Yasemin KOCAK

TITRE PRO RNCP 37827 – Développeur-euse en Intelligence Artificielle
Promo IA 4 Microsoft By Simplon
2023 – 2025

Livrable Cas Pratique E5

Rappel du cas d'usage : Rennes Métropole dispose d'une application de surveillance du trafic et permet d'avoir en direct l'état du trafic routier de l'agglomération de Rennes.
Lien de l'application avec bugs : https://github.com/bertrandfournel/rennes_traffic_ko

Contexte et objectif : En tant que développeur IA, il vous est demandé de monitorer, débugger et mettre en place une journalisation :

- Présenter le dispositif de monitorage applicatif et donner les principales fonctions
- Définir les seuils d'alerte et les méthodes pour détecter automatiquement les incidents
- Définir une méthode pour enregistrer dans un fichier de journalisation les erreurs avec une logique d'horodatage (module logging)
- Présenter la description de l'incident (ou des incidents), la méthode de résolution et le périmètre impacté
- Présenter la (les) solution(s) apportée(s), en expliquant la méthodologie, la démarche pour comprendre la panne et les tests effectués

Lien de l'application fonctionnelle :
https://github.com/Yas2410/Livrable_E5_Debugging_Monitoring

I. RÉSOLUTION DES ERREURS DE CODE

Lors de l'exécution de l'application, la console a retourné un certain nombre d'erreurs dans le code, de types différents (syntaxiques, d'indentation, de conception, etc.)

Dans le fichier ***get_data.py***

```
        for data_dict in self.data:
        #BUG : Problème d'indentation ici après la boucle for
        temp_df = self.processing_one_point(data_dict)
        res_df = pd.concat([res_df, temp_df])

        #BUG : la syntaxe n'est pas correcte, il manque le crochet fermant
        res_df = res_df[res_df.traffic != 'unknown'

        return res_df
```

```
PROBLEMS 3  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 2  COMMENTS        zsh - rennes_traffic

2024-08-26 16:27:11.716941: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
Traceback (most recent call last):
  File "/home/yaskck/projects/Livrables_E5/rennes_traffic/app.py", line 8, in <module>
    from src.get_data import GetData
  File "/home/yaskck/projects/Livrables_E5/rennes_traffic/src/get_data.py", line 27
    temp_df = self.processing_one_point(data_dict)
    ^
IndentationError: expected an indented block after 'for' statement on line 26
(simplon_env) ➜ rennes_traffic git:(master)
```

Dans ce fichier, on peut observer un problème d'indentation au niveau de la boucle « for », ce qui empêche le bon fonctionnement du code. De plus, on peut également noter un oubli de crochet fermant après « unknown ».

Toujours dans le même fichier, lors d'une nouvelle tentative d'exécution de l'application après correction du bug cité plus haut, la console a de nouveau retourné une erreur, de type « KeyError » comme le montre la console ci-dessous :

```
    def processing_one_point(self, data_dict: dict):
        # Création du DataFrame
        temp = pd.DataFrame(
            {
                key: [data_dict[key]]
                for key in [
                    'datetime',
                    #BUG : Renvoie une KeyError
                    'traffic_status',
                    'geo_point_2d',
                    'averagevehiclespeed',
                    'traveltime',
                    'trafficstatus'
                ]
            }
        )  # noqa
```

```
  File "/home/yaskck/projects/Livrables_E5/rennes_traffic/src/get_data.py", line 17, in <dictcomp>
    key: [data_dict[key]]
KeyError: 'traffic_status'
```

