MGL7760 - PROJET 1

Mise en place d'un environnement de développement local utilisant des conteneurs Docker

DOCUMENT DE SYNTHÈSE

Yao Kevin Amouzou: AMOY02319402
Khadija Dia : DIAK91360008

Table des matières

Introduction

La gestion d'une bibliothèque peut être une charge lourde et laborieuse, mais avec l'avènement des technologies web, de nombreuses tâches peuvent désormais être automatisées pour faciliter la gestion quotidienne de ces institutions. C'est dans ce contexte que, dans le cadre du cours de Qualité et productivité des outils logiciels, notre équipe a développé une application web de gestion de bibliothèque personnelle, destinée à faciliter les tâches des utilisateurs.

Cette application web doit permettre à l'utilisateur d'afficher la liste des livres, de les filtrer par catégorie, d'afficher un livre en particulier et ses détails et enfin d'effectuer une recherche par titre ou par auteur.

Nous avons développé l'application en utilisant le langage Python 3, le micro-framework Flask, le toolkit/ORM SQLAlchemy, Visual Studio Code, Github.

L'objectif principal était de se familiariser avec des outils de développement moderne utilisant des conteneurs, donc nous avons d'abord mis en place l'environnement de développement local avec Docker. Il s'agissait d'empaqueter notre application et ses dépendances dans des conteneurs pour une distribution et une exécution faciles. L'outil Docker Compose nous a permis de définir et exécuter cette application multi-conteneurs. Il permet de décrire les conteneurs nécessaires pour faire fonctionner l'application ainsi que les services associés tels que la base de données, dans un seul fichier de configuration. Il fallut ensuite utiliser un simple commande pour déployer tous les conteneurs nécessaires pour faire fonctionner l'application.

Nous avons défini les classes Livre, Auteur, Editeur et Catégorie dans un fichier Python séparé. En utilisant SQLAlchemy, nous avons créé les modèles correspondants à ces classes qui seront utilisés pour mapper les données en base de données.

Ensuite, nous avons utilisé Flask pour construire l'interface utilisateur de notre application en utilisant des templates HTML et en définissant les routes pour les pages de l'application. Nous pouvons également implémenter les fonctionnalités pour ajouter, supprimer, rechercher et mettre à jour des livres, des auteurs, des éditeurs et des catégories en utilisant les classes et les modèles que nous avons définis.

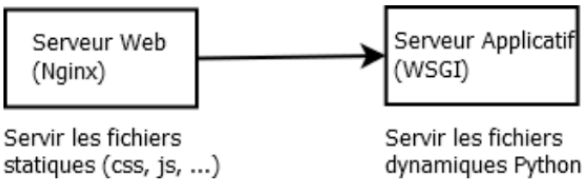
Tout au long du développement nous avons utilisé Github pour le développement collaboratif. Cela a permis le contrôle de version de notre code et le suivi des tâches, des bugs et des demandes de fonctionnalités en utilisant GitHub Pull Requests and Issues.

Dans cette synthèse, nous allons présenter notre démarche pour la création de l'environnement de développement, les choix techniques que nous avons faits et le fonctionnement du script shell gerer-conteneurs.sh. Dans un second temps, nous allons expliquer l'importance des enjeux reliés à la qualité et la productivité engendré par l'utilisation

d'outils logiciels modernes. Enfin, nous allons décrire l’architecture logicielle de notre application.

Création de l'environnement de développement

Etape 1 : 1 serveur web + 1 serveur applicatif



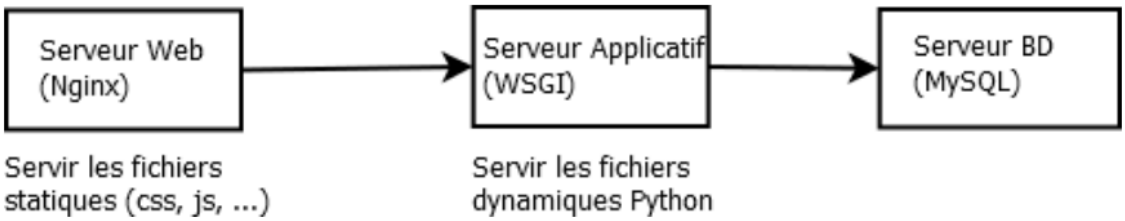
Créer un fichier de configuration Docker Compose : nous avons créé un fichier de configuration pour Docker Compose, qui définit les différents services et les paramètres nécessaires pour faire fonctionner l'application.

```
version: '3'
services:
    wsgi:
        build: ./
        restart: always
        environment:
            - FLASK_APP=app
            - FLASK_DEBUG=1
        deploy:
            mode: replicated
            replicas: 1
        links:
            - db
            - cache
        expose:
            - '5001'
        ports:
            - '5001:5001'
        volumes:
            - './web/project:/app'
        command: gunicorn -w 1 -b 0.0.0.0:5001 app:app
        # command: flask run --host=0.0.0.0
        depends_on:
            - db
server:
        build: ./nginx
        build: nginx
        restart: always
        ports:
            - '80:80'
        volumes:
            - './web:/usr/share/nginx/html'
            - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
        depends_on:
            - wsgi
```

