



Rapport rédigé à la suite d'un Projet effectué en :

M1 DAS

Pour :

RT0805 - Programmation répartie

Sous la supervision de

M. Flauzac Olivier , M. Massip Jean

Par

Houdaitillah Mohamed Bakri & MTARFI Souhail

Année : 2023/2024

A l'Université de Reims Champagne-Ardenne



Table des matières

I. Introduction.....	4
II. Conception du système.....	4
A. Architecture globale du système.....	4
1. Client.....	4
2. Serveur.....	5
3. Protocole de Communication.....	5
4. Base de Données.....	5
5. Tests Unitaires.....	5
6. Environnement de Déploiement.....	5
Figure 2 : Arborescence du code du projet.....	6
B. Conception du client.....	6
1. Fonctionnalités du Client.....	6
2. Organisation des Dossiers et Fichiers :	6
3. Protocole de Communication.....	7
4. Tests Unitaires.....	7
5. Intégration dans l'Environnement Docker.....	7
C. Conception du serveur.....	7
1. Organisation des Dossiers et Fichiers :	7
2. Communication gRPC.....	8
3. Gestion des Appareils.....	8
4. Intégration dans l'Environnement Docker.....	8
5. Tests Unitaires.....	8
D. Choix de la base de données.....	9
III. Développement du système.....	10
A. Implémentation du client.....	10
B. Implémentation du serveur.....	11
C. Tests unitaires et validation.....	13
1. Serveur.....	13
Figure 3 : Résultat du test pour le serveur (Couverture 87.5%).....	14
2. Client.....	14
IV. Environnement de déploiement.....	15
A. Configuration du fichier docker-compose.....	15
B. Mise en place de la base de données.....	16
C. Déploiement du système.....	18
V. Résultats et fonctionnement.....	18
A. Démonstration du fonctionnement du système.....	19
Figure 4 : Le serveur de db MongoDB.....	19

Figure 5 : Les différentes tables dans la base des données.....	19
Figure 6 : Machines stockées avec leurs informations.....	20
Figure 7 : Machines stockées avec leurs opérations.....	20
VI. Conclusion.....	20
VII. Annexes.....	21
A. Schémas de conception.....	21
Figure 6 : Schéma explicatif de l'architecture globale du projet.....	22
B. Exemples de fichiers JSON.....	22
VIII. Références.....	23
A. Sources documentaires utilisées.....	23

I. Introduction

Le présent rapport expose le projet de développement de deux programmes en langage Go, initié pour répondre aux besoins de notre entreprise. Cette dernière gère un parc d'appareils déployés chez divers clients, ces appareils générant quotidiennement des fichiers au format JSON contenant un résumé de leurs opérations. L'objectif du projet est de concevoir un système composé d'un client et d'un serveur, utilisant gRPC comme protocole de communication, afin de collecter ces données, les transmettre au serveur, les stocker dans la base de données MongoDB, et effectuer les mises à jour nécessaires.

Le client est chargé de lire les fichiers JSON générés par les appareils, tandis que le serveur est responsable de recevoir ces données via gRPC, de les stocker dans une base de données, et de gérer la création et la mise à jour des enregistrements correspondants. En outre, le système doit être en mesure de fournir des statistiques concernant le nombre total d'opérations ainsi que le nombre d'opérations échouées.

Pour assurer la qualité du code et garantir son bon fonctionnement, des tests unitaires seront élaborés, visant à obtenir une couverture minimale de 60% du code source. Enfin, le projet sera livré avec un environnement de déploiement fonctionnel, intégrant un fichier docker-compose pour la mise en place de la base de données nécessaire à l'exécution du système.

Ce rapport détaille les étapes de conception et de développement du projet, ainsi que son fonctionnement opérationnel.

II. Conception du système

A. Architecture globale du système

L'architecture globale du système proposé comprend deux composants principaux : un client et un serveur, qui communiquent entre eux via gRPC pour collecter, traiter et stocker des données provenant de fichiers JSON générés par des appareils distants. Voici une vue détaillée de cette architecture :

1. Client

- Le client est chargé de lire les fichiers JSON provenant des appareils distants.
- Il utilise gRPC pour envoyer ces données au serveur.

