



Institut National des Postes et Télécommunications

**Module : Communication P2P en temps réel sur le web (INE3)**

# Rapport de TP1 : Les Servlets HTTP

*Implémentation via Python Flask : Une approche moderne des concepts JEE*

Réalisé par :  
Aymane ASKRI

Encadré par :  
Asmae ELHAMZAOUI

Année Universitaire : 2025-2026

# Table des matières

<b>Introduction</b>	<b>2</b>
<b>1 Réalisation des Extensions</b>	<b>3</b>
1.1 Intégration d'une Base de Données SQLite dans l'Application Flask . . . . .	3
1.2 Mise en Place d'un Système de Rôles et de Restrictions d'Accès . . . . .	7
1.3 Sécurité et Blacklist (Filtres) . . . . .	7
1.4 Utiliser JSP (Templates) . . . . .	9
1.5 Gestion des erreurs (404, 500) . . . . .	10
<b>2 Synthèse des Concepts : Architecture Web et Protocole HTTP</b>	<b>11</b>
2.1 Le Modèle Client-Serveur et le Cycle de Vie HTTP . . . . .	11
2.2 Le Contrôleur : Routage et Traitement (Servlets vs Routes) . . . . .	11
2.3 La Vue : Génération de Contenu Dynamique (JSP vs Jinja2) . . . . .	12
2.4 Le Modèle : Persistance des Données (JDBC vs SQLite) . . . . .	12
2.5 Middleware et Aspect-Oriented Programming (Filtres vs Décorateurs) . . .	12
2.6 Gestion des Erreurs HTTP . . . . .	12
<b>Conclusion</b>	<b>14</b>

# Introduction

L'objectif initial de ce TP est de manipuler le protocole HTTP en utilisant la plateforme JEE et sa composante Servlet. Cependant, afin de se concentrer sur la maîtrise des concepts architecturaux (MVC, cycle de vie HTTP, Statelessness) plutôt que sur la complexité de configuration de Java, ce travail pratique a été réalisé en utilisant le langage **Python** et le micro-framework **Flask**.

Cette approche permet de reproduire fidèlement l'architecture demandée tout en offrant un code plus concis et lisible. Ce rapport détaille les étapes de réalisation et les extensions demandées, en mettant en parallèle les concepts Java EE et leur implémentation Python.