Une KeyError signifie ici que le code tente d'accéder à une clé, ici « traffic_status » qui n'existerait pas dans le dictionnaire. Pour vérifier cela, il peut être intéressant, par le biais d'un nouveau fichier, d'analyser le contenu du fichier JSON contenant les données API, comme ceci :

```
affic > json_analysis.py > ...
import requests

json_url = "https://data.rennesmetropole.fr/api/explore/v2.1/catalog/datasets/etat-du-trafic-en-temps-reel/exports/json?lang=fr&timezone=Euro

response = requests.get(json_url)

response.raise_for_status()

# Chargement du json
data = response.json()

# Affichage des premières lignes pour comprendre la structure de ce dernier
print("Premiers éléments de la structure JSON:")
print(data[:2])

# Quelles sont les clés disponibles?
if data:
    first_item = data[0]
    print("Clés disponibles dans le premier élément:")
    print(first_item.keys())
```

```
(simplon_env) → rennes_traffic git:(master) ✗ python json_analysis.py
Premiers éléments de la structure JSON:
[{'datetime': '2024-08-26T17:03:00+02:00', 'predefinedlocationreference': '10273_D', 'averagevehiclespeed': 74, 'traveltime': 10, 'traveltimereliability': 85, 'traffi
cstatus': 'freeFlow', 'vehicleprobemeasurement': 3, 'geo_point_2d': {'lon': -1.6502152435538264, 'lat': 48.044792756860865}, 'geo_shape': {'type': 'Feature', 'geometr
y': {'coordinates': [[[-1.651560961036608, 48.04488583237593], [-1.649942809885061, 48.04477391733652], [-1.648869526613244, 48.04469967350696]]], 'type': 'MultiLineS
tring'}, 'properties': {}}, 'gml_id': 'rva_troncon_fcd_v1_1.1', 'id_rva_troncon_fcd_v1_1': 1, 'hierarchie': "Réseau d'armature", 'hierarchie_dv': 'Réseau de transit',
 'denomination': 'Route départementale 34', 'insee': 35206, 'sens_circule': None, 'vitesse_maxi': 70}, {'datetime': '2024-08-26T17:03:00+02:00', 'predefinedlocationre
ference': '10273_G', 'averagevehiclespeed': 72, 'traveltime': 10, 'traveltimereliability': 55, 'trafficstatus': 'freeFlow', 'vehicleprobemeasurement': 1, 'geo_point_2
d': {'lon': -1.6501987288374653, 'lat': 48.04490011863635}, 'geo_shape': {'type': 'Feature', 'geometry': {'coordinates': [[[-1.648853009091766, 48.04480703509309], [-
1.649926294608918, 48.04488127907371], [-1.651544449125304, 48.044993194341636]]], 'type': 'MultiLineString'}, 'properties': {}}, 'gml_id': 'rva_troncon_fcd_v1_1.2',
'id_rva_troncon_fcd_v1_1': 2, 'hierarchie': "Réseau d'armature", 'hierarchie_dv': 'Réseau de transit', 'denomination': 'Route départementale 34', 'insee': 35206, 'sen
s_circule': None, 'vitesse_maxi': 70}]
Clés disponibles dans le premier élément:
dict_keys(['datetime', 'predefinedlocationreference', 'averagevehiclespeed', 'traveltime', 'traveltimereliability', 'trafficstatus', 'vehicleprobemeasurement', 'geo_p
oint_2d', 'geo_shape', 'gml_id', 'id_rva_troncon_fcd_v1_1', 'hierarchie', 'hierarchie_dv', 'denomination', 'insee', 'sens_circule', 'vitesse_maxi'])
(simplon_env) → rennes_traffic git:(master) ✗
```

On peut effectivement noter l'absence de la clé « traffic_status ». Aussi, l'ajout d'un « None » à la fonction afin de gérer l'absence d'éventuelles clés est nécessaire pour éviter les erreurs. On fait de même pour la section « geo_point_2d » sinon il y aura encore des erreurs de type « KeyError ». Le code corrigé pour la gestion des « KeyError » va donner ceci :

```
def processing_one_point(self, data_dict: dict):
    # Création du DataFrame temporaire à partir du dictionnaire "data_dict"
    temp = pd.DataFrame({
    # Avec suppression de "traffic_status" bug sur l'affichage de la légende # noqa
    # On va donc garder cette clé mais ajouter la condition "None" à la fonction  # noqa
    # qui va servir de valeur par défaut si la clé est inexistante, sans renvoyer d'erreurs # noqa
        key: [data_dict.get(key, None)] for key in ['datetime',
                                                    'traffic_status',
                                                    'geo_point_2d',
                                                    'averagevehiclespeed',
                                                    'traveltime', 'trafficstatus']  # noqa
    })

    temp = temp.rename(columns={'traffic_status': 'traffic'})

    # Ici, on va également avoir une KeyError si on laisse "lattitude" et "longitude"  # noqa
    # On remplace donc comme dans le fichier json, par "lat" et "lon"
    if 'geo_point_2d' in temp.columns and temp['geo_point_2d'].notna().any():
        temp['lat'] = temp.geo_point_2d.map(lambda x: x.get('lat', None) if isinstance(x, dict) else None)  # noqa
        temp['lon'] = temp.geo_point_2d.map(lambda x: x.get('lon', None) if isinstance(x, dict) else None)  # noqa
    else:
        temp['lat'] = None
        temp['lon'] = None

    del temp['geo_point_2d']

    return temp
```