On a définit les services "wsgi" et "nginx", qui seront exécutés dans des conteneurs séparés. Le service "wsgi" est construit à partir du répertoire courant et exécute l'application Flask avec Gunicorn. Le service "server" utilise une image personnalisée de Nginx et relie les services "wsgi" pour servir le contenu web.

Une fois que cela est fait, on exécute la commande *docker-compose up* pour créer et démarrer les conteneurs en utilisant la configuration de docker-compose.yaml.

Etape 2 :



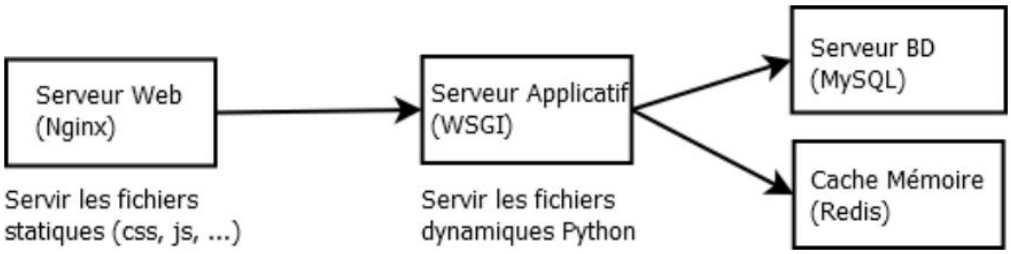
On rajoute le serveur de base de donnée MySQL :

```
db:
    image: mysql:5.7
    restart: always
    container_name: mysql
    ports:
        - '3307:3306'
    expose:
        - "3307"
    environment:
        - MYSQL_USER=user
        - MYSQL_PASSWORD=user123
        - MYSQL_DATABASE=flask_api
        - MYSQL_ROOT_PASSWORD=12345678
    volumes:
        - ./db:/docker-entrypoint-initdb.d/:ro
```

Docker-compose.yaml

Le service "db" utilise l'image Docker de mysql version 5.7 et mappe le port 3307 de la machine hôte sur le port 3306 du conteneur. Il utilise également des variables d'environnement pour définir le nom d'utilisateur, le mot de passe, la base de données et le mot de passe root pour l'image mysql. Il monte également un volume pour initialiser la base de données MySQL.

Etape 3 :



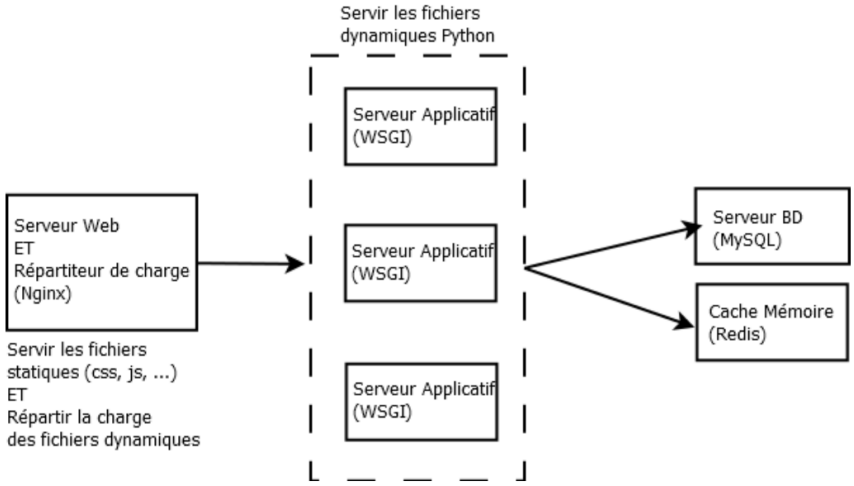
Dans cette étape on rajoute le conteneur du serveur Cache Redis.

```
cache:
        image: redis
        restart: always
        container_name: flask_cache
```

docker-compose.yaml

Il s'agit d'un service Docker pour le cache Redis utilisant l'image Redis officielle. Le paramètre image spécifie l'image à utiliser pour ce service. Le service est automatiquement redémarré en cas d'échec.

