Compétences C20 & C21 - Py Traducteur

* C.20 : Surveiller une application d’intelligence artificielle
* C.21 : Résoudre les incidents techniques

## Présentation de l'application

L’application est une interface Streamlit offrant l’accès à un transformer Huggingface permettant de traduire du texte, servie via API FastAPI.

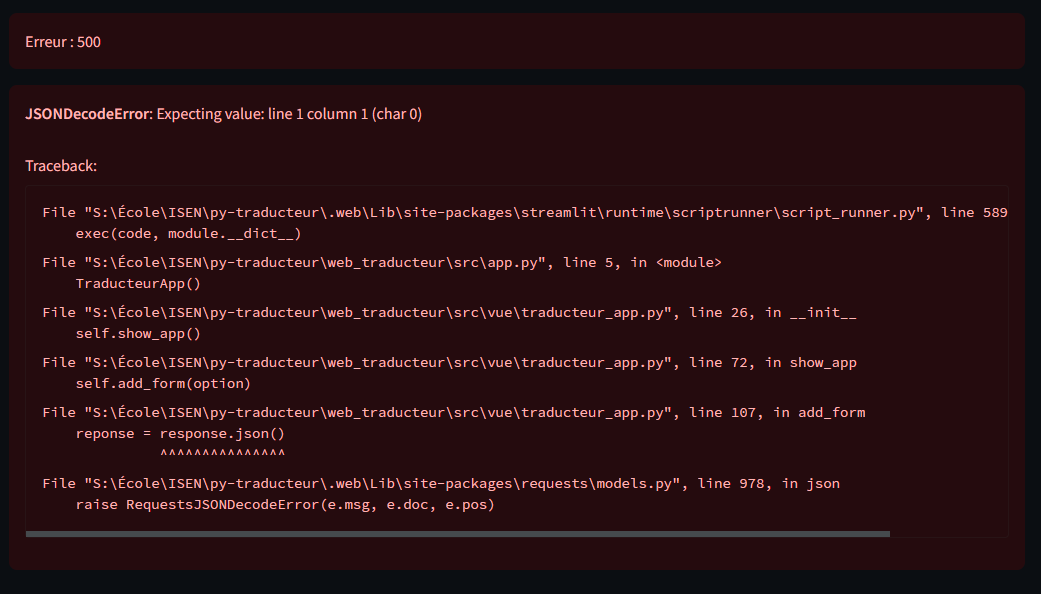
## Présentation de l'incident technique

J’ai pu me connecter avec succès grâce aux identifiants du fichier d’initialisation de la bdd. Les identifiants sont les suivants :   
ID  : Cleese  
MDP  : Sacré Graal!

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquementJ’accède à un historique de traduction :

J’essaie d’effectuer une traduction, mais j’obtiens une erreur 500, correspondant à ‘Internal Server Error’, indiquant une erreur dans l’API.



**Présentez le message d'erreur en console et expliquez-le.**

Une image contenant texte, capture d’écran, menu, Police

Description générée automatiquement

D’après le message d’erreur, on essaie de convertir une liste en valeurs à insérer dans mysql (la base de données), mais la conversion ne peut être effectuée, laissant penser que la liste possède un problème de valeur.

## Expliquez les recherches faites pour résoudre l'incident technique.

J’ai donc choisi d’afficher exactement ce à quoi ressemble la donnée avant de rentrer dans la fonction qui insère en base de données.   
Pour effectuer cela, j’ai inséré ce code juste avant l’enregistrement en BDD :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

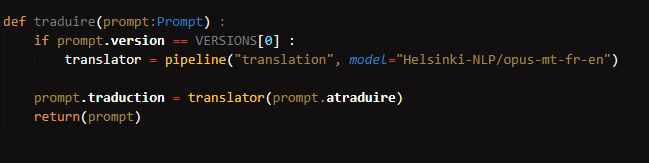
J’obtiens ce résultat :

*atraduire='Test de traduction' traduction=[{'translation\_text': 'Translation test'}] version='fr >> en' utilisateur=1*

On remarque que traduction affiche actuellement le dictionnaire complet au lieu de seulement la valeur traduite. Il faut donc aller à la sortie du modèle et extraire l’information du dictionnaire.

Il faut aussi faire attention à la sortie sur le serveur streamlit, étant donné que la donnée est bien traitée et est censée s’afficher correctement sur le site si on se fie au ticket.

## Expliquez la correction apportée et le test de validation

Je localise donc la fonction qui utilise le modèle : 

Et j’extrais l’information qui m’intéresse, donc d’après l’extraction que j’avais faite, elle se trouve dans une liste (visible grâce aux []) puis dans un dictionnaire sous la clé ‘translation\_text’.

Une image contenant texte, capture d’écran, Police

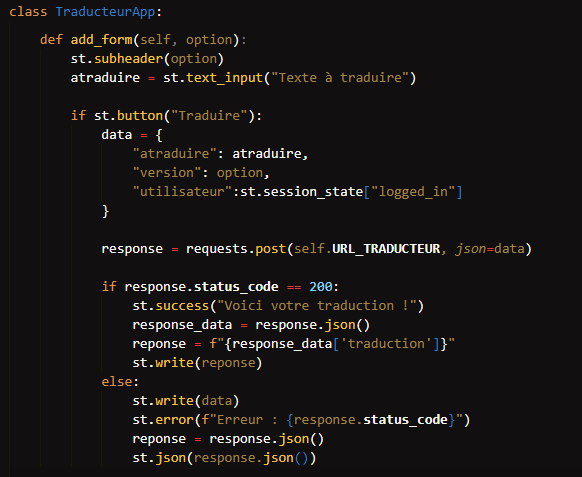
Description générée automatiquement

Je me rends également dans le code de l’appli web, et je localise en effet une extraction similaire à ce que je viens d’effectuer dans la fonction add\_form : Une image contenant texte, capture d’écran, Police

