Projet FitSize : performances et optimisations

Florian CHAPPAZ, Valentin DE OLIVEIRA, Sami IFAKIREN, Clément NGUYEN

I. Introduction

Le projet FitSize a pour objectif d'aider les consommateurs à trouver des vêtements à leur taille parmi plusieurs collections de différents créateurs. Pour cela, les clients et les créateurs doivent passer par un étape de scan de leurs vêtements, qui permettra au serveur de calculer une échelle et des coordonnées de points d'ancrage pour le vêtement en question, permettant ainsi un calcul de dimension. Nous n'allons pour le moment pas détailler le fonctionnement exact de ce processus, mais il faut simplement savoir que lorsque le client effectue la requête permettant ce calcul, un processus est lancé avec un script utilisant PyTorch et faisant appel à des modèles pré-entrainés que nous vons stocké. Cette phase de prédiction prend du temps (quelques secondes), et lorsque le serveur devra gérer plusieurs clients, cela peut devenir problématique. Nous allons donc analyser notre situation actuelle, puis tenter de trouver des solutions à ces problèmes.

II. L'existant : notre serveur Django de base

A la base, notre serveur contient un unique endpoint permettant de calculer les points d'ancrage d'un vêtement. Cet endpoint se trouve dans le fichier *keypoints/views.py*. Il reçoit un type de vêtement ainsi que les données d'une image. Comme nous pouvons le voir sur la Figure 1, le script de prédiction des points d'ancrage est appelé via un *subprocess*.

```
body_json = json.loads(request.body)
clothing = body_json['clothing']
img_data = body_json['image'] # base64 raw data (without prefix data:image/...)
wd = os.getcwd() + '/keypoints/code'

# Generate unique uuid
img_id = uuid.uuid4()

with open(wd + '/tmp'+str(img_id)+'.jpg', 'wb') as f:
    f.write(base64.b64decode(img_data)) # Valid only for jpg data
    f.close()

ai_exec = subprocess.Popen(['python', 'run_no_det.py',
                            '--clothing', clothing,
                            '--source', wd + '/tmp'+str(img_id)+'.jpg'],
                           stdout=subprocess.PIPE,
                           stderr=subprocess.STDOUT,
                           cwd=wd)
res = ai_exec.communicate()[0]
os.remove(os.getcwd() + '/keypoints/code/tmp'+str(img_id)+'.jpg')

# Read results and parse it to json to send back to client
```

Figure1 : Extrait de l'endpoint execScript

Le *subprocess* va mettre un certain temps à s'exécuter. C'est pourquoi la méthode *communicate()* va être appelée : elle permet de lire la sortie de l'exécution du *subprocess*. C'est pourquoi elle va attendre la fin du processus et va donc être bloquante.

C'est problématique car Django exécute les requêtes une par une et à la suite. Si une requête est dans une fonction bloquante, elle va alors bloquer toutes les requêtes derrière elle.

Nous pouvons nous en convaincre en utilisant l'outil Apache JMeter (https://jmeter.apache.org/). Cet outil va nous permettre de lancer plusieurs requêtes simultanées (via plusieurs threads) et donc de simuler plusieurs utilisateurs tentant d'accéder au serveur. Cet outil comporte d'autres fonctionnalités plus avancées dont nous pourrions avoir besoin plus tard.