Démarche

1. **Accéder à la base de donnée et aux fichiers JSon**
2. **Vérification avec le format Json de la consigne**
    - En relevant les erreurs.
3. **Comparaison entrée sortie**
    - En relevant les erreurs.
4. **Synthétiser les résultats de test**
5. **Dans un autre temps soumettre de nouvelles bases de données “extrêmes” pour connaître et/ou tester les limites du composant logiciel**

Ce que j’ai fait avant le 10/04

- Lecture de la base de données
- Lecture des fichiers Json en sortie du composant logiciel
- Comparaison du format entre les fichiers JSON et la consigne
    - nom
    - type
- Tri des fichiers Json en fonction de leur Timestamp
- *Tri des fichiers Json en fonction de leur ID (révélé non pertinent)*
- Comparaison des ID des fichiers Json :
    - incrémentation de 1 à chaque nouveau fichier
    - répétition
    - manquement
- Comparaison des Timestamp entre la BD et les Json
- Encodage en base64 des messages de la BD
- *Décodage du contenu des fichiers Json (révélé non pertinent)*
- Comparaison du contenu entre le BD et les Json
- Comparaison du contact des messages entre la BD et les Json
    - pour ce faire : décodage de l’ID des contacts de la BD
- Comparaison de la direction : émetteur récepteur

Ce qu’il manque

- Vérifier si les Timestamp des Json sont dans l’ordre
- Vérifier quels messages sont correctes à 100%
- Création d’un fichier rapport de test automatique, compilant les erreurs reconnues
- Créer une base de données test avec des cas de figure extrême (taille du contenu du message, mettre plusieurs destinataires, mettre d’autres destinataires, tailles limite de timestamp, changement du format de la base de données …). Dans le but de savoir si le composant est capable de relever des erreurs de base de données et/ ou relever lui même ses incapacités
- Éprouver les capacités de traitement du composant logiciel pour tester ses limites : vitesse d’envoie de requêtes, nombre de requêtes, taille des requêtes

11/04/2025

Comment j'ai réfléchi

J'ai :

- Entrée / Sortie
- Composant logiciel qui encode
- Format consigne

Ce que je dois faire avec ce que je sais :

- m'assurer que le **format** de la consigne est appliqué dans un premier temps => partir de **l'échelle macro** pour **zoomer** dans les attentes et les sorties attendues
  - format des fichiers Json : Nom des objets (keys) & leur Type (type des values). Tout ce qui est en plus, en moins, ou différent est source d'erreur.
- vérifier la conformité entrée / sortie
  - Etant donné que les fichiers ne sont pas dans l'ordre des messages de messaging.db j'ai décidé d'orienter ma comparaison entrée/sorties en comparant chacun des noms d'objets {id,timestamp,content … } et non en comparant message par message.
  - pour chacun des objets du fichier json voir la conformité exacte de la sortie par rapport à l'entrée. Tout ce qui est en plus, en moins, ou différent est source d'erreur.

Une question m'est venue, qu'est ce que je fais si j'ai des **erreurs** à ce niveau ? **comment interpréter** le fichier json concerné ? je fais une croix dessus ?
=> non il est **pertinent** de voir s'il y a des erreurs à des niveaux d'investigation plus bas : cela peut aider pour la compréhension des erreurs pour apporter des corrections au programme, ça peut être aussi bien une cause d'erreurs comme une conséquence d'autres erreurs.

Sur quoi baser ma comparaison : ID ou timestamp ? Que prendre comme erreur ou non. Avec ce cas précis, j'ai vite choisi de baser ma comparaison par rapport au **timestamp** par un rapide coup d'œil en voyant que timestamp n'avait pas d'erreur (tous uniques et présents dans la base de données mais cependant désordonnés) malgré un message manquant.
Si j'avais été dans un autre cas, je ne sais pas concrètement comment j'aurais continué sans avoir de doute … En développant des fonctions permettant de voir quelles sont les occurrences qui permettent de baser la comparaison des messages sur l'id ou le timestamp.

Condenser les informations

Système de **flag** et de remontées d'erreurs par l'ajout d'un **item** dans le dictionnaire que sont chaque fichier json après extraction. Permet de voir quels sont les fichiers json corrompus ou non en plus d'avoir un condensé des erreurs.

Autres Réflexions

- Est ce que je **connais les limites** du composant ?
  - **oui** : tester le composant à ses limites
    - sont elles confirmées
      - **plus basse** : communiquer pour dire que les limites sont plus basses que prévu => **prise en compte indispensable**

        - **plus hautes** : **permissives** => des limites plus hautes ne vont pas diminuer ou modifier les performances et la qualité du composant

    - **non** : chercher à les obtenir et voir le comportement
        - Est-ce pertinent de les chercher ? :
            - oui : si je ne connais pas le pire cas en entrée
                - Fixer les limites observées et communiquer au développeur pour savoir si ces limites sont prises en compte pour l'utilisation du composant. Dans le but de vivre avec les limites, de les contourner ou apporter des modifications pour les repousser
            - non : si je connais la situation critique (le pire cas) alors simplement la tester

Améliorer mon ébauche de programme de test par la méthode ? :

Je prendrais le problème dans un autre sens :

- Garder le "l'ordre" de test (échelle de test, "zoomer" dans les fichiers)
- Voir si l'entièreté du message json est présent dans la base de donnée avant de tester et non l'inverse => pas certain de la pertinence
- Éprouver le programme de test avec des bases de données différentes pour voir s'il est robuste
- Automatiser le test pour l'appliquer à d'autres bases de données
- Epurer & simplifier l'algo
- Extraire les erreurs dans un rapport de test
- Éprouver les capacités de traitement du composant logiciel pour tester ses limites : vitesse d'envoie de requêtes, nombre de requêtes, taille des requêtes

Améliorer le logiciel :

- Donner un nom croissant aux fichiers json. Trouver un moyen d'ordonnancer les messages json.
- Résoudre les erreurs.