Description générée automatiquement



Étant donné que cette transformation est maintenant effectuée à la source, je supprime le surplus ici :



Lors de la correction de l’erreur précédente, je me rends compte que la traduction pour l’autre version (en >> fr) n’est pas fonctionnelle car la condition ne vérifie que pour fr >> en (qui est VERSIONS[0])

J’ai donc rajouté cette fonctionnalité pour notre 2ème option, qui est VERSIONS[1] :

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

Après test, l’application marche bel est bien comme on le souhaite :   
Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

Description générée automatiquement

Et bien sûr, le côté anglais fonctionne également : Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

## Expliquez le versionnage de la correction dans Git et le déploiement sur GitHub.

Après avoir effectué les corrections nécessaires dans le code, j'ai utilisé Git pour versionner ces modifications.

J'ai d'abord vérifié les changements apportés aux fichiers avec la commande :

git status

Les fichiers modifiés étaient listés. J'ai ajouté ces fichiers à l'index pour les préparer au commit :

git add -A

Ensuite, j'ai créé un commit avec un message détaillé décrivant les corrections effectuées :

git commit -m "Correction de l'erreur 500 lors de la traduction et ajustement de la condition pour la traduction en >> fr"

Puis j’ai effectué le push final vers git afin de déployer les corrections :

git push origin main

Ainsi, les corrections ont été versionnées et déployées, assurant la mise à jour de l'application avec les nouvelles modifications.

## Ajoutez la documentation sur le dashboard en expliquant le choix des métriques, le choix de la technologie, la mise à jour des indicateurs et les alertes.

Pour monitorer notre application Py-Traducteur, j'ai choisi d'utiliser Grafana et Prometheus. Ces deux outils open-source sont largement adoptés pour la surveillance des applications et offrent une intégration fluide avec les services conteneurisés via Docker Compose.

* **Prometheus** est utilisé pour collecter et stocker les métriques de performance de l'application. Il scrute régulièrement les endpoints exposés pour recueillir les données.
* **Grafana** permet de visualiser ces métriques de manière interactive. Il offre des tableaux de bord personnalisables pour analyser les données en temps réel.

Voici les métriques que j'ai mises en place :

**Temps de réponse des requêtes (duration)**

* + **Pourquoi** : Mesurer le temps que met chaque requête à être traitée permet d'identifier les ralentissements et les points de congestion dans l'application.
  + **Comment** : Enregistré via le middleware log\_request\_time qui calcule la durée entre le début et la fin du traitement de chaque requête.

**Nombre d'erreurs HTTP par code de statut (http\_errors)**

**Pourquoi** : Suivre le nombre d'erreurs permet de détecter rapidement les problèmes côté serveur ou client.

* + **Comment** : Utilisation du compteur http\_errors avec des labels pour chaque code de statut HTTP.

**Nombre de requêtes par endpoint et méthode**

* + **Pourquoi** : Identifier quels endpoints sont les plus sollicités et optimiser leur performance si nécessaire.
  + **Comment** : Stocké grâce à la fonction enregistrer\_metric qui enregistre l'endpoint, la méthode HTTP, la durée et le code de statut.

**Mise à jour des indicateurs**

Une image contenant texte, capture d’écran, Police

Description générée automatiquementLes indicateurs sont mis à jour en temps réel grâce au middleware intégré dans l'application FastAPI :

Ce code permet de mesurer le temps de traitement de chaque requête, incrémenter le compteur d'erreurs HTTP en fonction du code de statut et enregistrer les métriques pertinentes choisies pour chaque requête.

**Configuration des alertes**

Des alertes ont été configurées pour assurer une surveillance proactive :

* **Seuils de temps de réponse** : Si le temps de réponse moyen dépasse un certain seuil, une alerte est déclenchée pour investiguer un possible problème de performance. Je l’ai configuré pour envoyer une alerte dès qu’il détecte un délai de réponse d’au moins 5 secondes dans un premier temps, pour effectuer des tests.

Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

Description générée automatiquement

* Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

  Description générée automatiquement**Taux d'erreurs élevé** : Un nombre anormalement élevé d'erreurs HTTP déclenche une alerte, indiquant un problème potentiel.

Une image contenant capture d’écran, ligne

Description générée automatiquementPrometheus récupère bien les codes http :

Une image contenant texte, logiciel, Logiciel multimédia, Système d’exploitation

Description générée automatiquementEt je reçois bien les alertes sur Discord, le moyen que j’ai choisi pour recevoir les alertes :

**Intégration via Docker Compose**

Le fichier docker-compose.yml permet de déployer l'ensemble des services nécessaires :

* **Base de données (db)** : Conteneur MySQL pour stocker les données de l'application.
* **Adminer** : Interface web pour gérer la base de données.
* **FastAPI** : Le serveur d'application qui héberge l'API de traduction.
* **Grafana** : Pour la visualisation des métriques.
* **Prometheus** : Pour la collecte et le stockage des métriques.

## Conclusion

L'utilisation combinée de Grafana et Prometheus nous offre une solution robuste et facile à implémenter pour le monitoring de l'application Py-Traducteur.

Les métriques choisies fournissent une vue d'ensemble de la performance et de la fiabilité du service.

Les indicateurs sont mis à jour en temps réel, et les alertes assurent une réactivité en cas de problèmes !