- Son rôle principal est de collecter les informations des fichiers JSON et d'initier la communication avec le serveur pour l'envoi de ces données.

2. Serveur

- Le serveur reçoit les données du client via gRPC.
- Il stocke ces données dans une base de données MongoDB.
- Il est responsable de la gestion des opérations de stockage et de mise à jour des enregistrements de données.
- Le serveur répond aux requêtes du client, notamment pour fournir des statistiques sur les opérations effectuées et échouées.

3. Protocole de Communication

- La communication entre le client et le serveur est basée sur **gRPC** (Remote Procedure Call) pour une transmission efficace et structurée des données.
- gRPC utilise HTTP/2 comme protocole de transport et offre une définition de service et une sérialisation des données via Protocol Buffers.

4. Base de Données

- Le système utilise MongoDB comme base de données pour stocker les données reçues des appareils.
- MongoDB est une base de données NoSQL, ce qui offre une flexibilité dans le stockage des données non structurées ou semi-structurées, comme les fichiers JSON.

5. Tests Unitaires

- Des tests unitaires seront développés pour garantir la qualité du code.
- L'objectif est d'atteindre une couverture minimale de 60% du code source, assurant ainsi une meilleure fiabilité et maintenabilité du système.

6. Environnement de Déploiement

- Le projet sera livré avec un environnement de déploiement fonctionnel.
- Un fichier docker-compose sera fourni pour faciliter le déploiement et la configuration de la base de données MongoDB nécessaire à l'exécution du système.

```
.
├── README.md
├── client
└── Dockerfile
```