Etape 4 :



Pour terminer on rajoute les deux autres serveurs applicatifs :

```
wsgi1:
        build: ./
        restart: always
        environment:
            - FLASK_APP=app
            - FLASK_DEBUG=1
        deploy:
            mode: replicated
            replicas: 1
        links:
            - db
            - cache
        expose:
            - '5002'
        ports:
            - '5002:5002'
        volumes:
            - './web/project:/app'
        command: gunicorn -w 1 -b 0.0.0.0:5002 app:app
        # command: flask run --host=0.0.0.0
        depends_on:
            - db
   wsgi2:
        build: ./
        restart: always
        environment:
            - FLASK_APP=app
            - FLASK_DEBUG=1
        deploy:
            mode: replicated
            replicas: 1
        links:
            - db
            - cache
        expose:
            - '5003'
        ports:
            - '5003:5003'
        volumes:
            - './web/project:/app'
        command: gunicorn -w 1 -b 0.0.0.0:5003 app:app
        # command: flask run --host=0.0.0.0
        depends_on:
            - db
```

Une fois que les conteneurs sont démarrés, on peut accéder à l'application en utilisant l'adresse IP et le port définis dans le fichier de configuration Docker Compose.

Création de l'application web

Nous avons créé un fichier principal pour l'application, appelé app.py. Dans ce fichier, nous avons initialisé Flask :

```
import os
from flask import Flask, render_template, request, url_for, redirect,jsonify
from flask_caching import Cache
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.sql import func

app = Flask(__name__)
cache = Cache(app, config={'CACHE_TYPE': 'redis', 'CACHE_REDIS_URL':
'redis://cache:6379/0'})
app.config['SQLALCHEMY_DATABASE_URI'] =
"mysql+mysqlconnector://user:user123@db:3306/flask_api"
db = SQLAlchemy(app)
```

app.py

On a commencé par créer une instance app de l'application Flask, puis on a créé un objet Cache configuré pour utiliser Redis en tant que backend de cache. Le cache est stocké dans la base de données 0 de Redis.

Nous avons aussi configuré l'URL de la base de données SQLAlchemy utilisée pour communiquer avec la base de données MySQL. Il utilise le pilote MySQL Connector pour établir la connexion, spécifie le nom d'utilisateur et le mot de passe pour accéder à la base de données, et définit le nom de la base de données et le port à utiliser.

Enfin nous avons créé une instance de SQLAlchemy *db*, qui est utilisée pour effectuer des opérations de base de données telles que la création de tables, l'ajout, la suppression et la mise à jour de données dans la base de données MySQL.

Il s'agissait ensuite de mettre en place la base de données. Nous avons utilisé l'ORM SQLAlchemy pour configurer la base de données et créer les modèles pour les livres, les auteurs, les éditeurs et les catégories :

```
class Auteur(db.Model):
id = db.Column(db.Integer, primary_key=True)
nom = db.Column(db.String(128), nullable=False)

class Editeur(db.Model):
id = db.Column(db.Integer, primary_key=True)
nom = db.Column(db.String(128), nullable=False)

class Categorie(db.Model):
id = db.Column(db.Integer, primary_key=True)
code = db.Column(db.String(128), nullable=False)
nom = db.Column(db.String(128), nullable=False)
```

app.py

Il s'agissait ensuite de définir les routes de l'application :

```
@app.route("/biblio", methods=["GET", "POST"])
def home(): #générer une page web qui affiche la liste de tous les livres
disponibles dans la bibliothèque
livres = Livres.query.all()
# return render_template ('biblio.html', livres=livres)
```

app.py

Voici l’exemple de la route "/biblio" qu’on a défini à l'aide du décorateur @app.route, qui peut gérer à la fois les requêtes GET et POST. La fonction home() est appelée lorsque cette route est accédée. À l'intérieur de cette fonction, elle récupère toutes les lignes de la table "Livres" en utilisant la méthode query.all() de SQLAlchemy, et les stocke dans la variable "livres".