# Chapitre 1

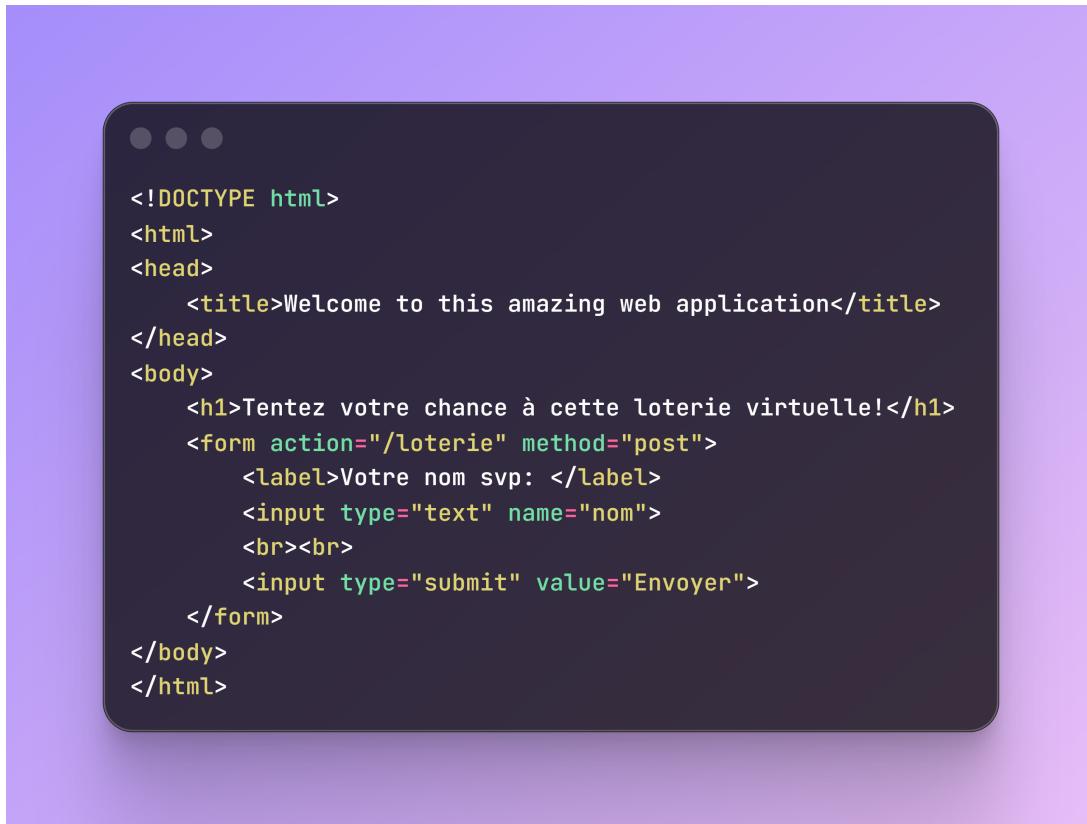
## Réalisation des Extensions

### 1.1 Intégration d'une Base de Données SQLite dans l'Application Flask

Pour cette extension, nous avons remplacé la technologie JDBC (Java Database Connectivity) par le module `sqlite3` natif de Python. SQLite a été choisi pour sa légèreté, évitant l'installation d'un serveur lourd comme MySQL pour un environnement de TP, tout en respectant la logique SQL standard.

#### Le Formulaire HTML

Voici le code du formulaire permettant la saisie de l'utilisateur :



```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome to this amazing web application</title>
</head>
<body>
    <h1>Tentez votre chance à cette loterie virtuelle!</h1>
    <form action="/loterie" method="post">
        <label>Votre nom svp: </label>
        <input type="text" name="nom">
        <br><br>
        <input type="submit" value="Envoyer">
    </form>
</body>
</html>
```

