SEPTEMBRE 2023-MAI 2025

CERTIFICATION DÉVELOPPEUR IA
CAROLINE ZUMBIEHL

RAPPORT E5

SIMPLON.CO

25 ans CCTIME GROUP

DEBUGGER ET MONITORER UNE APP PYTHON

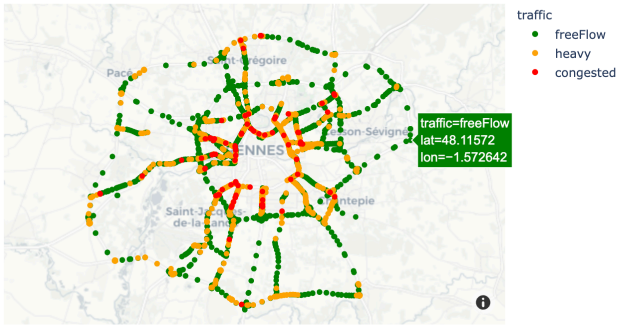
COMPÉTENCES VISÉES

- Surveiller une application d'intelligence artificielle, en mobilisant des techniques de monitorage et de journalisation, dans le respect des normes de gestion des données personnelles en vigueur, afin d'alimenter la feedback loop dans une approche MLOps, et de permettre la détection automatique d'incidents.
- Résoudre les incidents techniques en apportant les modifications nécessaires au code de l'application et en documentant les solutions pour en garantir le fonctionnement opérationnel.

PRESENTATION DU CAS PRATIQUE

Traffic Rennes

Traffic en temps réel



Prédiction d'embouteillage pour le centre ville

Choisissez une heure :

0h Envoyer

Prédiction : Libre

Rennes Metropole dispose d'une application de surveillance du traffic, pour :

- **avoir en temps réel l'état du traffic** routier de l'agglomération de Rennes par appel d'une API (voir ci dessous). Ces données sont ensuite affichées sur une **carte interactive** sous forme de points (représentants des lieux) avec différentes couleurs pour représenter l'état du trafic (vert pour fluide, orange pour dense et rouge pour bloqué)
- obtenir des **prédictions sur les conditions de trafic** basées sur un modèle de machine learning à une heure précise sélectionné par l'utilisateur via un formulaire. La réduction est ensuite affichée à l'utilisateur sous forme d'un texte coloré.

Cette application web utilise le framework Flask. Elle contient 4 fichiers principaux :

- **Index.html** est une page html contenant 2 éléments :
  - un **formulaire** de prédiction d'embouteillage dans le centre ville de Rennes. Ce formulaire prend une heure et renvoie une prédiction en fonction de l'heure. **hour** est envoyé à app.py.
  - Une **carte** montrant le trafic actuel
  - Index.html récupère de app.py 3 variables : **color_pred, text_pred, graph_json**
- **app.py** : fichier principal qui configure les toutes et la logique de traitement des requêtes et envoie des données au front. Elle gère 2 types de requêtes :
  - Les requêtes GET qui correspondent à l'envoi ou la mise à jour de la carte interactive par appel à l'api de rennes métropole lorsque l'utilisateur se connecte pour la première fois ou lorsqu'il met à jour sa page.
  - Les requêtes POST qui correspondent au cas où l'utilisateur sélectionne une heure et clique sur « envoyer ». La requête post prend l'heure sélectionnée par l'utilisateur, la passe au modèle de machine learning pour générer une prédiction sur l'état du trafic à cette heure là puis affiche cette prédiction avec une texte coloré à côté de la carte.
- **get_data.py** : qui permet la récupération des données de trafic depuis l'API rennes métropole
- **utils.py** : qui sert à la création des graphiques de données actuelles et aux prédictions du modèle pour les heures demandées.

Résultat de l'appel API de rennesmetropole.fr —> trafic en temps réel

https://data.rennesmetropole.fr/api/explore/v2.1/catalog/datasets/etat-du-trafic-en-temps-reel/exports/json?lang=fr&timezone=Europe%2FBerlin&use_labels=true&delimiter=%3B

Donne un Json qui contient 2858 points avec différentes données (vitesse, statut du trafic, temps de trajet…) listées ci dessous à l'heure précise à laquelle j'ai fait la requête.