Dans le fichier ***utils.py***

Peu d'erreurs ont été relevées dans ce fichier, à l'exception d'un oubli de virgule dans les paramètres de la map au niveau du paramètre « zoom » ainsi qu'un changement au niveau du paramètre « color » où « traffic » a été remplacé par « trafficstatus » sinon cela renvoyait une erreur. Enfin, remplacer le nombre 25 dans la fonction « prediction_from_model » par 24 permet un code fonctionnel sans erreurs.

Dans le fichier ***app.py***

Plusieurs erreurs de différents types ont été relevées dans ce fichier. Tout d'abord, l'ajout de la méthode « call() » a été nécessaire ici, sinon une erreur était retournée :

```
data_retriever = GetData(url="https://data.rennesmetropole.fr/api/explore/v2.1/catalog/datasets/etat-du-tra
# Ajout de la méthode call, sinon renvoi d'erreur : "TypeError: 'GetData' object is not callable"  # noqa
data = data_retriever.call()
```

Ici, et bien que « selected_hour = request.form ['hour'] » ne retourne pas d'erreur, c'est une bonne pratique d'ajouter la méthode « get() » car cela va permettre d'éviter les erreurs en cas d'informations indisponibles. Cela va rendre le code plus robuste :

```
        # BUG : Ajouter .get ET passer
        # "selected_hour" à "cat_predict" sinon erreur
        selected_hour = request.form.get('hour')

        cat_predict = prediction_from_model(model, selected_hour)
```

Enfin, une erreur au niveau du « render_template » a été relevée. En effet, le code retournait le html « home » qui n'existe pas puisque dans le projet, il s'agit de « index.html » :


```
        fig_map = create_figure(data)
        graph_json = pio.to_json(fig_map)
        # BUG : Le fichier html ne se nomme pas "home" mais "index"
        return render_template('index.html', graph_json=graph_json)
```

```
# BUG : Le fichier html ne se nomme pas "home" mais "index"
# Ajouter aussi selected_hour
return render_template('index.html',
                       graph_json=graph_json,
                       text_pred=color_pred_map[cat_predict][0],
                       color_pred=color_pred_map[cat_predict][1],
                       selected_hour=selected_hour)
```

Dans le fichier ***index.html***

Des erreurs de balises ont été relevées dans le code html du projet. En effet, les balises fermantes étaient parfois différentes des balises ouvrantes. Une balise « style » a également été ajoutée afin d'éviter la répétition des éléments CSS directement dans les div. Enfin, le titre a également été modifié, ce qui est maintenant plus parlant que « Scatter Plot ».

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Rennes Traffic : Prédiction d'embouteillage</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-QWTKZyjpPEjISv5Wa
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDV
    <script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
    <!-- Plus propre et lisible d'ajouter ici une balise style ( à défaut d'un fichier css) avec tous les éléments css -->
    <style>
        #content {
            display: flex;
            justify-content: space-between;
        }
        #scatter-plot, #scatter-predict {
            width: 50%;
        }
    </style>
</head>
<body>