FIGURE 1.1 – Code du formulaire HTML

## Le Contrôleur (App.py)

Ce fichier remplace la classe `GreetingServlet` ainsi que la configuration `web.xml`. Il gère le routage et la logique métier.

```
from flask import Flask, request, render_template
from db import init_db
import sqlite3
import random

app = Flask(__name__)

init_db() # On crée la table au lancement

# Route pour la page d'accueil (affiche le formulaire)
# Équivalent au fichier HTML statique servi par Tomcat
@app.route('/')
def home():
    return render_template('index.html')

# Route qui traite le formulaire (Équivalent à
# GreetingServlet)
@app.route('/Loterie', methods=['GET', 'POST'])
def loterie():
    nom_prenom = "Anonymous"

    # Équivalent de request.getParameter("nom")
    if request.method == 'POST':
        nom_recu = request.form.get('nom')
    else:
        nom_recu = request.args.get('nom') # Pour gérer le
    GET aussi

    if nom_recu:
        nom_prenom = nom_recu.upper()

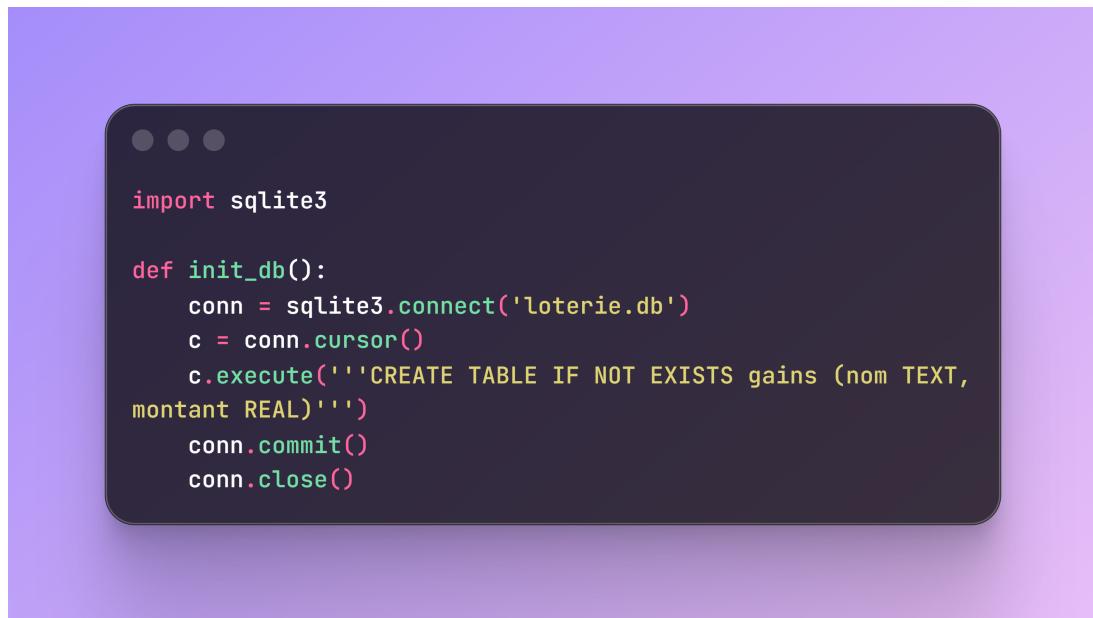
    # Logique métier (calcul du gain)
    gain = round(random.random() * 10, 2)
    conn = sqlite3.connect('loterie.db')
    c = conn.cursor()
    c.execute("INSERT INTO gains (nom, montant) VALUES (?, ?)", (nom_prenom, gain))
    conn.commit()
    conn.close()

    # Construction de la réponse (Équivalent de out.println)

    # Note: En Python moderne, on utilise des f-strings pour
    insérer les variables
    html_response = """
<html>
<body>
    <h1>Greetings {nom_prenom}!</h1>
    <p>Vous avez gagné: <b>{gain}</b> millions de
    dollars!</p>
    <a href="/">Rejouer</a>
</body>
</html>
"""
    """
```

