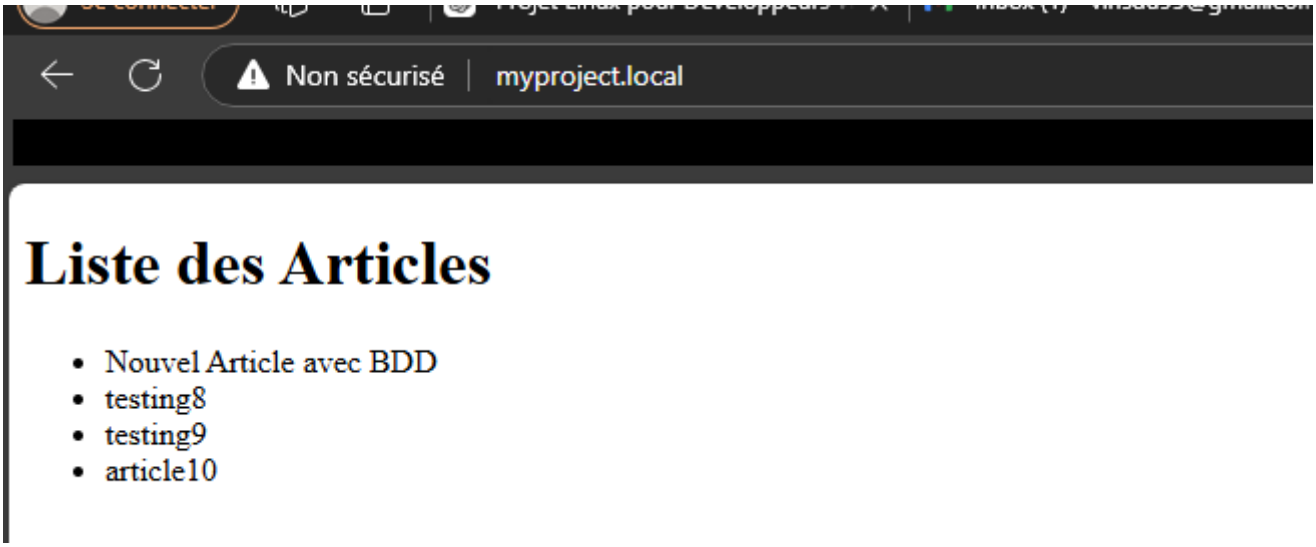
```
Author : Vincent LAINE
Company: ANDN
School: ESGI
Class: IW 3
```

TP : Mise en place d'une application Web multi-services avec Flask, WebSocket, et Nginx

Objectif :

Développer une application web qui affiche une liste d'articles récupérée via une API Flask. La mise à jour des articles sera gérée en temps réel via un serveur WebSocket. Un proxy Nginx sera utilisé pour gérer les requêtes HTTP et WebSocket.



Fonctionnement global :

1. **Flask API** (`app.py`) :

- Gère les articles (GET et POST).
- Lorsqu'un nouvel article est ajouté (POST), Flask envoie une requête HTTP POST à ton serveur WebSocket pour notifier tous les clients connectés.
- Accessible via `http://myproject.local/api/`

2. **Serveur WebSocket** (`websocket.go`) :

- Gère les connexions WebSocket, maintient une liste des clients connectés.
- Envoie des notifications `"Nouveau message reçu"` à tous les clients connectés lorsque Flask envoie une notification.
- Accessible via `http://myproject.local/ws/`

1. **Interface HTML / JavaScript** (`index.html`) :

- Se connecte au serveur WebSocket.
- Met à jour la liste des articles lorsque le serveur WebSocket envoie une notification.
- Accessible via `http://myproject.local/`

4. Base de donnée
    - Stocke les informations dans une base de donnée pour la persistance
5. Utiliser des services pour démarrer les applications

Récapitulatif final des étapes du TP :

1. Préparer l'environnement (installation de Python, Go, Nginx, SQLite).
2. Configurer Flask avec SQLite et Flask-SQLAlchemy pour stocker les articles dans une base de données persistante.
3. Mettre en place un serveur WebSocket en Go pour les notifications en temps réel.
4. Configurer Nginx comme proxy pour gérer les requêtes HTTP et WebSocket.
5. Créer une interface HTML/JavaScript pour interagir avec l'API Flask et le serveur WebSocket.
6. Tester le fonctionnement en temps réel de l'application avec l'ajout d'articles via `curl` et les notifications WebSocket.