Cette fonction retourne ensuite un modèle rendu "biblio.html" à l'aide de la fonction render_template() de Flask, et passe la variable "livres" au modèle pour le rendu. Le but du modèle "biblio.html" est de générer une page Web qui affiche la liste de tous les livres disponibles dans la bibliothèque.

```
{% extends 'main.html' %}

{% block content %}
<h1 class="title">{% block title %} Livres {% endblock %}</h1>
<div class="content">
{% for livre in livres %}
<div class="student">
<p><b>#{{ livre.id }}</b></p>
<b>
<p class="name">{{ livre.titre }} {{ livre.description }}</p>
</b>
<p>{{ livre.auteur }}</p>
<p>{{ livre.categorie }}</p>
<p>Ajouté le : {{ livre.date_creation }}</p>
<div class="bio">
<h4>Description</h4>
<p>{{ livre.bio }}</p>
</div>
</div>
{% endfor %}
</div>
{% endblock %}
```

biblio.html

Nous avons utilisé le moteur de template Jinja2 qui permet de générer du HTML dynamique en utilisant des variables, des boucles et des conditions. On définit un bloc de contenu appelé "content" qui étend un modèle de page principale ("main.html"). Dans ce bloc de contenu, il y a une boucle qui itère sur chaque livre dans la liste "livres" et qui affiche les détails de chaque

livre, tels que l'identifiant, le titre, la description, l'auteur, la catégorie, la date de création et la biographie.

Le script Shell gerer-conteneurs.sh

Ce script shell permet de lancer une infrastructure de développement de manière automatisée pour une application web WSGI, comprenant une base de données MySQL et un serveur web.

La première partie permet de convertir un fichier CSV en une base de données SQL en utilisant des requêtes SQL d'insertion :

```
#importer à partir d'un fichier csv

# Check if the filename is set as environment variable
if [[ -z "${CSV_FILE}" ]]; then
CSV_FILE="biblio.csv"
fi
# Create an empty SQL file
DB_FILE=biblio.sql
touch $DB_FILE
# Skip the first line
tail -n +2 $CSV_FILE > csv_file_data.csv
while IFS=, read -r titre description isbn annee_apparition image auteur editeur categorie
do
echo "
INSERT INTO Livres (titre, description, isbn, annee_apparition, image, auteur, editeur, categorie)
VALUES ('$titre', '$description', '$isbn', '$annee_apparition', '$image', '$auteur', '$editeur','$categorie' );" >> $DB_FILE
done < csv_file_data.csv
```

gerer-conteneur.sh

On vérifie si une variable CSV_FILE est définie, sinon on la définit avec la valeur "biblio.csv". Puis, on crée un fichier SQL vide nommé "biblio.sql". La troisième étape est d'extraire les données à partir du fichier CSV, en sautant la première ligne (l'en-tête) et de stocker les données dans un nouveau fichier CSV nommé "csv_file_data.csv". On va ensuite parcourir les lignes du fichier "csv_file_data.csv" et insérer les données dans le fichier SQL créé précédemment en utilisant des requêtes SQL d'insertion. Chaque ligne du fichier CSV est divisée en champs en utilisant la virgule comme séparateur (IFS=,). Les champs sont stockés dans des variables correspondant aux colonnes de la table de la base de données. Une requête SQL d'insertion est créée pour chaque ligne, avec les valeurs de champ correspondantes insérées dans la requête. Et on termine par ajouter la requête SQL d'insertion au fichier SQL en utilisant l'opérateur de redirection ">>".

Les enjeux

L'utilisation d'outils logiciels modernes est importante dans le développement de logiciels car elle permet améliorer la qualité et la productivité logicielle.

En effet, ils permettent une collaboration efficace entre les membres de l'équipe, quelle que soit leur localisation. Les outils de gestion de projets, les outils de suivi de problèmes, les outils de gestion de versions et les plates-formes de communication en temps réel permettent à l'équipe de travailler de manière synchronisée, même si les membres de l'équipe sont dispersés dans le monde entier. C'est le cas de Github, un service web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions Git. C'est un outil collaboratif qui nous a permis de travailler ensemble et à distance, en fournissant des outils de gestion des versions, de suivi des problème, etc.

Les conteneurs Docker offrent une grande flexibilité en permettant leur déploiement sur différentes plateformes, sans dépendre de l'environnement d'exécution. Ils facilitent également le processus de développement en garantissant une cohérence dans le fonctionnement du code à chaque étape. Les conteneurs assurent une isolation complète entre les applications et les systèmes d'exploitation, évitant les conflits de dépendances et offrant un environnement sécurisé. Les conteneurs sont également légers et portables, permettant une création, un test et un déploiement rapides. Enfin, les conteneurs favorisent la collaboration entre les développeurs en permettant le partage d'environnements de développement et d'images de conteneurs pour un travail en équipe efficace.