## La Couche de Données (DB.py)

Python intègre SQLite par défaut. Ce module gère la persistance des données (gains de loterie).



```
import sqlite3

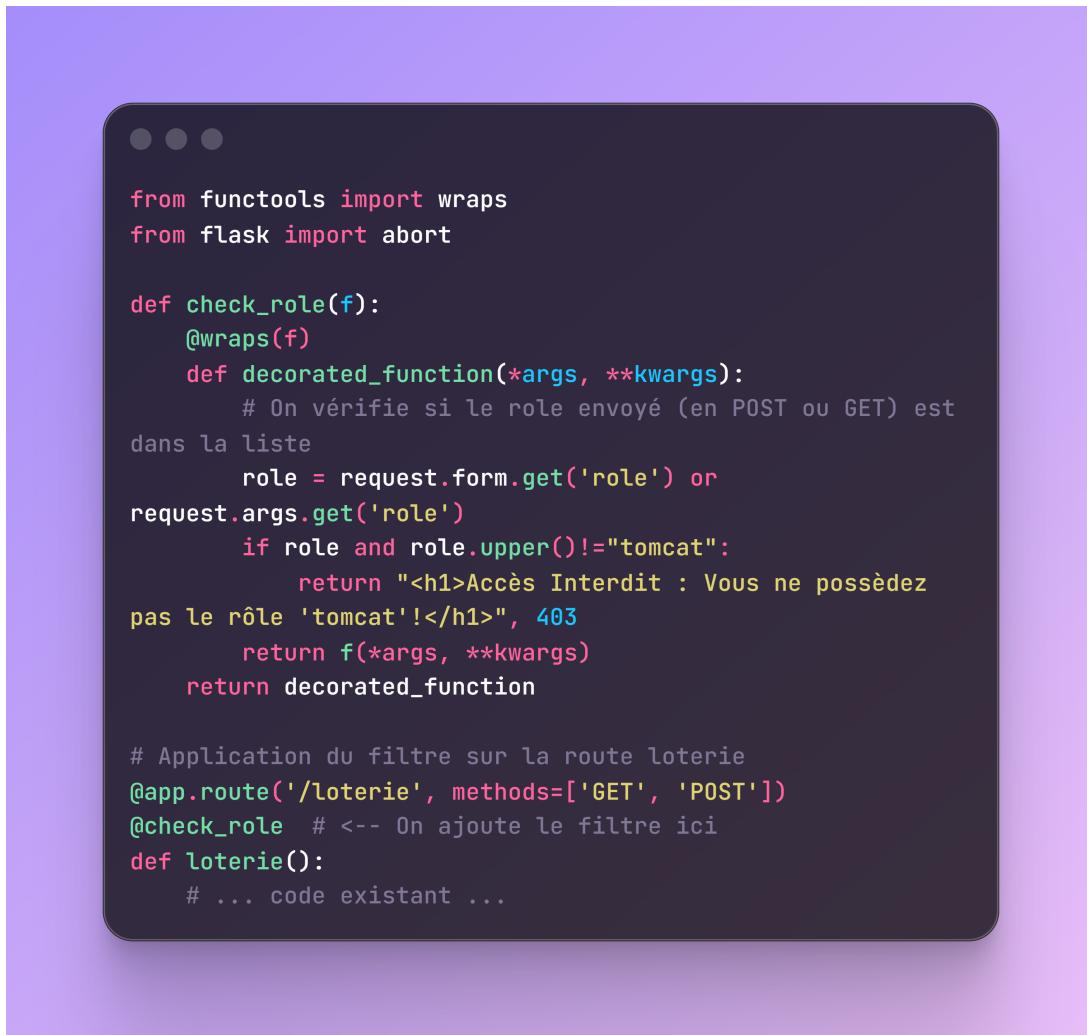
def init_db():
    conn = sqlite3.connect('loterie.db')
    c = conn.cursor()
    c.execute('''CREATE TABLE IF NOT EXISTS gains (nom TEXT,
montant REAL)''')
    conn.commit()
    conn.close()
```

FIGURE 1.3 – Gestion de la base de données (db.py)

## 1.2 Mise en Place dun Système de Rôles et de Restrictions dAccès

Pour restreindre l'accès, nous utilisons le concept de **Décorateur** en Python. C'est l'équivalent fonctionnel des mécanismes de sécurité basés sur les rôles dans le conteneur de servlets.

Voici comment créer un décorateur pour bloquer les utilisateurs ne possédant pas le rôle 'tomcat' :



```

from functools import wraps
from flask import abort

def check_role(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        # On vérifie si le rôle envoyé (en POST ou GET) est
        # dans la liste
        role = request.form.get('role') or
        request.args.get('role')
        if role and role.upper()!="tomcat":
            return "<h1>Accès Interdit : Vous ne possédez
pas le rôle 'tomcat'!</h1>", 403
        return f(*args, **kwargs)
    return decorated_function

# Application du filtre sur la route loterie
@app.route('/loterie', methods=['GET', 'POST'])
@check_role # <-- On ajoute le filtre ici
def loterie():
    # ... code existant ...

```

FIGURE 1.4 – Décorateur pour la vérification du rôle utilisateur

## 1.3 Sécurité et Blacklist (Filtres)

En Java EE, on utilise des **Filters** pour intercepter les requêtes. En Python Flask, on utilise également des **Décorateurs** pour appliquer une logique transversale (AOP - Aspect Oriented Programming).