Étape 1 : Préparer l'environnement

1. **Installez les dépendances nécessaires :**

- Python, Go, Nginx, SQLite, etc.
- **Piste :** Utilisez le gestionnaire de paquets pour installer les logiciels nécessaires à la mise en place de votre environnement.

1. **Configurer un domaine local** (facultatif) :

- Modifiez le fichier hôte de votre machine pour utiliser un domaine local, comme `myproject.local`.

Étape 2 : Créer l'API Flask

1. **Créer un projet Flask :**

- Créez un dossier pour le projet et configurez un environnement Python. Vous devez installer Flask et toute autre dépendance nécessaire pour faire tourner une API.

1. **Écrire une API Flask :**

- Implémentez une API Flask qui gère deux routes : une pour récupérer les articles (GET) et une pour en ajouter (POST).

Code

```
import requests
from flask import Flask, jsonify, request

app = Flask(__name__)

articles = [{"id": 1, "title": "Premier Article"}]

@app.route('/api/articles', methods=['GET'])
def get_articles():
return jsonify(articles)

@app.route('/api/articles', methods=['POST'])
def add_article():
new_article = request.json
articles.append(new_article)

# Notifie le serveur WebSocket après un POST
try:
# Envoie une requête POST à ton serveur WebSocket pour notifier les clients
requests.post('http://localhost:8080/notify')
except Exception as e:
```

```
print(f"Erreur lors de l'envoi de la notification WebSocket: {e}")

return jsonify(new_article), 201

if __name__ == '__main__':
app.run(host='0.0.0.0', port=5000)
```

2. **Lancer l'API Flask** :

- L'application doit fonctionner en arrière-plan pour pouvoir interagir avec un serveur WebSocket plus tard.
- **Piste :** Utilisez un outil pour garder l'application en cours d'exécution, même après avoir quitté le terminal.

Étape 3 : Créer un serveur WebSocket en Go

1. **Créer un projet Go :**

- Créez un projet pour un serveur WebSocket.
- **Piste :** Vous aurez besoin d'une bibliothèque spécifique pour gérer les connexions WebSocket en Go.

Installer Gorilla WebSocket :

```
go get github.com/gorilla/websocket
```

1. **Implémentez un serveur WebSocket :**

- Le serveur doit gérer les connexions des clients WebSocket et envoyer des notifications lorsqu'un nouvel article est ajouté via l'API Flask.*

Code

```
package main

import (
    "log"
    "net/http"
    "github.com/gorilla/websocket"
)

var upgrader = websocket.Upgrader{
    CheckOrigin: func(r *http.Request) bool {
        return true
    },
}

var clients = make(map[*websocket.Conn]bool) // clients WebSocket connectés

func handleWebSocket(w http.ResponseWriter, r *http.Request) {
    conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        log.Println(err)
        return
    }
    defer conn.Close()
    clients[conn] = true

    for {
```

```
        _, message, err := conn.ReadMessage()
        if err != nil {
            log.Println("Erreur WebSocket:", err)
            delete(clients, conn)
            break
        }
        log.Printf("Message reçu : %s", message)
    }
}

// Fonction pour notifier tous les clients WebSocket
func notifyClients() {
    for client := range clients {
        err := client.WriteMessage(websocket.TextMessage, []byte("Nouveau message reçu"))
        if err != nil {
            log.Println("Erreur lors de l'envoi du message :", err)
            client.Close()
            delete(clients, client)
        }
    }
}

// Endpoint HTTP pour recevoir les notifications de Flask
func notifyHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Notification reçue de Flask")
    notifyClients() // Notifie tous les clients WebSocket
}

func main() {
    http.HandleFunc("/ws/", handleWebSocket)
    http.HandleFunc("/notify", notifyHandler) // Endpoint pour les notifications Flask
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

1. **Lancer le serveur WebSocket :**

- Assurez-vous que votre serveur WebSocket tourne en arrière-plan, et qu'il est accessible depuis d'autres services.
- Utilisez `go run`

Étape 4 : Configurer Nginx pour gérer le Proxy

1. **Configurer Nginx comme proxy inverse :**

- Configurez Nginx pour rediriger les requêtes HTTP vers l'API Flask et les connexions WebSocket vers votre serveur Go.

Enxemple d'un bout du code du reverse proxy