Visual Studio Code est un éditeur de code rapide et léger avec une prise en charge étendue de plusieurs langages de programmation. Il intègre une fonctionnalité native de gestion de contrôle de version Git, qui simplifie la collaboration sur les projets. VS Code est également équipé d'un débogueur intégré qui permet la détection et la résolution des erreurs dans les applications. De plus, VS Code possède une grande variété d'extensions qui peuvent être intégrées avec d'autres outils de développement. Nous avons intégré plusieurs extensions telles que Docker, Flask, Github, SQLAlchemy, ce qui a permis d'obtenir un environnement de développement intégré. En somme, l'utilisation de Visual Studio Code a permis une meilleure productivité, collaboration, personnalisation, intégration avec d'autres outils, ainsi qu'une amélioration de la qualité du code en travaillant de manière plus efficace.

L’architecture logicielle

Nous avons utilisé le micro-framework Flask pour la création de l'application, vous avons donc opté pour une architecture de type Modèle-Vue-Contrôleur (MVC), qui est un modèle courant pour les applications web. Il vise à diviser les tâches d'une application en trois parties bien distinctes afin de les rendre plus facilement modifiables. Chaque partie est responsable d'une tâche spécifique, ce qui permet d'isoler les différentes couches de l'application et de mieux les organiser.

Le modèle représente les données et la logique de l'application et la représentation des données. Dans notre application, le modèle est décrit par l’ORM SQLAlchemy. Il permet d’effectuer des opérations sur la base de données mySQL, telles que la création, la mise à jour et la suppression, etc.

Pour commencer, nous avons définit les classes Livre, Auteur, Editeur et Catégorie. La classe Livre a des attributs tels que l’id, le titre, la description, l'ISBN, l’année d’apparition, l’image, la catégorie, la date de création et la date de modification. Elle aura également des méthodes rechercher un livre par titre ou par auteur. La classe Auteur a comme attributs l’id et le nom. Elle a également les méthodes pour rechercher un auteur par id ou par nom. La classe Editeur a comme attributs l’id et le nom et aura également des méthodes pour rechercher un éditeur par nom ou par id. La classe Catégorie a comme attributs l’id le code et le nom et également des méthodes pour rechercher une catégorie par id ou par code.

La vue représente l'interface utilisateur de l'application. Elle gère responsable l'affichage des données et de l'interaction avec l'utilisateur. Dans notre application Flask, la vue est une fonction Python qui traite les requêtes HTTP reçues et de renvoie la réponse en HTML.

Le contrôleur agit comme un intermédiaire entre le modèle et la vue. Le contrôleur gère les événements utilisateur et utilise le modèle pour mettre à jour les données et répondre aux actions de l'utilisateur. Dans notre application Flask, le contrôleur est représenté par les fonctions de routage qui sont utilisées pour répondre aux requêtes HTTP et manipuler les données du modèle.

Par exemple, lorsqu’un utilisateur effectue une recherche de livre par titre, il va interagir avec la vue en saisissant le titre dans le champ de saisie du formulaire. Cela va envoyer une requête HTTP au contrôleur, qui est ici notre fonction de routage *filtre_recherche()*. Cette fonction va ensuite utiliser le modèle via SQLAlchemy pour récupèrer la liste des livres ayant le titre recherché, puis va renvoyer la vue via le fichier HTML nomdufichier, pour afficher la liste trouvée.

En utilisant cette structure de conception, les différentes parties de l'application sont séparées en modules distincts et faciles à comprendre et à gérer. Cette meilleure organisation rend le développement, le débogage et la maintenance de l'application plus faciles et plus efficaces.

Conclusion

Ce projet nous a permis de développer une application web de gestion de bibliothèque personnelle en utilisant le langage Python, le micro-framework Flask, le toolkit/ORM SQLAlchemy, Visual Studio Code et Github. Docker Compose a été une outil pratique pour définir et exécuter des applications multi-conteneurs. Il permet de décrire les conteneurs nécessaires pour faire fonctionner l'application ainsi que les services associés tels que la base de données, dans un seul fichier de configuration.

En séparant les différentes responsabilités de l'application et en facilitant le déploiement sur différentes plateformes, les conteneurs Docker ont offert une isolation complète entre les applications et les systèmes d'exploitation, tandis que l'utilisation de GitHub a permis une collaboration efficace entre les membres de l'équipe. Enfin, Visual Studio Code a amélioré la productivité de l'équipe en permettant une la complétion de code, la détection d'erreurs et une intégration avec d'autres outils de développement. Grâce à ces outils, le projet a abouti à la création d'une application fonctionnelle, mettant en pratique de nombreuses compétences en matière de développement logiciel, de développement web, de gestion de bases de données et de collaboration en équipe en utilisant des outils modernes. Nous avons également rencontré des défis techniques tels que la création de modèles de données et la création des conteneurs des serveurs à l'étape 4, que nous avons pu surmonter grâce à notre travail d'équipe et à nos efforts de recherche.