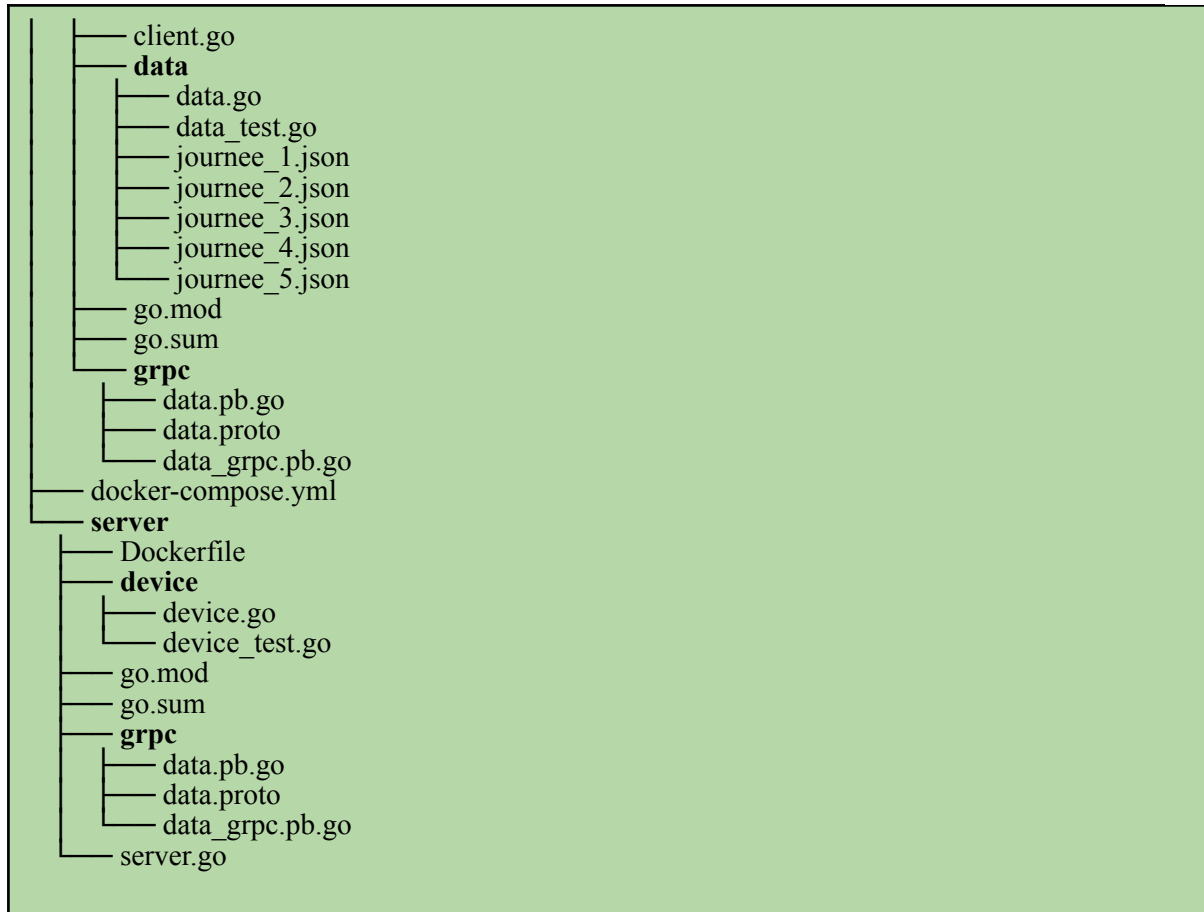


Figure 2 : Arborescence du code du projet

B. Conception du client

1. Fonctionnalités du Client

- Le client est responsable de la lecture des fichiers JSON provenant des appareils distants, qui sont stockés dans le dossier client/data.
- Il transforme les données JSON en structures de données internes utilisables par le système.
- Utilise gRPC pour établir une connexion avec le serveur et envoyer les données collectées.

2. Organisation des Dossiers et Fichiers :

- **client** : Dossier principal contenant le code du client.
 - **Dockerfile** : Fichier Docker permettant de construire une image Docker pour le client.
 - **client.go** : Implémente la logique principale du client, y compris la lecture des fichiers et l'envoi des données via gRPC.

data : Dossier contenant les fichiers nécessaires à la lecture et au traitement des données.

- **data.go** : Contient les fonctions de lecture et de traitement des fichiers JSON.
- **data_test.go** : Fichiers de tests unitaires pour le package data.
- **grpc** : Dossier contenant les fichiers nécessaires à la communication via gRPC.
 - **data.proto** : Fichier de définition de message et de service gRPC pour la communication client-serveur.
 - **data.pb.go** et **data_grpc.pb.go** : Fichiers générés à partir du fichier data.proto pour la communication gRPC.

3. Protocole de Communication

- Utilise gRPC pour établir une connexion avec le serveur.
- Le fichier data.proto définit les messages et les services nécessaires à la communication client-serveur.
- Les fichiers data.pb.go et data_grpc.pb.go sont générés à partir de data.proto et contiennent le code nécessaire pour la communication gRPC.

4. Tests Unitaires

- Le package **data** contient des tests unitaires (**data_test.go**) pour valider le bon fonctionnement de la lecture et du traitement des fichiers JSON.

5. Intégration dans l'Environnement Docker

- Le fichier Dockerfile dans le dossier client permet de construire une image Docker pour le client, facilitant ainsi le déploiement dans différents environnements.
- L'image Docker du client peut être utilisée avec le **docker-compose.yml** fourni pour déployer l'ensemble du système (client et serveur) de manière cohérente.

Cette conception du client garantit une structure modulaire, où chaque composant est responsable d'une tâche spécifique, facilitant ainsi la maintenance, l'extension et le test du client. La séparation des préoccupations entre la lecture des données, la communication via gRPC et les tests unitaires contribue à la fiabilité et à la robustesse du client dans le cadre du projet global.

C. Conception du serveur

1. Organisation des Dossiers et Fichiers :

Le serveur est structuré dans le dossier *server* de notre projet.

- *server* : Dossier principal contenant le code du serveur.
- *Dockerfile* : Fichier Docker permettant de construire une image Docker pour le serveur.
- *server.go* : Implémente la logique principale du serveur, y compris la gestion de la communication gRPC et le traitement des données reçues.
- *device* : Dossier contenant les fichiers relatifs aux appareils (peut être lié à la gestion des données ou à d'autres fonctionnalités spécifiques du serveur).
 - *device.go* : Contient les définitions et le code lié à la gestion des appareils connectés au serveur.
 - *device_test.go* : Fichiers de tests unitaires pour le package device, permettant de valider le bon fonctionnement des fonctionnalités relatives aux appareils.