DISPOSITIF DE MONITORAGE APPLICATION

On veut s'assurer que l'application fonctionne correctement que ce soit en phase de développement ou en phase de production.

PRINCIPALES FONCTIONS

Le monitorage application en temps réel inclut

- la détection des erreurs
- le suivi de l'utilisation des ressources (par exemple la mémoire et le CPU)
- et l'analyse des performances.

OUTILS UTILISÉS

- Module **flask_monitoring dashboard** intégré à Flask. Va permettre de suivre les performances de l'application comme le temps de réponse des requêtes ou l'utilisation des ressources
- Module **logging,** module standard de python qui permet d'enregistrer des évènement dans l'application y compris les erreurs. Ceci permet de traquer les erreurs et comprendre ce qui se passe au sein de l'application.

Seuils d'alerte et méthodes pour détecter automatiquement les incidents : Lorsque les seuils fixés sont dépassés, cela indique un problème potentiel. Ces seuils sont mis en place automatiquement dans flask_monitoringdashboard. Il ne semble pas possible de les personnaliser. Il serait cependant possible de développer de petits scripts pour suivre le nombre d'erreur par exemple dans le fichier des logs :

- **Le temps de réponse des requêtes HTTP** : par exemple si une requête prend plus de 500ms, il peut y avoir un soucis de performance.
- **Le taux d'erreurs HTTP** : une augmentation des réponses 5 xxx (erreurs server) est un signe de problème
- **L'utilisation de la mémoire** : si l'application consomme plus de mémoire que prévu, cela peut être un indicateur d'inefficacité ou de pertes de mémoire.

JOURNALISATION DES ERREURS AVEC HORODATAGE AVEC LOGGING

Etape 1 : Configuration de base de logging, ajoutée au début des fichiers. Après import de la librairie, on configure le nom du fichier de log où seront enregistrés les évènements, la précision que seules les erreurs sont suivis et enfin le format de l'enregistrement des messages.

```
import logging
logging.basicConfig(
    filename='app.log',
    level=logging.ERROR,
    format='%(asctime)s %(levelname)s %(name)s %(message)s')
logger = logging.getLogger(__name__)
```

Etape 2, utilisation de logging pour capturer les erreurs dans différents fichiers try/except. Ajout de messages qui apparaissent dans le fichier des logs.

```
@app.route('/', methods=['GET', 'POST'])
def index():
    try:
        if request.method == 'POST':
            …
        else:
            …
    except Exception as e:
        logger.error(f"Erreur dans la route principale : {e}")
        return "Une erreur est survenue", 500
```

Etape 3, revue du journal des log pour y visualiser la connection ou les erreurs. Fichier app.log.

```
2024-08-27 09:03:26,449 INFO werkzeug [31m[1mWARNING: This
is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.[0m
 * Running on http://127.0.0.1:5000
2024-08-27 09:03:26,463 INFO werkzeug [33mPress CTRL+C to
quit[0m
2024-08-27 09:06:41,656 INFO werkzeug 127.0.0.1 - - [27/Aug/2024
09:06:41] "GET / HTTP/1.1" 200 -
2024-08-27 09:06:44,134 INFO werkzeug 127.0.0.1 - - [27/Aug/2024
09:06:44] "[33mGET /favicon.ico HTTP/1.1[0m" 404 -
2024-08-27 09:19:13,779 ERROR __main__ Erreur dans la route
principale : Exception encountered when calling Sequential.call
().

[1mInput 0 of layer "dense_93" is incompatible with the layer:
expected axis -1 of input shape to have value 24, but received
input with shape (1, 25)[0m

Arguments received by Sequential.call():
  • inputs=tf.Tensor(shape=(1, 25), dtype=int64)
  • training=False
  • mask=None
2024-08-27 09:19:13,781 INFO werkzeug 127.0.0.1 - - [27/Aug/2024
09:19:13] "[35m[1mPOST / HTTP/1.1[0m" 500 -
```