Voici l'implémentation d'un décorateur pour bloquer une liste noire (Blacklist) d'utilisateurs :



```
from functools import wraps
from flask import abort

BLACKLIST = ["VOLDEMORT", "SAURON"] # Liste noire

def check_blacklist(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        # On vérifie si le nom envoyé (en POST ou GET) est
        # dans la liste
        nom = request.form.get('nom') or
        request.args.get('nom')
        if nom and nom.upper() in BLACKLIST:
            return "<h1>Accès Interdit : Vous êtes sur la
liste noire !</h1>", 403
        return f(*args, **kwargs)
    return decorated_function

# Application du filtre sur la route loterie
@app.route('/loterie', methods=['GET', 'POST'])
@check_blacklist # <-- On ajoute le filtre ici
def loterie():
    # ... code existant ...
```

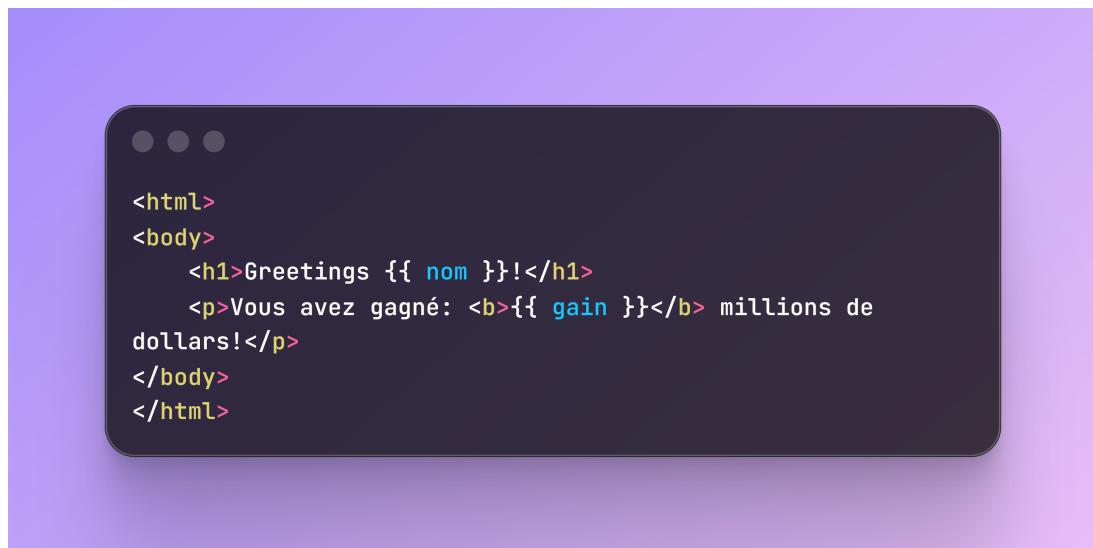
FIGURE 1.5 – Implémentation du filtre Blacklist via un décorateur

## 1.4 Utiliser JSP (Templates)

L'instruction demande de ne plus écrire le code HTML à l'intérieur du code métier (Java/Python). Flask utilise le moteur de template **Jinja2**, qui est l'équivalent direct de la technologie **JSP** (JavaServer Pages).

### Le Template (Vue)

Nous créons un fichier `templates/resultat.html` qui contient la structure de la page avec des placeholders pour les variables dynamiques.

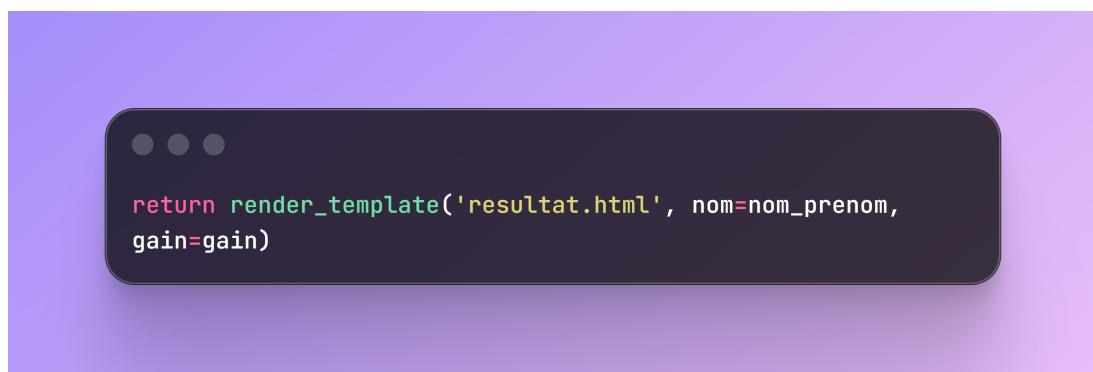


```
<html>
<body>
    <h1>Greetings {{ nom }}!</h1>
    <p>Vous avez gagné: <b>{{ gain }}</b> millions de
dollars!</p>
</body>
</html>
```