2. Communication gRPC

- Le serveur utilise **gRPC** pour la communication avec le client.
- Le dossier *grpc* dans le dossier *server* contient les fichiers nécessaires à la définition et à l'implémentation des services gRPC.
 - *data.proto* : Fichier de définition de message et de service gRPC pour la communication client-serveur.
 - *data.pb.go* et *data_grpc.pb.go* : Fichiers générés à partir de data.proto pour la communication gRPC, contenant le code nécessaire pour implémenter les services définis dans le protocole.

3. Gestion des Appareils

- Le package *device* dans le dossier *server* contient la logique et les fonctionnalités spécifiques liées à la gestion des appareils.
- Le fichier *device.go* définit les structures de données et les méthodes associées pour interagir avec les appareils connectés au serveur.

4. Intégration dans l'Environnement Docker

- Le fichier *Dockerfile* dans le dossier *server* permet de construire une image Docker pour le serveur, facilitant ainsi le déploiement et la configuration dans différents environnements.
- L'image Docker du serveur peut être utilisée conjointement avec le fichier docker-compose.yml pour déployer l'ensemble du système (client et serveur) de manière cohérente.

5. Tests Unitaires

- Des tests unitaires sont développés pour valider le bon fonctionnement du serveur, en couvrant les différentes fonctionnalités telles que la gestion des appareils et la communication gRPC.
- Les fichiers de tests unitaires (device_test.go, etc.) permettent de s'assurer que les fonctionnalités du serveur sont correctement implémentées et répondent aux spécifications.

Cette conception du serveur assure une séparation claire des préoccupations, avec des modules distincts pour la communication gRPC, la gestion des appareils et la logique principale du serveur. Cette organisation favorise la maintenabilité, l'extensibilité et la fiabilité du serveur dans le contexte du projet global, en s'appuyant sur des pratiques de développement robustes et testées.

D. Choix de la base de données

Pour le projet de développement du système client-serveur, le choix de la base de données doit prendre en compte les besoins spécifiques en termes de stockage, de performances et de facilité d'intégration avec l'architecture globale du système. Dans ce contexte, le choix de MongoDB est justifié pour les raisons suivantes :

❖ Flexibilité du Schéma :

MongoDB est une base de données NoSQL qui offre une flexibilité du schéma, ce qui est particulièrement avantageux pour le stockage de données semi-structurées telles que les fichiers JSON générés par les appareils distants.

Cette flexibilité permet de stocker des données variées sans avoir à définir un schéma rigide à l'avance, ce qui correspond bien aux besoins du projet où les données peuvent être évolutives et hétérogènes.

❖ Gestion de Données JSON :

MongoDB est nativement conçu pour gérer des documents au format JSON, ce qui simplifie le stockage, la recherche et la manipulation des données provenant des fichiers JSON générés par les appareils distants.

Les opérations de lecture et d'écriture sur des données JSON sont optimisées dans MongoDB, offrant ainsi des performances adaptées aux besoins du système.

❖ Scalabilité Horizontale :

MongoDB prend en charge la scalabilité horizontale, ce qui permet de faire évoluer facilement la capacité de stockage et les performances du système en ajoutant simplement de nouveaux nœuds au cluster MongoDB.

Cette capacité de scalabilité est essentielle pour garantir que le système peut gérer efficacement une grande quantité de données provenant de nombreux appareils distants.

Intégration avec Go (langage utilisé pour le client et le serveur) :

MongoDB offre des bibliothèques officielles bien prises en charge pour Go (via le pilote MongoDB pour Go), ce qui simplifie l'intégration de la base de données dans le code du client et du serveur.

L'utilisation d'un pilote bien pris en charge garantit une compatibilité étroite avec les fonctionnalités et les performances de MongoDB.

❖ **Community et Documentation :**

MongoDB bénéficie d'une large communauté d'utilisateurs et d'une documentation exhaustive, ce qui facilite le développement, le dépannage et la maintenance du système.

En cas de besoin d'aide ou de résolution de problèmes, il est généralement facile de trouver des ressources et des exemples pour travailler avec MongoDB.