SUIVI DE LA PERFORMANCE

Etape 1: Installation du suivi de la performance à la fin de l'app juste avant if __name__==« __main__ » et pip install flask_monitoringdashboard

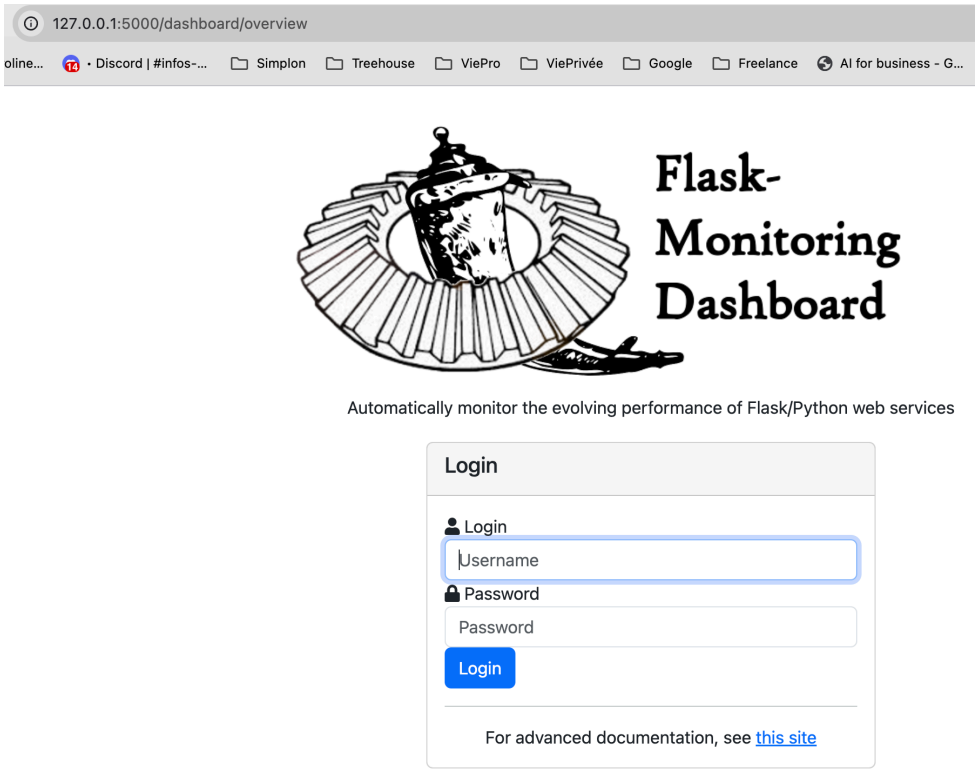
```
import flask_monitoringdashboard as dashboard
# suivi de la performance
dashboard.bind(app)
dashboard.config.monitor_level = 3 # high level
```

Cela conduit à la création d'une base de données flask_monitoringdashboard.db à la racine du projet

Etape 2: utilisation du dashboard : http://127.0.0.1:5000/dashboard/overview

Mot de passe admin, username admin.

On les changerait en production.



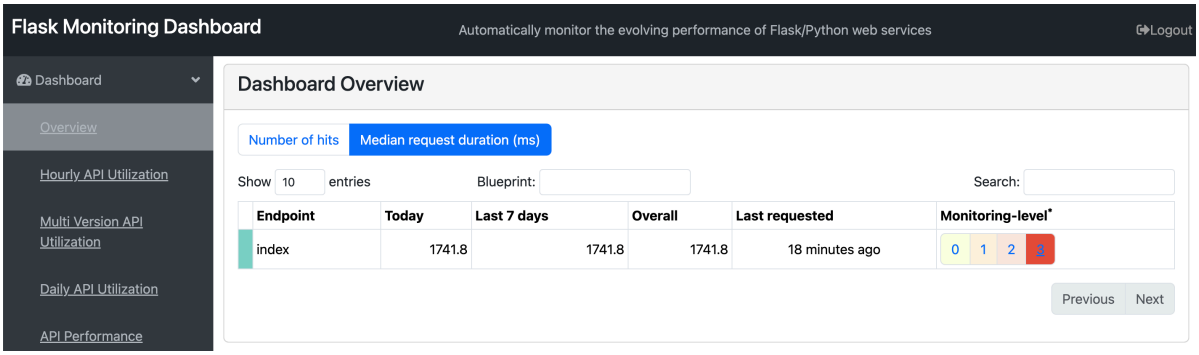
Etape 3 : faire en sorte d'avoir des données dans le dashboard,