FIGURE 1.6 – Fichier Template Jinja2 (resultat.html)

### Modification du Contrôleur

Nous modifions la fonction Python pour utiliser la méthode `render_template`, séparant ainsi la vue du contrôleur.



```
return render_template('resultat.html', nom=nom_prenom,
gain=gain)
```

FIGURE 1.7 – Appel du moteur de template dans le contrôleur

## 1.5 Gestion des erreurs (404, 500)

Cette partie correspond à la configuration de la balise <error-page> dans le fichier web.xml de Java EE. En Flask, nous utilisons des gestionnaires d'erreurs (`errorhandler`) spécifiques.

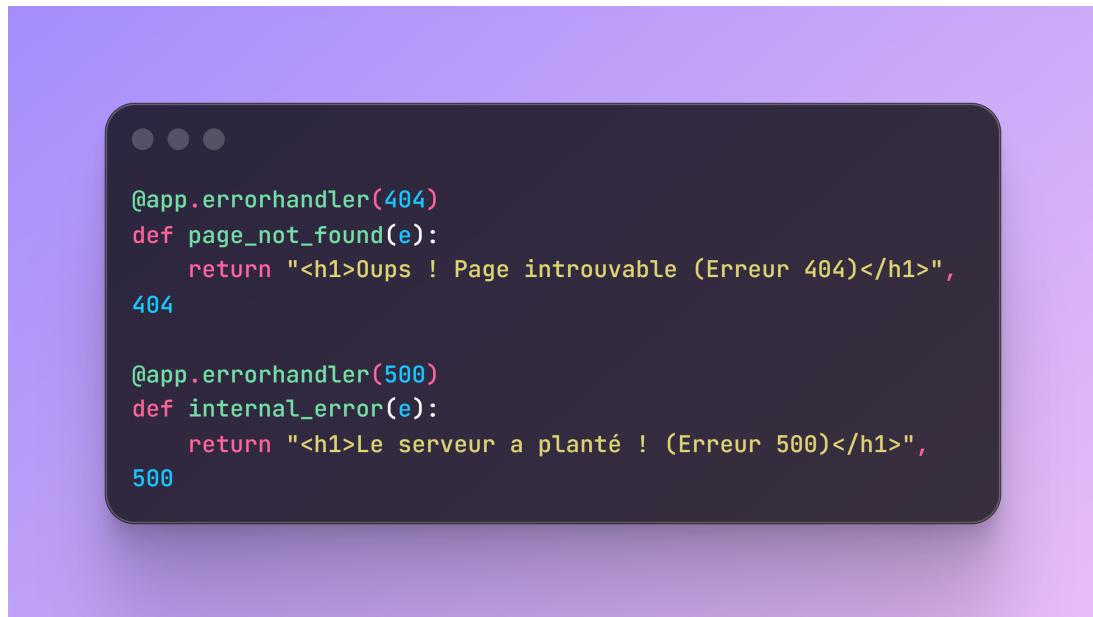


FIGURE 1.8 – Gestion personnalisée des codes d’erreur HTTP

# Chapitre 2

## Synthèse des Concepts : Architecture Web et Protocole HTTP

Ce Travail Pratique (TP) a permis d'explorer les fondements du développement web dynamique et la mise en œuvre du protocole HTTP au sein d'une architecture client-serveur. Bien que l'implémentation ait été réalisée avec le framework Python Flask, les concepts manipulés restent universels et transposables à l'architecture Java EE (Servlets/JSP).

Voici les piliers conceptuels maîtrisés lors de ce TP :

### 2.1 Le Modèle Client-Serveur et le Cycle de Vie HTTP

Le cœur de ce TP repose sur la compréhension du dialogue entre le client (navigateur) et le serveur (notre application Python).