En conclusion, le choix de MongoDB comme base de données pour ce projet est motivé par sa flexibilité du schéma, sa capacité à gérer efficacement des données JSON, sa scalabilité horizontale, son intégration bien prise en charge avec le langage Go, ainsi que par la robustesse de sa communauté et de sa documentation. Ce choix permettra de construire un système client-serveur performant et évolutif, adapté aux besoins spécifiques de collecte et de traitement de données provenant d'appareils distants.

III. Développement du système

A. Implémentation du client

Le client Go présenté est le programme conçu pour interagir avec un serveur gRPC, une architecture de communication distante. Écrit en langage Go, il est conçu pour envoyer des fichiers JSON au serveur à des intervalles réguliers.

Le fichier principal, ***Client.go***, orchestre le processus. Il commence par importer les packages nécessaires, notamment "fmt" pour la gestion de l'affichage, "os" pour les opérations système et "time" pour la gestion du temps. Ensuite, la fonction principale ***main()*** est définie. À l'intérieur de cette fonction, une boucle itère cinq fois, et à chaque itération, elle appelle la fonction ***ShowDevices()*** pour envoyer un fichier JSON au serveur. Entre chaque envoi, le programme attend trois secondes à l'aide de `time.Sleep(3 * time.Second)`. Une fois la boucle terminée, le programme se termine proprement avec `os.Exit(0)`.

Le fichier *data.go* contient la logique métier du client. Il définit le package "data" et importe les packages nécessaires tels que "fmt", "io/ioutil", "os", "log", et le package gRPC *"google.golang.org/grpc"*, ainsi que le package généré à partir du fichier proto *"Bakri-Souhail/GoGrpcClient/grpc"*. Ce fichier contient également la définition d'une structure Data qui représente le contenu JSON.

Les fonctions ***ReadFiles()*** et ***OpenJsonFile()*** sont responsables de la lecture des fichiers JSON et de l'ouverture des fichiers JSON en fonction du choix spécifié, respectivement. La fonction ***ShowDevices()*** orchestre l'ouverture du fichier JSON, sa lecture et son envoi au serveur. La fonction ***SendToServer()*** établit une connexion gRPC avec le serveur et envoie la chaîne JSON au serveur.

Le fichier `proto grpc/data.proto` définit le service ***StringService*** avec une méthode `SendString` qui prend une demande de type ***StringRequest*** et renvoie une réponse de type ***StringResponse***. Ce fichier est utilisé pour générer le code Go nécessaire à la communication avec le serveur gRPC.

Ce client Go constitue un composant essentiel d'un système plus large, communiquant avec un serveur via gRPC pour envoyer des données JSON de manière périodique.

B. Implémentation du serveur

Le serveur Go présenté constitue un composant central de notre système, conçu pour recevoir, traiter et enregistrer les données envoyées par un client via gRPC, un framework de communication à distance. Ce serveur traite les données JSON transmises par le client, extrait les informations pertinentes, puis les stocke dans une base de données MongoDB pour une gestion efficace.

❖ Fichier `server.go` :

Le fichier ***server.go*** définit un serveur gRPC qui écoute sur le port **50051** pour recevoir des requêtes clients et fournir des réponses en retour. Voici ce que chaque partie du code accomplit :

- Définition de la structure du serveur (`server`) :

Le serveur est défini comme une structure ***server*** qui implémente l'interface ***StringServiceServer*** générée par gRPC pour le service `StringService`.

- Fonction `main()` :

La fonction ***main()*** initialise le serveur en écoutant les connexions entrantes sur le port 50051 via ***net.Listen()***.

En cas d'erreur lors de l'écoute, elle affiche un message d'erreur et arrête l'exécution.

Ensuite, elle crée un nouveau serveur gRPC avec ***grpc.NewServer()***.

Le service ***StringService*** est enregistré sur le serveur gRPC avec ***pb.RegisterStringServiceServer(s, &server{})***, où `s` est le serveur créé et ***&server{}*** est une instance du serveur que nous avons défini.

Une fois que le serveur gRPC est prêt à recevoir des connexions, la fonction ***Serve()*** est appelée sur le serveur avec ***s.Serve(lis)***.

Si une erreur se produit pendant le démarrage du serveur, elle est gérée en affichant un message d'erreur et en arrêtant l'exécution.