```
location /ws/ {
proxy_pass http://127.0.0.1:8080;
proxy_http_version 1.1;
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
```

Le reste de la configuration de nginx est une configuration de reverse proxy classique.

Étape 5 : Créer l'interface HTML

1. **Créer une interface utilisateur :**

- Créez un fichier HTML qui interagit avec votre API Flask et votre serveur WebSocket.

Code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Application Web</title>
</head>
<body>
    <h1>Liste des Articles</h1>
    <ul id="articles"></ul>

    <script>
        // Fonction pour actualiser la liste des articles
        function updateArticles() {
            fetch('/api/articles')
                .then(response => response.json())
                .then(data => {
                    const articlesList = document.getElementById('articles');
                    articlesList.innerHTML = ''; // Efface la liste actuelle
                    data.forEach(article => {
                        let li = document.createElement('li');
                        li.textContent = article.title;
                        articlesList.appendChild(li);
                    });
                });
        }

        // Connexion au WebSocket
        const ws = new WebSocket('ws://myproject.local/ws/');

        // Gérer la réception des messages WebSocket
        ws.onmessage = function(event) {
            console.log('Message WebSocket reçu:', event.data);
            if (event.data === 'Nouveau message reçu') {
                // Actualise la liste des articles quand un nouveau message est reçu
                updateArticles();
            }
        };

        // Gérer la connexion établie
        ws.onopen = function() {
            console.log('Connexion WebSocket établie');
            ws.send('Message depuis le navigateur');  // Envoie un message de test
        };

        // Gérer les erreurs WebSocket
        ws.onerror = function(event) {
            console.error('Erreur WebSocket:', event);
        };
```

```
        // Gérer la fermeture du WebSocket
        ws.onclose = function(event) {
            console.log('Connexion WebSocket fermée:', event);
        };

        // Charger initialement la liste des articles
        updateArticles();
    </script>
</body>
</html>
```

1. **Connecter l'interface avec l'API Flask et le WebSocket :**

- Vous devez récupérer la liste des articles via l'API Flask et actualiser automatiquement la page lorsqu'une notification est reçue via WebSocket.

Tester l'application. Vous devriez pouvoir faire des requêtes curl POST non persistantes. Voir **Flux de travail attendu & tests** plus bas.

Étape 6 : Ajouter une base de données SQLite

1. **Ajouter une base de données pour la persistance :**

- Modifiez l'API Flask pour enregistrer les articles dans une base de données SQLite.

Étape 1 : Installer SQLite et Flask-SQLAlchemy

1. **Installer SQLite** (si ce n'est pas déjà installé sur votre machine) :

```
sudo apt install sqlite3
```

2. **Installer l'extension Flask-SQLAlchemy** dans l'environnement virtuel de Flask :

Activez l'environnement virtuel de Flask :

```
source ~/flask-api/venv/bin/activate
```

Ensuite, installez **Flask-SQLAlchemy** et **Flask-Migrate** pour gérer la base de données :

```
pip install flask-sqlalchemy flask-migrate
```

1. **Gérer les migrations de base de données :**

- Vous devrez créer une structure pour les articles dans la base de données et gérer les migrations.

