l'application, elles gèrent les routes et les interactions entre modèles et templates et répond au requêtes HTTP.

- **Models** :

  Dans ce fichier on peut définir les tables SQL et les relations entre entités (one to many, many to many,…)

- **Template** :

  Gérer l'affichage dynamique des pages web en combinant HTML et données serveur via le moteur de template Jinja2, commes on évite la duplacation du code.

- **Static** :

  Stocker et organiser les fichiers non dynamique comme css, js, img,…

- **Instance** :

  Stocker les configurations sensibles et les fichiers spécifiques comme database, config.py.

EXPLICATION DU CODE

- **app.py :**

Ce fichier est le point d'entrée principal de l'application, il permet de crée l'applications, configuration de bases de données, enregistrer les composants et lance le serveur de développement.

```
from flask import Flask
```

>> Importe la classe Flask depuis le package flask

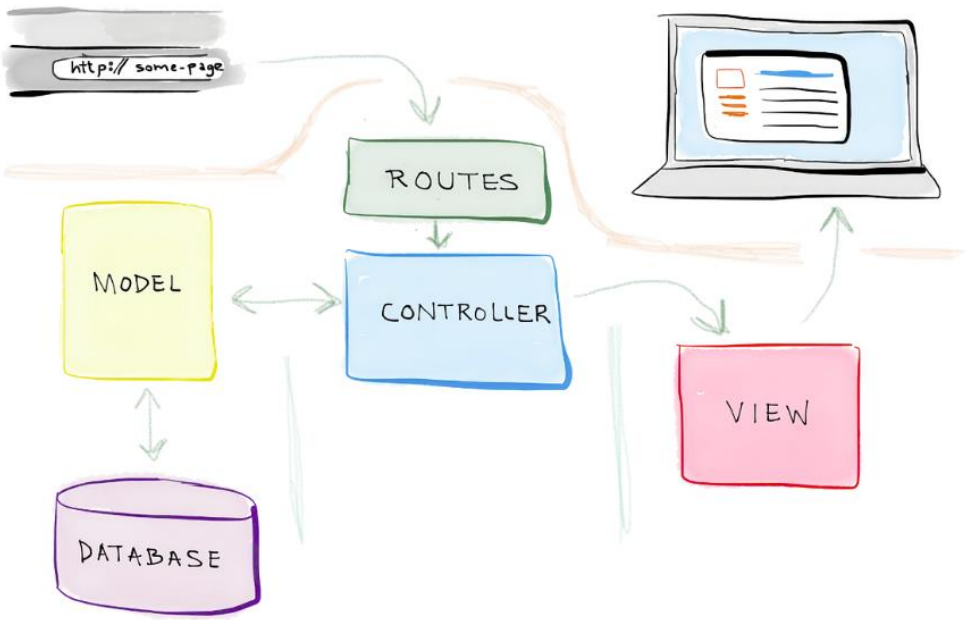
```
from models.database import db
```

>> Importe l'instance de SQLAlchemy (db) depuis un module local

```
from controllers.auth_controller import auth_blueprint
```

>> Importe un Blueprint (groupe de routes) pour l'authentification.
Le Blueprint enregistrent chaque opération effectuée sur l'application. Lorsque Flask génère une URL à partir d'un point de terminaison, il relie la fonction de vue à un Blueprint.
Nous utilisons les Blueprints de Flask car c'est un moyen de décomposer une application en éléments plus petits.

- ✓ Nous pouvons enregistrer un Blueprint sur une application à un préfixe d'URL.
- ✓ Nous pouvons enregistrer plusieurs Blueprints sur une application



```
app = Flask(__name__)
```

>> Crée l'instance principale de l'application Flask et __name__ aide Flask à trouver les ressources (templates, static files).