- **Méthode *SendString()* :**

Cette méthode est une implémentation de la méthode ***SendString*** définie dans l'interface ***StringServiceServer***.

Elle reçoit une requête ***StringRequest*** du client contenant un message JSON représentant les opérations effectuées sur différents appareils.

À l'intérieur de cette méthode, la fonction ***CountOperations()*** du package ***device*** est appelée pour traiter la requête du client.

En fonction du résultat de ***CountOperations()***, cette méthode construit une réponse en formatant les statistiques d'opérations pour chaque appareil et les renvoie au client sous la forme d'une ***StringResponse***.

- ❖ **Fichier *device.go* :**

Le fichier ***device.go*** contient les définitions et les méthodes essentielles pour gérer les informations relatives aux appareils et interagir avec la base de données MongoDB. Voici un aperçu détaillé de son contenu :

- **Fonction *init()* :**

La fonction ***init()*** est une fonction spéciale en Go qui est automatiquement appelée lors du chargement du package dans lequel elle est définie.

Dans ce contexte, elle initialise le client MongoDB en établissant une connexion à la base de données spécifiée à l'aide de ***mongo.Connect()***.

En cas d'échec de la connexion, la fonction ***panic()*** est utilisée pour arrêter brusquement l'application et afficher un message d'erreur.

- **Structure *Device* :**

La structure ***Device*** est définie pour représenter les informations d'un appareil. Elle inclut les champs suivants :

DeviceName : Le nom de l'appareil.

TotalOperations : Le nombre total d'opérations effectuées par l'appareil.

FailedOperations : Le nombre d'opérations échouées pour l'appareil.

Ces champs sont annotés avec des tags JSON pour indiquer le nom des champs dans les données JSON correspondantes.

- **Méthode *StoreStatistics()* :**

La méthode ***StoreStatistics()*** est associée à la structure ***Device*** et permet d'enregistrer les statistiques d'un appareil dans la collection ***device_statistics*** de la base de données MongoDB. Elle utilise le client MongoDB pour insérer les statistiques sous forme de document JSON dans la collection spécifiée.

En cas d'erreur lors de l'insertion, elle retourne une erreur indiquant l'échec de l'opération.

- **Méthode *StoreOperations()* :**

La méthode ***StoreOperations()*** est une autre méthode de la structure ***Device*** qui enregistre les détails des opérations d'un appareil dans la collection ***device_operations*** de la base de données MongoDB.

Elle utilise le client MongoDB pour insérer les détails des opérations sous forme de document JSON dans la collection spécifiée.

Si une erreur survient lors de l'insertion, elle retourne une erreur indiquant l'échec de l'opération.

- **Fonction *CountOperations()* :**

La fonction ***CountOperations()*** prend une chaîne de données JSON représentant les opérations effectuées sur différents appareils.

Elle décode cette chaîne JSON en une structure de données appropriée, puis boucle à travers chaque appareil pour compter le nombre total d'opérations et le nombre d'opérations échouées.

Pour chaque appareil, elle crée une instance ***Device*** pour stocker les statistiques calculées et utilise les méthodes ***StoreStatistics()*** et ***StoreOperations()*** pour enregistrer ces informations dans la base de données MongoDB.

En résumé, les fichiers ***server.go*** et ***device.go*** travaillent ensemble pour fournir un système robuste de gestion des données d'appareils via gRPC et MongoDB. Le serveur gRPC écoute les requêtes clients et utilise les fonctionnalités définies dans ***device.go*** pour traiter les données JSON et les stocker efficacement dans la base de données. Cette architecture garantit une gestion efficace des informations d'appareils dans un environnement distribué.

C. Tests unitaires et validation

1. Serveur

Le fichier ***device_test.go*** contient des tests unitaires pour les fonctionnalités du package ***device*** dans le serveur Go. Voici une explication du contenu de ce fichier :

Le package ***device_test*** est utilisé pour regrouper les tests unitaires liés au package ***device*** du serveur Go.

Les importations sont effectuées pour inclure les dépendances nécessaires aux tests, telles que le package à tester (Bakri-Souhail/server/device), des outils de test (testing), des assertions facilitées (github.com/stretchr/testify/assert), et les fonctionnalités de base de la base de données MongoDB (go.mongodb.org/mongo-driver/mongo).

