Explication de l'interface web et de l'API

Ce code Python est une application Flask qui fournit une interface pour consulter des logs stockés dans une base de données MongoDB. Elle permet également de vérifier l'état de différents services réseau en fonction de leur disponibilité sur certaines connexions et de générer des graphiques du nombre de logs générés par minute pour les 30 dernières minutes.

Importation des modules

```
import socket
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
import requests
from flask import Flask, render_template, jsonify
from flask_restful import Api
from pymongo import MongoClient
```

Le programme commence par importer les modules nécéssaires :

- `socket` : permet de créer une connexion réseau.
- `datetime` : fournit des classes pour manipuler des dates et des heures.
- `timedelta` : permet de calculer des différences entre deux dates.

- `matplotlib.pyplot` : permet de créer des graphiques à partir de données.
- `requests` : permet d'envoyer des requêtes HTTP.
- `Flask` : est un framework web pour Python.
- `render_template` : permet de générer des pages web à partir de modèles HTML.
- `jsonify` : permet de convertir des objets en format JSON.
- `Flask_restful` : permet de créer des API RESTful avec Flask.
- `MongoClient` : est une classe fournie par PyMongo, une bibliothèque Python pour interagir avec MongoDB.

Initialisation des objets

```
app = Flask(__name__)
api = Api(app)
client = MongoClient("mongodb://root:example@127.0.0.1:27017/")
db = client["logs_db"]
logs = db["logs"]
```

Cette partie du code initialise les objets qui seronts utilisés tout au long de l'execution

- On crée une instance de la classe Flask, qui est un objet qui représente l'application web. Flask est un framework Python pour le développement web.
- Ensuite, on crée une instance de la classe Api qui est un outil permettant de créer une API RESTful en utilisant Flask.
- Puis on initialise un client MongoDB en utilisant la bibliothèque pymongo. La fonction MongoClient prend en argument une URI de connexion à la base de données MongoDB. Dans ce script, l'URI de connexion est "mongodb://root:example@127.0.0.1:27017/", qui se connecte à une instance de MongoDB exécutée localement (le docker étant executé avec le paramètre `--network host`, le docker partage les ports de l'hôte. Nous pouvons donc utiliser l'adresse 127.0.0.1 pour atteindre le docker si il est hébergé sur la même machine que ce script) avec l'utilisateur "root" et le mot de passe "example".

La quatrième ligne de code sélectionne la base de données "logs_db" dans MongoDB.

La cinquième ligne de code sélectionne la collection "logs" dans la base de données "logs_db". Une collection dans MongoDB est similaire à une table dans une base de données relationnelle. C'est dans cette collection que seront stockés les logs.

Les différentes API

Liste des différents services, et du nombre total de logs par service

```python
@app.route('/services')
def services():
    # Recuperation de toutes les valeurs (distinctes) de la clé "service"
    services = logs.distinct("service")
    # Initialisation du dictionnaire de résultats
    result = {}
    # Boucle sur les services
    for service in services:
        # Récupération du nombre de documents générés par service
        count = logs.count_documents({"service": service})
        # Ajout du résultat au dictionnaire
        result[service] = count

    # Retour du dictionnaire de résultats
    return jsonify(result)
```

Cette partie de code définit une route de l'API Flask qui permet de récupérer le nombre de documents générés depuis la dernière seconde pour chaque service.

Dans un premier temps, la fonction récupère toutes les valeurs distinctes de la clé "service" stockées dans la collection "logs" de la base de données MongoDB. Ensuite, elle initialise un dictionnaire vide qui sera utilisé pour stocker les résultats, puis elle boucle sur chaque service récupéré. Pour chaque service, la fonction récupère le nombre de documents générés par ce service en effectuant une requête dans la base de données avec la méthode `count_documents` de la collection "logs". Enfin, la fonction ajoute le résultat au dictionnaire de résultats, avec le nom du service comme clé.

Finalement, la fonction retourne le dictionnaire de résultats sous forme de réponse JSON grâce à la méthode `jsonify` de Flask.

Generation du graph

```python
@app.route('/logs/minutes')
def logs_per_minute():
    # Récupération des données de la base de données
    now = datetime.now()
    start_time = now - timedelta(minutes=30)
    logs_list = list(logs.find({"timestamp": {"$gte": start_time.timestamp()}}, {"_id": -1,
"timestamp": 1}))

    # Création d'un dictionnaire pour stocker le nombre de logs par minute
    logs_per_minute = {}
    for log in logs_list:
        timestamp = datetime.fromtimestamp(log["timestamp"])
        minute = timestamp.replace(second=0, microsecond=0)
        if minute in logs_per_minute:
            logs_per_minute[minute] += 1
        else:
            logs_per_minute[minute] = 1

    # Création d'un graphique à partir des données
    plt.figure()
    plt.title("Nombre de logs par minute (30 dernières minutes)")
    plt.xlabel("Temps")
    plt.ylabel("Nombre de logs")
    plt.xticks(rotation=45)
    x = []
    y = []
    for minute, count in logs_per_minute.items():
        x.append(minute.strftime("%Y-%m-%d %H:%M"))
        y.append(count)
    plt.plot(x, y)
    plt.tight_layout()
    plt.savefig("static/logs_per_minute.png")
    return {"status": "ok"}
```