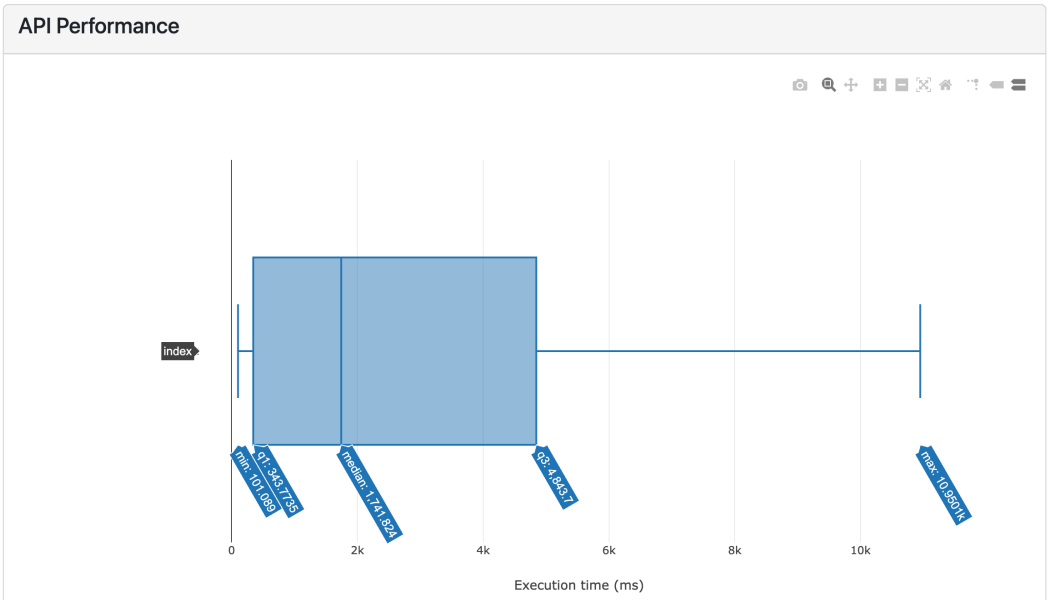
- en générant plus requêtes sur la route /,
- en ajustant la configuration pour passer au niveau le plus haut
- En allant regarder la base de données générée : dans le terminal : sqlite3 flask_monitoringdashboard.db puis SELECT * FROM request;

```
(env) MacBook-Pro-de-Caroline:rennes_traffic_ko-main carolinezumbiehl$ sqlite3 flask_monitoringdashboard.db
SQLite version 3.39.5 2022-10-14 20:58:05
Enter ".help" for usage hints.
sqlite> SELECT * FROM request;
1|1|10950.097322464|2024-08-27 08:29:59.272905|1.0|None|127.0.0.1|200
2|1|1035.80594062805|2024-08-27 08:29:59.282022|1.0|None|127.0.0.1|200
3|1|5423.20203781128|2024-08-27 08:29:59.286561|1.0|None|127.0.0.1|200
4|1|3105.19504547119|2024-08-27 08:30:07.797594|1.0|None|127.0.0.1|200
5|1|1741.82415008545|2024-08-27 08:30:07.799221|1.0|None|127.0.0.1|200
6|1|113.095998764038|2024-08-27 08:30:15.058867|1.0|None|127.0.0.1|200
7|1|101.089000701904|2024-08-27 08:30:18.666349|1.0|None|127.0.0.1|200
sqlite>
```

Etape 4 : exploiter le dashboard

L'**overview** donne un résumé du nombre et de la durée des requêtes récentes (execution time, daily API utilization, hourly API utilization). En cliquant sur notre **endpoint index**, on obtient d'autres indicateurs plus détaillés (status code distribution, outliers…)





DEBUGGAGE ET GESTION DES ERREURS

On veille à ce que l'application soit lancée en mode debug=True lors du développement de l'application.