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'
```

>> Configure le chemin de la base de données SQLite
sqlite:/// = protocole SQLite
users.db = nom du fichier de base de données

```
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

>> Désactive une fonctionnalité de suivi des modifications de SQLAlchemy pour améliorer les performances

```
app.config['SECRET_KEY'] = 'mysecretkey'
```

>> Clé secrète nécessaire pour les sessions utilisateur et les tokens CSRF.

```
db.init_app(app)
```

>> Lie l'instance SQLAlchemy (db) à l'application Flask.

```
app.register_blueprint(auth_blueprint)
```

>> Enregistre le Blueprint d'authentification.

```
with app.app_context():
    db.create_all()
```

>> Crée les tables de la base de données si elles n'existent pas
app.context() est utilisé puisque nous somme au dehors d'une requête HTTP.

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

>> Condition pour lancer le serveur.
debug=True active les messages d'erreur détaillés, le debugger interatif et rechargement automatique du code.

```
    app.run(host='0.0.0.0', port=5050)
```

>>Lorsqu'on utilise docker il faut exposé le port.

- **requirements.txt :**

Ce fichier liste toutes les dépendances Python nécessaires au projet avec leurs versions spécifiques. Afin d'éviter les conflits de versions entre dépendances exemple : SQLAlchemy 2.x est requis pour Flask-SQLAlchemy 3.x

```
alembic==1.15.2                 # Outil de migrations pour
SQLAlchemy
aniso8601==10.0.0               # Parsing des dates ISO 8601
(pour Flask-RESTful)
blinker==1.9.0                  # Système de signaux pour les
notifications
cachelib==0.13.0                # Backend de cache simple
click==8.1.8                    # Framework pour les commandes
CLI
colorama==0.4.6                 # Couleurs dans la console
(Windows)
Flask==3.1.0                    # Framework web principal
Flask-Caching==2.3.1            # Système de caching pour Flask
Flask-Migrate==4.1.0            # Gestion des migrations DB
Flask-RESTful==0.3.10           # Construction d'APIs REST
Flask-SQLAlchemy==3.1.1         # Intégration ORM avec Flask
greenlet==3.1.1                 # Support pour le parallélisme
itsdangerous==2.2.0             # Signatures cryptographiques
Jinja2==3.1.6                   # Moteur de templates
Mako==1.3.9                     # Moteur de templates alternatif
MarkupSafe==3.0.2               # Sécurisation des balises HTML
python-dotenv==1.1.0            # Chargement des variables .env
pytz==2025.2                    # Gestion des fuseaux horaires
six==1.17.0                     # Utilitaires de compatibilité
SQLAlchemy==2.0.39              # ORM principal
typing_extensions==4.13.0       # Support des types
watchdog==6.0.0                 # Rechargement automatique en
dev
Werkzeug==3.1.3                 # Toolkit WSGI (serveur de dev)
```

- **controllers/auth_controller.py :**

```
from flask import Blueprint, render_template, request,
redirect, url_for, session, flash
```

>> Importe les composants Flask nécessaires :

- Blueprint pour créer des routes modulaires
- render_template pour générer des pages HTML
- request pour accéder aux données des requêtes
- redirect et url_for pour la navigation
- session pour gérer les sessions utilisateur
- flash pour afficher des messages temporaires

```
from models.user_model import User
from models.database import db
```

>> Importe le modèle User pour interagir avec la table utilisateurs et l'instance db de SQLAlchemy pour les opérations de base de données

```
from werkzeug.security import generate_password_hash,
check_password_hash
```

>> Importe les fonctions de sécurité pour :

- generate_password_hash : Hasher les mots de passe
- check_password_hash : Vérifier les mots de passe hashés