La fonction *Test_StoreToDatabase_CountOperations* est définie pour tester deux fonctionnalités principales : *StoreToDatabase()* et *CountOperations()*.

Dans cette fonction de test :

Un exemple d'appareil *deviceInstance* avec des valeurs fictives est créé pour être stocké dans la base de données.

La méthode *StoreToDatabase()* est appelée pour stocker l'appareil fictif dans la base de données MongoDB. Si une erreur se produit, le test échoue avec un message d'erreur.

Ensuite, la connexion à la base de données est établie et la fonction *FindOne()* est utilisée pour récupérer l'appareil précédemment stocké. Si une erreur se produit, le test échoue avec un message d'erreur.

Les assertions sont utilisées pour comparer les propriétés de l'appareil stocké avec les valeurs d'origine.

Après cela, un exemple de données JSON est défini représentant deux appareils avec différentes opérations pour tester la fonction *CountOperations()*.

Les résultats attendus sont également définis.

La fonction *CountOperations()* est appelée avec les données JSON de test et les résultats sont comparés avec les résultats attendus à l'aide des assertions.

Si une erreur se produit lors de l'appel à *CountOperations()*, le test échoue avec un message d'erreur.

Enfin, le test vérifie que les résultats obtenus correspondent aux résultats attendus.

Ces tests garantissent que les fonctionnalités du package device sont correctement implémentées et produisent les résultats attendus dans divers scénarios, ce qui contribue à assurer la fiabilité et la stabilité du serveur Go dans son ensemble.

```
toto@srv:~/Projet0805/GogRPC/server$ docker container exec -it 351a0fbc60b0 /bin/bash
root@351a0fbc60b0:/go/src/app# cd device/
root@351a0fbc60b0:/go/src/app/device# go test -cover -coverprofile=cover.out ./...
ok      Bakri-Souhail/server/device    0.103s  coverage: 87.5% of statements
```

Figure 3 : Résultat du test pour le serveur (Couverture 87.5%)

2. Client

Le package *data_test* est importé pour exécuter les tests unitaires pour le package data du projet. Les imports incluent également les packages nécessaires pour les tests et les assertions.

La fonction **Test_Data** est définie comme une fonction de test pour tester les différentes fonctionnalités du package data.

Dans la première partie du test, un fichier temporaire est créé à l'aide de `os.CreateTemp()` pour simuler un fichier JSON. Des données JSON sont écrites dans ce fichier temporaire à l'aide de `WriteString()`. Ensuite, la fonction `ReadFiles()` du package data est appelée pour lire les données du fichier temporaire. Les données lues sont ensuite comparées aux données attendues à l'aide de `assert.Equal()` pour vérifier si la lecture du fichier fonctionne correctement.

Dans la deuxième partie du test, la fonction `OpenJsonFile()` du package data est testée. Deux cas sont testés : l'ouverture d'un fichier JSON existant et l'ouverture d'un fichier JSON inexistant. L'assertion `assert.NoError()` est utilisée pour vérifier qu'aucune erreur n'est retournée lors de l'ouverture du fichier existant, tandis que l'assertion `assert.Error()` est utilisée pour vérifier qu'une erreur est retournée lors de l'ouverture du fichier inexistant.

À la fin de chaque test, la fonction `defer` est utilisée pour supprimer le fichier temporaire une fois le test terminé, garantissant ainsi que les ressources temporaires sont nettoyées après l'exécution du test.

IV. Environnement de déploiement

A. Configuration du fichier docker-compose

Le fichier Docker Compose définit une configuration multi-conteneurs pour un environnement de développement. Il comprend plusieurs services déployés dans des conteneurs Docker distincts.

Le premier service, nommé "db", utilise l'image Docker officielle de MongoDB. Il est configuré pour redémarrer automatiquement et expose le port 27017 pour permettre l'accès à la base de données. Les variables d'environnement sont définies pour spécifier le nom d'utilisateur et le mot de passe de l'administrateur de la base de données. Le deuxième service, appelé "db-client", utilise l'image Docker de mongo-express, une interface web pour MongoDB.