Ensuite on lance l'application : python3 rennes_traffic_ko-main/app.py

Présenter la description de l'incident (ou des incidents si il y en a plusieurs), la méthode de résolution et le périmètre impacté.

Présenter la(les) solution(s) apportée(s), en expliquant la méthodologie, votre démarche pour comprendre la panne, les tests que vous avez effectués.

ERREURS DE SYNTAXE

Oubli des parenthèse pour la méthode to_json() ligne 57 de app.py

```
fig_map = create_figure(data)
graph_json = fig_map.to_json
```

Oubli de **fermer crochets** dans get_data()

```
res_df = res_df[res_df.traffic != 'unknown'
```

Erreurs d'**indentation** :

```
  File "/Volumes/Expansion/AAA/debug-monitor-ai/rennes_traffic_ko-main/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
```

```
        for data_dict in self.data:
        temp_df = self.processing_one_point(data_dict)
        res_df = pd.concat([res_df, temp_df])
```

Il faut indenter les 2 lignes.

Oubli d'une virgule dans utils.py

```
       zoom=10
        height=500,
```

Erreur dans le nom d'une propriété du json. Il s'agit de trafficstatus sans underscore.

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

ERREURS DE CONCEPTION

Ajout de **requirements.txt** et des packages nécessaires au fonctionnement de l'application : tensorflow, pandas, plotly, flask.

Oubli d'**appeler le paramètre hour** dans la fonction prediction_from_model() dans app.py. Cette fonction est définie dans utils.py

```
cat_predict = prediction_from_model(model)
```

Erreur dans le nom des propriétés du json qui sont **lat** et **lon** et non lattitude et longitude.

```
temp['lat'] = temp.geo_point_2d.map(lambda x: x['lattitude'])
temp['lon'] = temp.geo_point_2d.map(lambda x: x['longitude'])
```

Erreur dans le template rendu : initialement c'est home.html alors que le nom du template est index.html.

Retirer les packages qui ne servent à rien dans app.py car jamais appelés dans app.py

```
import plotly.graph_objs as go
import plotly.express as px
import numpy as np
```

Changement de la taille du vecteur à 24 et non 25 dans utils.py :

```
def prediction_from_model(model, hour_to_predict):
    input_pred = np.array([0]*25)
```

SOURCES

Documentation flask Monitoring Dashboard :
https://flask-monitoringdashboard.readthedocs.io/en/latest/index.html

Documentation logging de python:
https://docs.python.org/fr/3/library/logging.html
https://docs.python.org/fr/3/howto/logging.html

Repos sources:
https://github.com/bertrandfournel/rennes_traffic_ko/

ANNEXE - AUTRE EXEMPLE - VISION CAR POC

PRINCIPE

Ce programme est une application web construite avec Flask, qui permet à un utilisateur de téléverser une image. L'image est ensuite traitée par un modèle de deep learning pour produire une "masque de prédiction". Ce masque est une image qui identifie certaines caractéristiques ou segments de l'image d'origine. L'application renvoie à l'utilisateur l'image originale et le masque prédictif, les deux encodés en base64 pour pouvoir les afficher sur une page web

L'image est en mode RGB

DEBUG / MONITORING

Mise en place logging et journalisation de tous les évènements —> fichier app.log. Ajout de try / except et de log dans le code.

```
17  2024-08-27 17:36:58,642 - INFO - Prediction successful for image:
    jacek-dylag-PMxT0XtQ--A-unsplash.jpg
18  2024-08-27 17:36:58,985 - INFO - 127.0.0.1 - - [27/Aug/2024 17:36:58]
    "POST /upload HTTP/1.1" 200 -
19  2024-08-27 17:40:04,633 - INFO - 127.0.0.1 - - [27/Aug/2024 17:40:04]
    "GET / HTTP/1.1" 200 -
```

Correction des erreurs :