- **Statelessness (Sans état)** : Nous avons constaté que le protocole HTTP est sans état. Chaque requête est indépendante. Pour conserver une information (comme le nom de l'utilisateur) entre deux pages, nous avons dû transmettre explicitement les données via des formulaires ou les stocker en base de données.
- **Verbes HTTP (GET vs POST)** :
  - **GET** a été utilisé pour demander des ressources (afficher le formulaire). Les paramètres sont visibles dans l'URL, ce qui est limité en sécurité.
  - **POST** a été utilisé pour soumettre des données (le nom de l'utilisateur) au serveur pour traitement. Les données sont encapsulées dans le corps de la requête, offrant plus de sécurité et de capacité.

### 2.2 Le Contrôleur : Routage et Traitement (Servlets vs Routes)

L'équivalent du composant `HttpServlet` en Java EE a été matérialisé par les fonctions de vue ("View Functions") en Flask.

- **Mapping d'URL** : Nous avons associé des URLs spécifiques (ex : `/loterie`) à des fonctions Python via le mécanisme de routage (décorateur `@app.route`). Cela correspond à la configuration du fichier `web.xml` ou aux annotations `@WebServlet` en Java.

- **Logique Métier** : Le code serveur ne se contente pas de servir des fichiers statiques ; il exécute un algorithme (génération d'un nombre aléatoire, mise en majuscule) avant de construire la réponse. C'est le cœur dynamique de l'application.

## 2.3 La Vue : Génération de Contenu Dynamique (JSP vs Jinja2)

Le TP a mis en évidence la nécessité de séparer la logique de traitement (code Python) de la présentation (code HTML).

- **Templating** : Au lieu de concaténer des chaînes de caractères (pratique lourde et sujette aux erreurs), nous avons utilisé un moteur de template (Jinja2, équivalent aux JSP).
- **Injection de données** : Le serveur "injecte" des variables (le gain, le nom) dans des "trous" prévus à cet effet dans le fichier HTML (`{{ variable }}`). Cela permet de générer une page unique pour chaque utilisateur tout en gardant une structure HTML propre.

## 2.4 Le Modèle : Persistance des Données (JDBC vs SQLite)

L'extension du TP nous a fait passer d'une application "volatile" (où les données disparaissent après le traitement) à une application "persistante".

- **Interaction Base de Données** : L'utilisation de SQL via la bibliothèque `sqlite3` (équivalent à JDBC) a permis de stocker l'historique des gains.
- **ACID** : Bien que simple, cette implémentation respecte les principes transactionnels (ouverture de connexion, exécution de requête, commit, fermeture).

## 2.5 Middleware et Aspect-Oriented Programming (Filtres vs Décorateurs)

Pour gérer la sécurité (Blacklist) et les restrictions d'accès, nous avons implémenté un mécanisme d'interception.

- En Java EE, cela se fait via des **Filtres** (`javax.servlet.Filter`) qui interceptent la requête avant qu'elle n'arrive à la Servlet.
- En Python, nous avons utilisé le pattern **Décorateur**. Cela permet d'appliquer une logique transversale (vérification d'un nom interdit) sans polluer le code de la fonction principale. C'est un concept clé pour la maintenance et la sécurité du code.

## 2.6 Gestion des Erreurs HTTP

Nous avons appris à ne pas laisser le serveur planter silencieusement ou afficher des erreurs techniques à l'utilisateur.

L'implémentation de gestionnaires personnalisés pour les codes 404 (Not Found) et 500 (Internal Server Error) permet de contrôler l'expérience utilisateur même en cas de problème, un standard essentiel dans les applications web professionnelles.

# Conclusion

En réalisant ce TP avec Python Flask, nous avons reproduit fidèlement l'architecture **MVC (Modèle-Vue-Contrôleur)** préconisée par Java EE.

- **Modèle** : Base de données SQLite.
- **Vue** : Templates HTML/Jinja2.
- **Contrôleur** : Fonctions Python et Routage Flask.

Cette approche a permis de démystifier la complexité des Servlets tout en se concentrant sur la logique des protocoles et l'architecture logicielle.