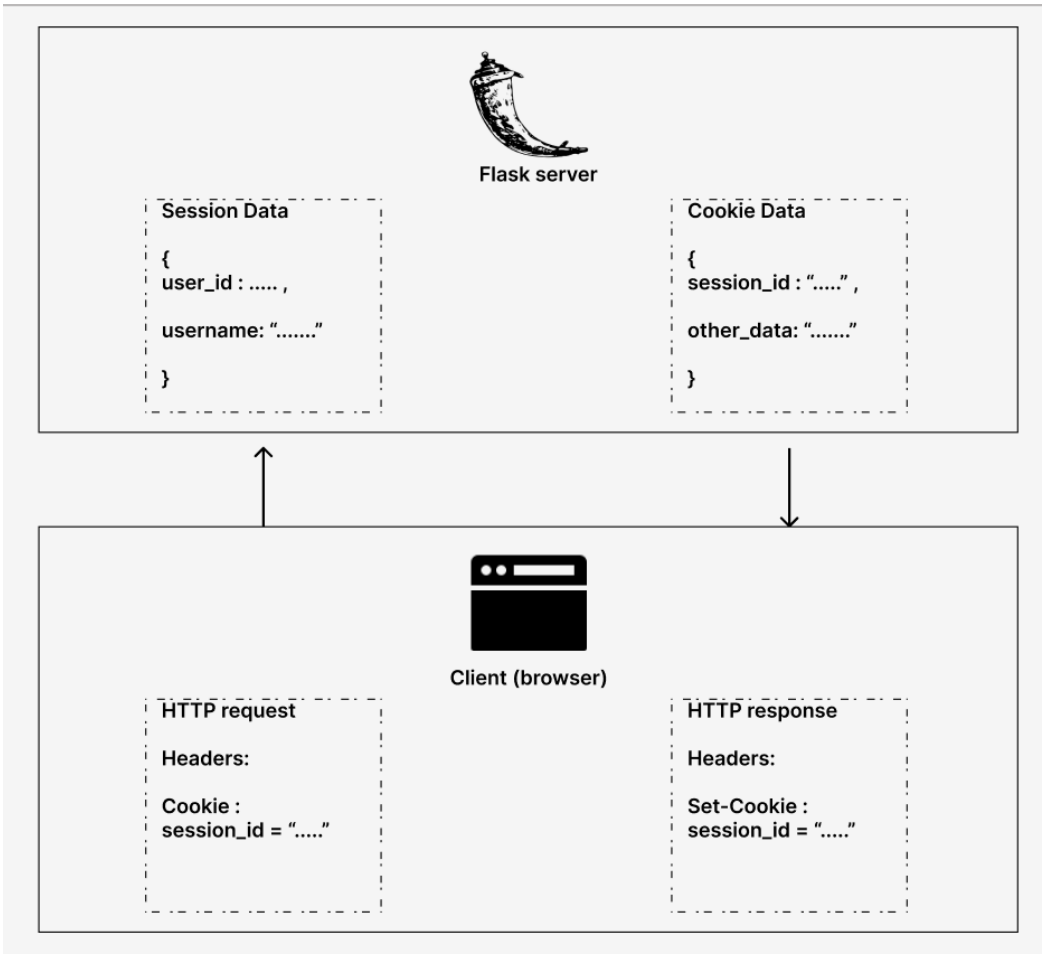
```
auth_blueprint = Blueprint('auth', __name__)
```

>> Crée un Blueprint nommé auth pour organiser les routes liées à l'authentification

```
@auth_blueprint.route('/')
def index():
    if 'username' in session:
        return render_template('dashboard.html',
username=session['username'])
    return redirect(url_for('auth.login'))
```

>> Vérifie si un utilisateur est connecté (présence dans session) si connecté affiche le tableau de bord avec le nom d'utilisateur, sinon redirige vers la page de login.

Mécanisme des sessions Flask :



Quand un utilisateur se connecte (login), Flask crée un objet session côté serveur Ces données sont stockées côté serveur (en mémoire, dans une DB Redis, ou un système de fichiers).
Flask génère un cookie sécurisé contenant uniquement un ID de session (pas les données sensibles) stocké dans le navigateur.