- Ajout des packages et requirements.txt
- appel de home.html dans render template au lieu de index.html
- Erreur indentation if image.filename ==‘’: et bloc de code suivant
- Erreur dans le nom de répertoire nommé template alors que flask recherche templates
- Changement de l’appel du modèle

```
import tensorflow as tf
model = tf.keras.models.load_model(
       "models/unet_vgg16_categorical_crossentropy_raw_data.keras",
       compile=False)
```

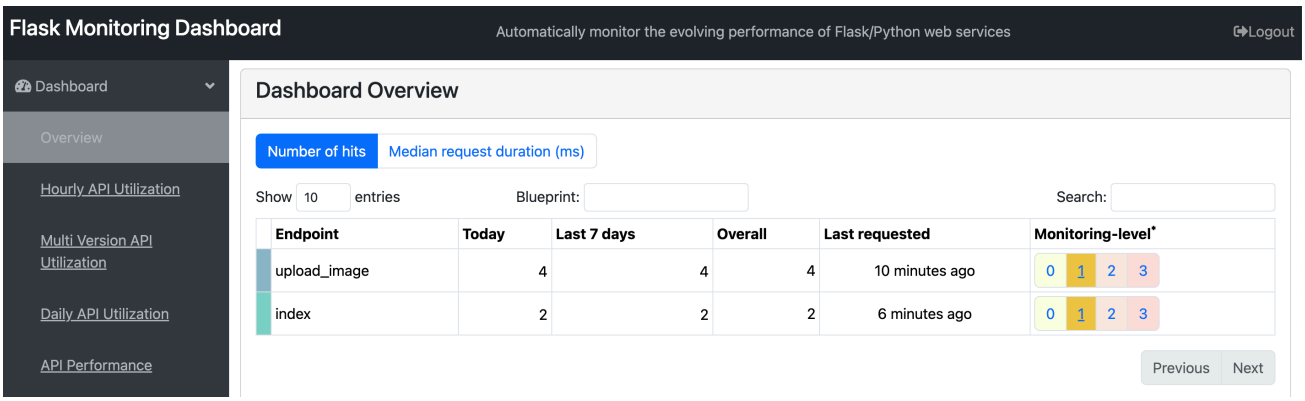
- Erreur dans la taille de l’image (IMAGE_SIZE) 512 corrigé en 256
- Correction **return jsonify(…)** au lieu de **json(…)**. Json est un module non une fonction (possible json.dumps() pour convertir en json). Mais Jsonify est une fonction flask qui permet de convertir des objets python (dictionnaires listes en JSON en créant une réponse HTTP correctement formatée avec le bon en tête MIME (Content-type:application/json) et en gérant des aspects de sécurité comment l’encodage des caractères spéciaux pour éviter les injections. C ‘est bien ce qu’on veut ici.
- Ajout de la ligne de **conversion de l’image en binaire** car elle était manquante

```
buffered_mask = BytesIO()  # création object BytesIO binaire
predict_mask.save(buffered_mask, format="PNG") # stokage image en binaire
base64_mask = base64.b64encode(buffered_mask.getvalue()).decode("utf-8")
                                    # conv binaire à texte (base64)
```

- Conversion RGB si nécessaire

```
if img.mode == 'RGBA':
                img = img.convert('RGB')
```

Performance, mise en place de flask_monitoringdashboard —> route /dashboard



ANNEXE TRAITEMENT DES IMAGES

Une image est une grille de données binaires : une grille de pixels, chaque pixel étant représenté dans le cas RGB par 3 valeurs qui sont encodées en binaire. Le pixel (255, 0, 0) est donc encodé en 11111111 00000000, 00000000.

Pillow (PIL) bibliothèque python pour le traitement des images. Ici permet d'ouvrir l'image.

Numpy : pour la manipulation de tableaux multidimensionnels (ici ce sont les données d'images qui seront manipulées)

Base 64 : module pour encoder et décoder des données en base64, ce qui permet de transmettre des données binaires (composées de 0 et 1) comme les images ou des fichiers sous forme de texte ou chaines de caractères. Des images encodés en base64 peuvent être envoyées au navigateur sous forme de JSON et le navigateur peut alors les afficher directement.

BytesIO : programme utilisé pour convertir les images en données binaires puis ces images sont encodées en base 64