    <h2>Traffic Rennes</h2>
    <div id="content">
        <div id="scatter-plot"></div>
        <div id="scatter-predict">
            <h3>Prédiction d'embouteillage pour le centre ville</h3>
            <!-- Balise fermante différente -->
                <h4>Choisissez une heure :</h1>
                <form action="/" method="post">
```

Pour les modifications mineures touchant à l'ensemble des fichiers telles que les espaces, les sauts de ligne, et les lignes trop longues, Flake8 est un excellent outil de vérification de la qualité du code. Il vérifie la conformité du code avec les conventions de codage, afin de rendre le code plus lisible et maintenable.

II. FLASK MONITORING DASHBOARD

Le monitoring fait référence à l'ensemble des outils et méthodes utilisés pour surveiller et évaluer le fonctionnement et les performances d'une application. Il va permettre de s'assurer que cette dernière fonctionne correctement, de manière efficace et sécurisée. Parmi les outils et méthodes employés, on peut citer :

- **La surveillance en temps réel** qui va permettre de suivre les performances de façon continue
- **La détection d'anomalies** afin d'identifier des comportements inattendus
- **L'analyse de performances** par le biais de métriques détaillées
- **Les alertes** en cas de dépassement de seuils critiques

Pour les applications développées avec Flask, il existe un outil permettant de mettre en place le monitoring de l'application : Flask Monitoring Dashboard

Après avoir effectué un *pip install flask_monitoringdashboard*, le fichier app.py a été configuré comme suit :

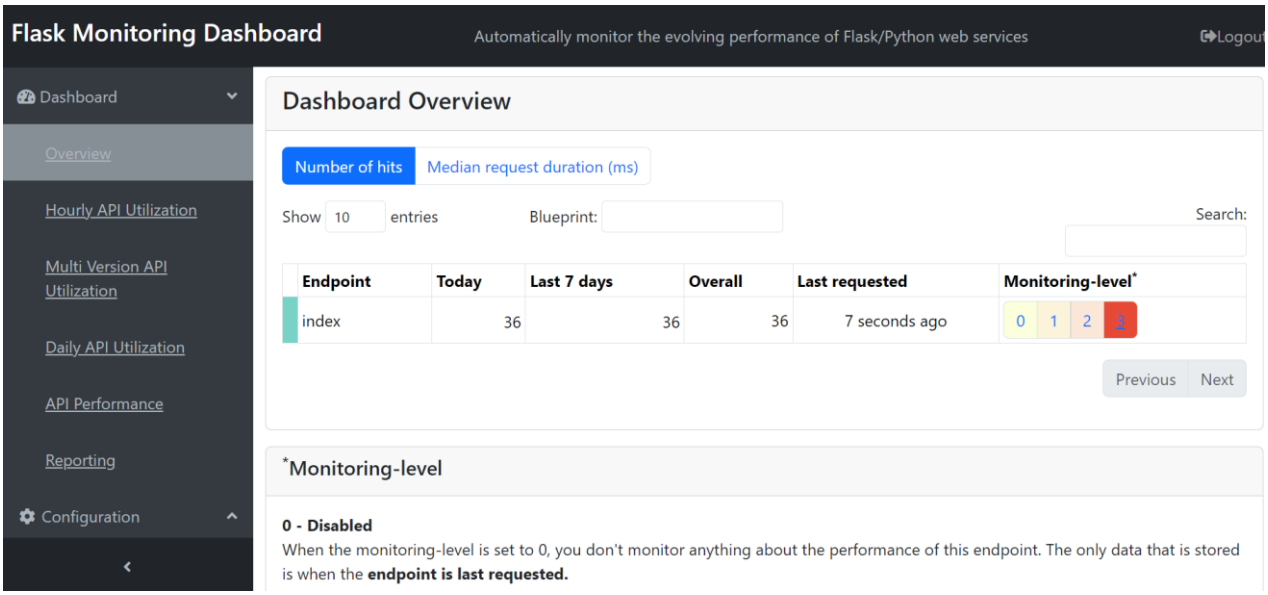
```
# Ajout du module Flask monitoring Dashboard
import flask_monitoringdashboard as dashboard # type: ignore

# Flask monitoring dashboard : Liaison du tableau de bord à l'application Flask et ajustement des params  # noqa
dashboard.bind(app)
dashboard.config.monitor_level = 3
dashboard.config.enable_logging = True