Nouveau code pour Flask

```
import os
import requests
from flask import Flask, jsonify, request
from flask_sqlalchemy import SQLAlchemy
```

```python
from flask_migrate import Migrate

# Initialisation de l'application Flask
app = Flask(__name__)

# Configuration de la base de données SQLite
basedir = os.path.abspath(os.path.dirname(__file__))
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' + os.path.join(basedir, 'app.db')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# Initialisation de la base de données et des migrations
db = SQLAlchemy(app)
migrate = Migrate(app, db)

# Modèle d'articles
class Article(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(120), nullable=False)

# Endpoint pour récupérer les articles
@app.route('/api/articles', methods=['GET'])
def get_articles():
    articles = Article.query.all()
    articles_list = [{"id": article.id, "title": article.title} for article in articles]
    return jsonify(articles_list)

# Endpoint pour ajouter un nouvel article
@app.route('/api/articles', methods=['POST'])
def add_article():
    new_article_data = request.json
    new_article = Article(title=new_article_data['title'])
    db.session.add(new_article)
    db.session.commit()

    # Notifie le serveur WebSocket après un POST
    try:
        requests.post('http://localhost:8080/notify')
    except Exception as e:
        print(f"Erreur lors de l'envoi de la notification WebSocket: {e}")

    return jsonify({"id": new_article.id, "title": new_article.title}), 201

# Exécution de l'application
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Étape 7 : Déployer avec des services systemd

1. **Créer des services systemd pour gérer Flask et WebSocket :**

- Créez des fichiers de service qui permettront de lancer et de surveiller les applications Flask et Go.
- **LES APPLICATIONS DEVRONT ETRE IMPLEMENTEE DANS LES BONS DOSSIER, DONC PAS DANS LE ROOT PAR EXEMPLE**
- **Piste :**
  - Les services doivent redémarrer automatiquement en cas de plantage.
  - Il vous faudra utiliser `WorkingDirectory=` dans le fichier de service
  - Il faudra build l'application en go pour en faire un binaire exécutable depuis /bin avec `go build -o`
  - Le ExecStart de python utilisera son environnement pour lancer l'API

1. **Surveiller les services :**

- Utilisez les outils de journalisation pour surveiller le fonctionnement des services et détecter les erreurs potentielles.

```
18:22:48 root@multi-webapp [redacted] # systemctl status websocket.service
● websocket.service - Serveur WebSocket Go
     Loaded: loaded (/etc[redacted]; enabled; preset: enabled)
     Active: active (running) since Tue 2024-09-24 18:22:46 CEST; 7s ago
   Main PID: 49921 [redacted]
      Tasks: 6 (limit: 4620)
     Memory: 2.6M
        CPU: 4ms
     CGroup: /system.slice/websocket.service
             [redacted]

sept. 24 18:22:46 multi-webapp systemd[1]: Started websocket.service - Serveur WebSocket Go.
18:22:53 root@multi-webapp [redacted] # systemctl status flask-api.service
● flask-api.service - API Flask
     Loaded: loaded (/etc[redacted]disabled; preset: enabled)
     Active: active (running) since Tue 2024-09-24 16:31:39 CEST; 1h 51min ago
   Main PID: 49663 [redacted]
      Tasks: 1 (limit: 4620)
     Memory: 50.7M
        CPU: 1.032s
     CGroup: /system.slice/flask-api.service
             [redacted]