Cette fonction crée un graphique représentant le nombre de logs enregistrés par minute dans la base de données pour les 30 dernières minutes. Voici une description des étapes clés de la fonction :

- La fonction récupère tous les logs enregistrés dans la base de données au cours des 30 dernières minutes.
- Elle créée un dictionnaire pour stocker le nombre de logs enregistrés pour chaque minute.
- Elle itère sur chaque log pour obtenir l'horodatage (timestamp) et détermine la minute à laquelle il a été enregistré.

- Elle incrémente le compteur de logs pour cette minute dans le dictionnaire logs_per_minute.
- Elle créée un graphique à partir des données stockées dans le dictionnaire logs_per_minute. Elle utilise la bibliothèque matplotlib pour créer le graphique.
- Le graphique est enregistré dans le dossier statique du serveur Web.
- La fonction renvoie un objet JSON indiquant que la requête a réussi.

Il est important de noter que la fonction ne renvoie pas le graphique lui-même, mais simplement un objet JSON pour indiquer que la requête s'est bien déroulée. Le graphique peut ensuite être affiché dans une page web en utilisant une balise HTML `<img>` pointant vers le fichier `static/logs_per_minute.png`.

Collecte des N derniers logs pour un ou tous les services

```
@app.route('/logs/<string:service>/<int:limit>')
def get_logs(service, limit):
    if service == "all":
        result = logs.find({}).sort("id", -1).limit(limit)
    else:
        result = logs.find({"service": service}).sort("id", -1).limit(limit)
    return [{"service": log["service"],
             "id": str(log["_id"]),
             "timestamp": log["timestamp"],
             "filename": log["filename"],
             "line": log["line"]}
            for log in result]
```

La fonction `get_logs()` est une route qui permet de récupérer les derniers logs pour un service donné, ainsi que les informations associées, tels que le nom du fichier et le numéro de ligne dans le fichier où le log a été généré.

La route prend deux arguments : `service` et `limit`. `service` est une chaîne de caractères représentant le nom du service pour lequel on veut récupérer les logs, et `limit` est un entier qui représente le nombre de logs que l'on souhaite récupérer.

Le code commence par vérifier si le nom du service fourni est "all". Si tel est le cas, tous les logs sont récupérés en utilisant la méthode `find()` de MongoDB sans aucun filtre. Sinon, seuls les logs du service spécifié sont récupérés en utilisant la méthode `find()` avec le filtre `"service": service`. Dans les deux cas, les logs sont triés par ordre décroissant d'ID (qui est

une clé qui s'incrément à chaque insert de document dans la base de donnée), et le nombre de logs est limité au nombre spécifié par la variable `limit`.

Enfin, la fonction retourne une liste de dictionnaires, où chaque dictionnaire contient les informations sur un log spécifique. Ces informations sont extraites des champs de chaque document, tels que le nom du service, l'ID du document (sous forme de chaîne de caractères), le timestamp, le nom du fichier et le numéro de ligne dans le fichier où le log a été généré. Cette liste est renvoyée sous forme de réponse JSON.


```
[
  {
    "_id": {"$oid": "63ee2f2c8ac4f7587ddd64c5"},
    "filename": "daemon.log",
    "id": 0,
    "line": "Apr 11 06:51:51 metasploitable named[4541]: starting BIND 9.4.2 -u bind\n",
    "service": "daemon",
    "timestamp": 1676554028.0037723
  },
  ...
]
```


Affichage de l'état des status

```python
def check_service_status(hostname, port):
    try:
        sock = socket.create_connection((hostname, port), timeout=2)
        return True
    except OSError:
        return False

@app.route('/status')
def status():
    hostname = '127.0.0.1'
    apache_status = 'Running' if check_service_status(hostname, 80) else 'Stopped'
    mysql_status = 'Running' if check_service_status(hostname, 3306) else 'Stopped'
    telnet_status = 'Running' if check_service_status(hostname, 23) else 'Stopped'
    return render_template(
        'status.html',
        apache_status=apache_status,
        mysql_status=mysql_status,
        telnet_status=telnet_status)
```

La fonction `status()` est une route Flask qui renvoie l'état de trois services sur un serveur local. Cette fonction utilise la fonction `check_service_status()` pour vérifier l'état de chaque service. La fonction `check_service_status()` essaie de créer une connexion avec l'hôte et le port spécifiés et renvoie `True` si la connexion est réussie, sinon elle renvoie `False`.