À chaque nouvelle requête, le navigateur envoie automatiquement le cookie dans les headers, exemple :
GET /dashboard HTTP/1.1
Cookie: session_id= value alphanumeric

Flask utilise l'ID du cookie pour retrouver les données associées.
Si la session est modifiée, Flask met à jour le cookie

```
@auth_blueprint.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
```

>> Accepte les méthodes GET (afficher le formulaire) et POST (traiter l'inscription) et récupère les données du formulaire.

```
        existing_user =
User.query.filter_by(username=username).first()
        if existing_user:
            flash('Username already exists!')
            return redirect(url_for('auth.register'))
```

>>Vérifie si l'utilisateur existe déjà si c'est le cas on affiche un message d'erreur.

```
        hashed_password = generate_password_hash(password)
        new_user = User(username=username,
password=hashed_password)
        db.session.add(new_user)
        db.session.commit()
```

>> Hash le mot de passe, crée un nouvel utilisateur et l'ajoute à la base de données

```
        flash('Account created successfully! Please login.')
        return redirect(url_for('auth.login'))
```

>> Confirme la création du compte et redirige vers la page de login

```
    return render_template('register.html')
```

>> Affiche le formulaire d'inscription pour les requêtes GET

```
@auth_blueprint.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
```

>> Accepte GET (formulaire) et POST (traitement) et récupère les identifiants

```
        user = User.query.filter_by(username=username).first()

        if user and check_password_hash(user.password,
password):
            session['username'] = username
            return redirect(url_for('auth.index'))
```

>> Cherche l'utilisateur en base si trouvé et mot de passe valide il va stocker le nom d'utilisateur en session et redirige vers la page d'accueil

```
        else:
            flash('Invalid username or password')
    return render_template('login.html')
```

>> Affiche un message d'erreur si échec sinon affiche le formulaire de connexion pour les requêtes GET

```
@auth_blueprint.route('/logout')
def logout():
    session.pop('username', None)
    return redirect(url_for('auth.login'))
```

>> Supprime le nom d'utilisateur de la session et redirige vers la page de login

---

- **models/database.py :**

Ce fichier initialiser et exporter l'instance SQLAlchemy qui servira de pont entre Flask et votre base de données.

```
from flask_sqlalchemy import SQLAlchemy
```

>> Importe la classe SQLAlchemy depuis l'extension Flask-SQLAlchemy qui va servir à intègre SQLAlchemy (ORM puissant) avec Flask et simplifier la gestion des sessions de base de données.

- **models/user_model.py :**

Représente une table dans votre base de données et permet d'interagir avec les utilisateurs.

```
from models.database import db

class User(db.Model):
```

>> Importe l'instance db créée dans database.py
Pour la classe User qui hérite de db.Model :
Transforme la classe en modèle SQLAlchemy
Permet la création automatique de la table en base de données

```
    id = db.Column(db.Integer, primary_key=True)
```

>> Défini une colonne tel que :
id : Nom de la colonne
db.Integer : Type de données (entier)
primary_key=True : Clé primaire auto-incrémentée

```
    username = db.Column(db.String(80), unique=True,
nullable=False)
```

>> colonne username:
String(80) : Chaîne de caractères (max 80 caractères)
unique=True : Valeur unique
nullable=False : Champ obligatoire

```
    password = db.Column(db.String(200), nullable=False)
```

>> Colonne password :
String(200) : Taille suffisante pour stocker un hash
nullable=False : Champ obligatoire

La requête sql équivalent à cette partie est :

```
CREATE TABLE user (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username VARCHAR(80) UNIQUE NOT NULL,
    password VARCHAR(200) NOT NULL
);
```

- **.gitignore :**

  Spécifie les fichiers/dossiers que Git doit ignorer. Afin d'évite de commettre accidentellement des données sensibles, réduire la taille du dépôt Git et empêche les conflits de cache entre développeurs

---

- **.dockerignore:**

  Spécifie ce que Docker doit exclure lors de la construction de l'image. Ceci est pour réduire la taille de l'image Docker et accélère la construction en ignorant les fichiers inutiles aussi pour améliorer la sécurité en excluant les fichiers sensibles