sept. 24 16:35:02 multi-webapp python3[49663]: 127.0.0.1 - - [24/Sep/2024 16:35:02] "POST /api/articles HTTP/1.0" 201 -
sept. 24 16:35:02 multi-webapp python3[49663]: 127.0.0.1 - - [24/Sep/2024 16:35:02] "GET /api/articles HTTP/1.0" 200 -
sept. 24 17:06:03 multi-webapp python3[49663]: 127.0.0.1 - - [24/Sep/2024 17:06:03] "GET /api/articles HTTP/1.0" 200 -
sept. 24 17:14:00 multi-webapp python3[49663]: 127.0.0.1 - - [24/Sep/2024 17:14:00] "POST /api/articles HTTP/1.0" 201 -
sept. 24 17:14:15 multi-webapp python3[49663]: 127.0.0.1 - - [24/Sep/2024 17:14:15] "GET /api/articles HTTP/1.0" 200 -
sept. 24 17:31:07 multi-webapp python3[49663]: 127.0.0.1 - - [24/Sep/2024 17:31:07] "POST /api/articles HTTP/1.0" 201 -
sept. 24 17:31:14 multi-webapp python3[49663]: 127.0.0.1 - - [24/Sep/2024 17:31:14] "GET /api/articles HTTP/1.0" 200 -
```

Flux de travail attendu & tests :

1. **Ajout d'un article avec POST (via `curl`) :**

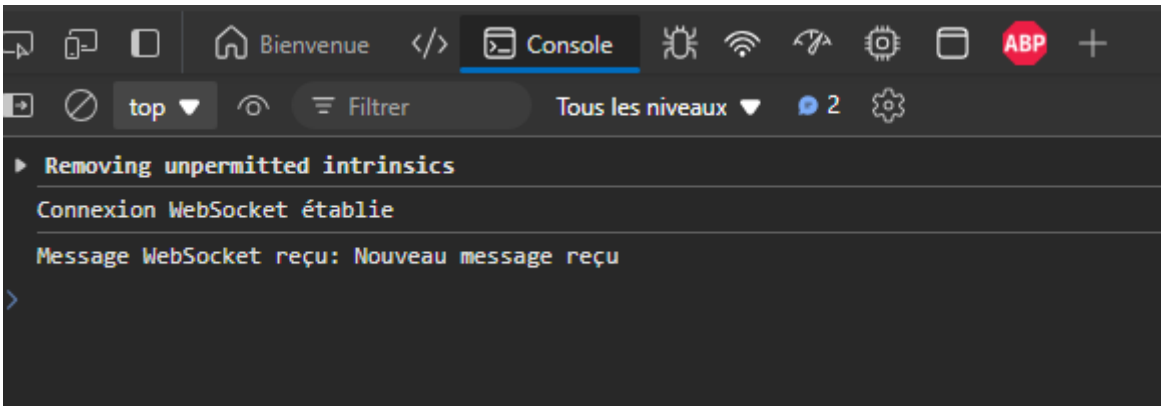
- Quand tu fais un POST avec `curl` :

```
curl -X POST http://myproject.local/api/articles -H "Content-Type: application/json" -d '{"id": 3, "title": "mon super article"}'
```

- Flask ajoute l'article à la liste et envoie une notification au serveur WebSocket.

4. **Notification via WebSocket :**

- Le serveur WebSocket reçoit la requête de Flask et envoie un message `"Nouveau message reçu"` à tous les clients connectés.
- Les clients WebSocket (dans ton navigateur) reçoivent ce message et déclenchent l'actualisation de la liste des articles.



5. Persistance dans la base de donnée.

- Après ces requêtes, les infos sont stockés dans la base de donnée.

```
sqlite> SELECT * FROM article;
1|Nouvel Article avec BDD
2|testing8
3|testing9
4|article10
5|mon super article
6|mon super article
sqlite>
```

Points importants à découvrir par vous-même :

1. Comment installer et configurer les dépendances pour chaque technologie.
2. Comment faire communiquer Flask, Go, et Nginx ensemble.
3. Comment gérer une base de données SQLite avec Flask.
4. Comment utiliser les services systemd pour automatiser le démarrage des applications.

Bonus

Les bonus sont à effectuer EN PLUS du projet de base. Ils ne le remplace pas.

- Conteneurisation en micro services de l'application
- Utilisation de vos propres projets web
- Ajouter un aspect sécurité comme du HTTPS
- Au choix, surprenez moi.