# FLASK MONITORING DASHBOARD : Si utilisation d'un fichier de config pour se connecter au dashboard  # noqa
dashboard.config.init_from(file='/config.cfg')
```

Après avoir importé le module, je vais **lier le Dashboard à mon application Flask « app »** afin que ce dernier puisse me retourner les performances et métriques propres à l'application.
Le **« monitor_level »** va quant à lui définir le niveau de surveillance des performances. Ces niveaux vont du moins détaillé (0) au plus détaillé (3). Et, le **« enable_logging »** va permettre d'activer la journalisation des activités du dashboard, comme par exemple l'enregistrement des erreurs ou des avertissements.
A la place de ces 3 lignes de paramétrages, on peut également créer un fichier, ici « config.cfg » qui va reprendre un certains nombres de paramètres, comme le niveau de monitoring pour la gestion des endpoints ou encore les identifiants de connexion.

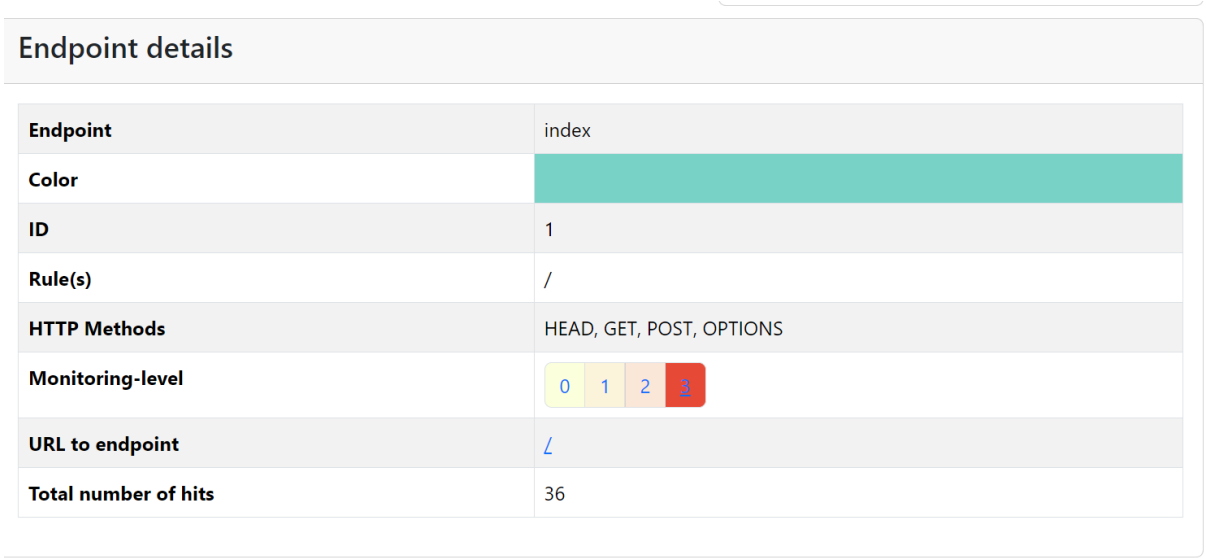
Pour lancer le dashboard, et après avoir au préalable lancé l'application Flask, il suffit d'ajouter /dashboard à l'url de l'application Flask :



Cette page présente un aperçu global des performances de l'application Flask.

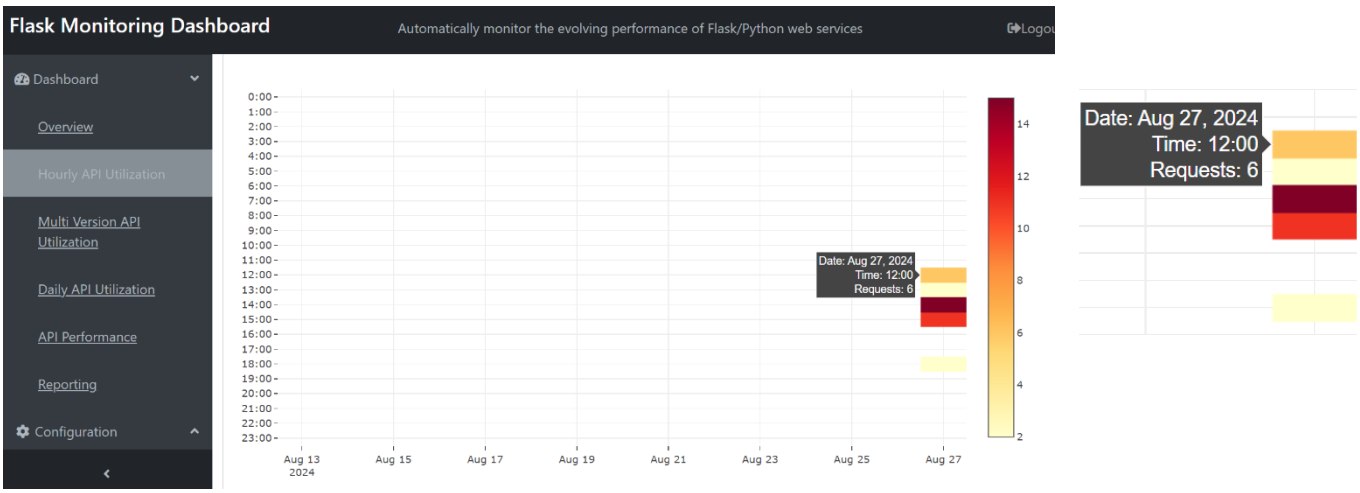
- Le « endpoint » fait ici référence à ma route principale, à savoir « index »
- On peut également voir les différentes requêtes effectuées au global, du jour ou sur une semaine. Ici par exemple, 36 requêtes ont été effectuées