La fonction `check_service_status()` est une fonction qui vérifie si un service est en cours d'exécution en essayant de se connecter à un hôte distant sur un port spécifique. Elle prend en entrée l'adresse IP (`hostname`) et le numéro de port (`port`) d'un service à vérifier.

La fonction essaie de créer une connexion en utilisant la méthode `socket.create_connection()` de la bibliothèque Python `socket`. Si la connexion réussit, elle retourne `True`, ce qui signifie que le service est en cours d'exécution. Sinon, elle retourne `False`, ce qui signifie que le service n'est pas en cours d'exécution ou qu'il est inaccessible.

Dans la route `status()`, trois ports sont vérifiés : le port 80 pour Apache, le port 3306 pour MySQL et le port 23 pour Telnet. Les résultats de chaque vérification sont stockés dans les variables `apache_status`, `mysql_status` et `telnet_status`, qui sont passées à la fonction `render_template()` pour générer une réponse HTML. Cette réponse HTML contiendra les états des trois services.

Ainsi, cette fonction permet de vérifier rapidement l'état de ces trois services sur un serveur local et de renvoyer cette information sous forme de page HTML.

Affichage de l'interface Dashboard

```
@app.route('/')
def index():
    return render_template("index.html")
```

La fonction `index()` va générer la page web du Dashboard. Cette page contient du code JavaScript qui va s'occuper de générer l'affichage de toutes les informations

Le code JavaScript utilise la bibliothèque jQuery afin de charger et l'actualiser les données dans rafraichir la page. Il s'agit d'un script qui met à jour une page web toutes les quelques secondes pour afficher des informations sur les logs et les services en cours d'exécution.

Le code utilise également `setInterval()` pour appeler les fonctions `updateServices()`, `updateLogs()` et `refreshLogsGraph()` toutes les quelques secondes pour mettre à jour les informations sur la page.

```
$(document).ready(function () {
    updateServices();
    updateLogs();
    setInterval(updateServices, 5000);
    setInterval(updateLogs, 5000);
    setInterval(refreshLogsGraph, 60000);
});
```

Le code s'exécute lorsque le document est chargé, en appelant la fonction `$(document).ready(function(){...})`. Cette fonction met ensuite en place trois fonctions qui sont appelées toutes les quelques secondes pour mettre à jour les informations sur la page.

```
function updateServices() {
    $.getJSON("/services", function (data) {
        $("#apache_access").text(data.apache_access);
        $("#apache_error").text(data.apache_error);
        $("#auth").text(data.auth);
        $("#daemon").text(data.daemon);
        $("#dpkg").text(data.dpkg);
        $("#kern").text(data.kern);
        $("#mail").text(data.mail);
    });
}
```

La première fonction `updateServices()` envoie une requête GET à l'URL "/services" et récupère les données sous forme de JSON. Les données sont ensuite utilisées pour mettre à jour les éléments HTML ayant les identifiants "apache_access", "apache_error", "auth", "daemon", "dpkg", "kern" et "mail".

```
function updateLogs() {
    $.getJSON("/logs/all/20", function (data) {
        $("#logs-table").empty();
        for (var i = 0; i < data.length; i++) {
            var row = "<tr><td>" + data[i].service + "</td><td>" + data[i].filename + "</td>
<td>" + data[i].timestamp +
                "</td><td>" + data[i].line + "</td></tr>";
            $("#logs-table").append(row);
        }
    });
}
```

La deuxième fonction `updateLogs()` envoie une requête GET à l'URL "/logs/all/20" pour récupérer les dernières 20 entrées de journal système. Les données sont ensuite formatées en HTML et ajoutées à un tableau HTML ayant l'identifiant "logs-table".

```
function refreshLogsGraph() {
    // Effectue une requête GET à l'URL "/logs/minutes"
    $.ajax({
        url: "/logs/minutes",
        method: "GET",
        dataType: "json"
    })
        .done(function (data) {
            if (data.status === "ok") {  // Vérifie si la réponse contient "status: ok"
                // Rafraîchit l'image avec l'ID "logs_graph_img"
                const img = $("#logs_graph_img");
                img.attr("src", "{{ url_for('static', filename='logs_per_minute.png') }}?" +
Date.now());
            }
        })
        .fail(function (jqXHR, textStatus, errorThrown) {
            console.error(errorThrown);  // Gère les erreurs de requête
        });
}
```

La troisième fonction `refreshLogsGraph()` est appelée toutes les minutes et envoie une requête GET à l'URL "/logs/minutes" pour récupérer le nombre de messages de journal système par minute. Si la réponse contient "status: ok", elle rafraîchit l'image ayant l'identifiant "logs_graph_img" pour afficher un graphique mis à jour.