- On peut aussi voir l'heure de la dernière requête
- Le niveau de monitoring, expliqué précédemment

Si l'on clique sur l'index, on est renvoyée sur une page de détails identiques avec en plus l'url de l'endpoint et les différentes méthodes http :

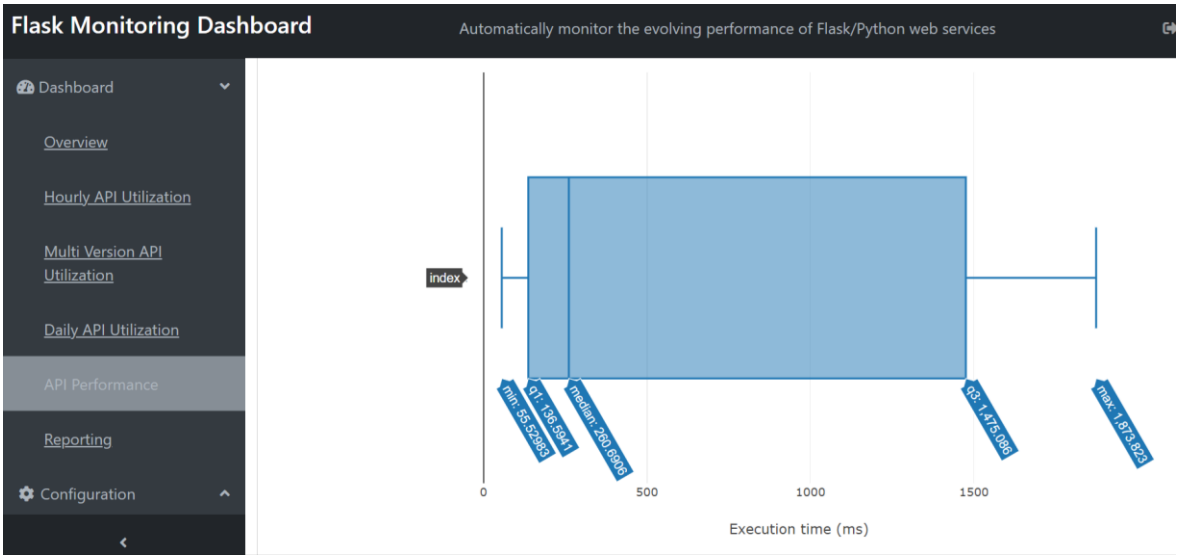


Sur le menu de gauche, en fonction de la section choisie, on va pouvoir voir plus en détails les différentes sections de monitoring.

La section **Hourly API Utilization** va par exemple retourner l'utilisation de l'API, par heure. **Cela peut aider à detecter les pics de charge au cours d'une journée** :



La section **API Performance** va quant à elle offrir des détails sur les performances des différents endpoints de l'API. On va pouvoir identifier par exemple le temps de réponse, le nombre de requêtes réussies ou au contraires échouées. **Cela peut aider à débugger et optimiser.**



III. LE MODULE LOGGING DE PYTHON

Le module logging python fournit un système de journalisation robuste qui va recenser les différents évènements survenant lors de l'exécution d'une application :

- Il va permettre de différencier les évènements par niveau de sécurité : INFO / WARNING / ERROR/ etc.
- Les gestionnaires, appelés *handlers*, vont permettre de récupérer les logs soit dans la console, soit dans un fichier externe, etc.

Le tout, dans le but d'aider à diagnostiquer les problèmes et mieux comprendre le comportement de l'application

Ici, le fichier **app.py** a été configuré comme suit :

```
# Configuration du module Logging
app.logger.setLevel(logging.DEBUG)
handler = logging.FileHandler('app.log')
formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
handler.setFormatter(formatter)
app.logger.addHandler(handler)
```

- Je commence par **définir le niveau de log de l'application**, ici configuré sur « DEBUG ». Cela signifie que tous les messages niveau DEBUG et supérieur (à savoir INFO, WARNING, ERROR et CRITICAL) vont être capturés. On s'assure ainsi que tous les évènements sont enregistrés.
- Je créé ensuite un **gestionnaire de fichiers** qui va renvoyer les logs dans ***app.log***
- Je paramètre ensuite la façon dont je veux que soient renvoyés mes logs, par le biais du *formatter :* **date / heure / niveau de log et le message**

```
# Pour la gestion des erreurs avec Logging Python
@app.before_request
def before_request_logging():
    app.logger.debug(f"Détails de la requête - Méthode : {request.method}, Chemin : {request.path}")  # noqa


@app.errorhandler(Exception)
def handle_exception(e):
    app.logger.error(f"Exception non gérée: {e}", exc_info=True)
    return render_template("error.html"), 500
```

- Ici, j'ai ajouté un décorateur qui va permettre d'enregistrer les détails de chaque requête entrante avant qu'elle ne soit traitée.

- Le second va permettre de gérer toutes les exceptions qui ne sont pas gérées ailleurs dans l'application

Enfin, j'ai ajouté dans le code des actions permettant d'enregistrer les logs et renvoyer des messages explicites en fonction du succès ou de l'échec des évènements. **Cela va améliorer la traçabilité des actions et en faciliter la compréhension ce qui va permettre une gestion plus efficace des erreurs et une meilleure maintenance**.

```python
# Puis, dans certaines parties du code, ajout de gestions des logs
try:
    data_retriever = GetData(url="https://data.rennesmetropole.fr/api/explore/v2.1/catalog/datasets/etat-du-trafic-en-temps-reel/
    # Ajout de la méthode call, sinon renvoi d'erreur : "TypeError: 'GetData' object is not callable"  # noqa
    data = data_retriever.call()
    app.logger.debug("Données récupérées avec succès.")
except Exception as e:
    app.logger.warning(f"Un problème est survenu lors de la récupération des données : {e}")  # noqa

# Ajout d"un try/except au cas où le modèle ne chargerait pas
try:
    app.logger.info('Essai du chargement du modèle...')
    model = load_model('model.h5')
    app.logger.info('Le modèle a été chargé avec succès')
except Exception as e:
    app.logger.error(f"Erreur lors du chargement du modèle : {e}")
    model = None
```

Le fichier app.log va quant à lui renvoyer ce type d'information :

```
74  2024-08-27 15:47:24,308 - DEBUG - Prédiction effectuée pour l'heure sélectionnée : 2
75  2024-08-27 15:52:46,942 - DEBUG - Prédiction effectuée pour l'heure sélectionnée : 4
76  2024-08-27 17:22:45,362 - DEBUG - Données récupérées avec succès.
77  2024-08-27 17:22:45,363 - INFO - Essai du chargement du modèle...
78  2024-08-27 17:22:45,541 - INFO - Le modèle a été chargé avec succès
79  2024-08-27 17:55:35,474 - DEBUG - Données récupérées avec succès.
80  2024-08-27 17:55:35,475 - INFO - Essai du chargement du modèle...
81  2024-08-27 17:55:35,610 - INFO - Le modèle a été chargé avec succès
82  2024-08-27 18:31:32,231 - DEBUG - Données récupérées avec succès.
83  2024-08-27 18:31:32,231 - INFO - Essai du chargement du modèle...
84  2024-08-27 18:31:32,376 - INFO - Le modèle a été chargé avec succès
85  2024-08-27 18:31:40,928 - DEBUG - Données récupérées avec succès.
86  2024-08-27 18:31:40,928 - INFO - Essai du chargement du modèle...
87  2024-08-27 18:31:41,053 - INFO - Le modèle a été chargé avec succès
88  2024-08-27 18:32:21,760 - DEBUG - Prédiction effectuée pour l'heure sélectionnée : 6
89
```