Il est également configuré pour redémarrer automatiquement et expose le port 8081 pour accéder à l'interface web. Les variables d'environnement sont définies pour spécifier le nom d'utilisateur, le mot de passe de l'administrateur de la base de données et l'URL de connexion à MongoDB. Ce service dépend du service "db" pour s'assurer que la base de données est prête avant de démarrer. Ensuite, deux autres services sont configurés : "server" et "client".

Ils sont construits à partir de fichiers Dockerfile situés dans les répertoires "server" et "client" respectivement. Ces services exposent respectivement les ports 50051 et 50052 pour permettre la communication avec les applications serveur et client.

Ils dépendent tous les deux du service "db-client" pour s'assurer que l'interface web MongoDB est opérationnelle avant de démarrer. En résumé, ce fichier Docker Compose simplifie le déploiement et la gestion d'un environnement de développement complet comprenant une base de données MongoDB, une interface web pour cette base de données, un serveur et un client d'application.

B. Mise en place de la base de données

Pour mettre en place la base de données MongoDB dans notre environnement de développement en utilisant Docker Compose, nous avons configuré correctement le fichier *docker-compose.yml* situé dans le répertoire racine de notre projet.

Ce fichier Docker Compose définit plusieurs services Docker pour créer un environnement complet comprenant MongoDB, mongo-express, ainsi que les applications serveur et client.

Voici une explication détaillée de chaque service :

❖ Service db (MongoDB) :

```
version: "3"
services:
  db:
    image: mongo
    container_name: bdd-mongo
    restart: always
    ports:
      - 27017:27017
    environment:
      - MONGO_INITDB_ROOT_USERNAME=root
      - MONGO_INITDB_ROOT_PASSWORD=root
```

- Utilise l'image Docker officielle de MongoDB.
- Définit le nom du conteneur comme **bdd-mongo**.
- Redémarre le conteneur automatiquement en cas d'échec.
- Expose le port **27017** pour permettre l'accès à la base de données MongoDB.
- Définit le nom d'utilisateur et le mot de passe de l'administrateur de la base de données avec les variables d'environnement **MONGO_INITDB_ROOT_USERNAME** et **MONGO_INITDB_ROOT_PASSWORD**.

❖ Service *db-client* (Mongo Express) :


```
db-client:
  image: mongo-express
  restart: always
  ports:
    - 8081:8081
  environment:
    - ME_CONFIG_MONGODB_ADMINUSERNAME=root
    - ME_CONFIG_MONGODB_ADMINPASSWORD=root
    - ME_CONFIG_MONGODB_URL=mongodb://root:root@db:27017/
  depends_on:
    - db
```

- Utilise l'image Docker de mongo-express pour fournir une interface web à MongoDB.
- Redémarre le conteneur automatiquement en cas d'échec.
- Expose le port **8081** pour accéder à l'interface web de mongo-express.
- Définit le nom d'utilisateur et le mot de passe de l'administrateur pour mongo-express.
- Définit l'URL de connexion à MongoDB avec les informations d'authentification.

❖ **Service *server* :**

```
server:
  build:
    context: ./server
    dockerfile: Dockerfile
  ports:
    - "50051:50051"
  depends_on:
    - db-client
```

- Construit l'image du serveur à partir du Dockerfile situé dans le répertoire *./server*.
- Expose le port **50051** pour permettre la communication avec l'application serveur.
- Dépend du service ***db-client*** pour s'assurer que mongo-express est démarré avant le serveur.

❖ **Service *client* :**

```
client:
  build:
    context: ./client
    dockerfile: Dockerfile
  ports:
    - "50052:50052"
```

```
depends_on:  
  - server
```

- Construit l'image du client à partir du Dockerfile situé dans le répertoire *./client*.
- Expose le port **50052** pour permettre la communication avec l'application client.
- Dépend du service *server* pour s'assurer que le serveur est démarré avant le client.

C. Déploiement du système

Pour déployer le système sur une machine virtuelle Linux, vous devez d'abord vous positionner dans le répertoire racine du projet où se trouve le fichier Docker Compose. Assurez-vous d'avoir Docker et Docker Compose installés sur la machine virtuelle. Une fois dans le répertoire racine du projet, vous pouvez lancer la commande suivante :

```
docker-compose up --build
```

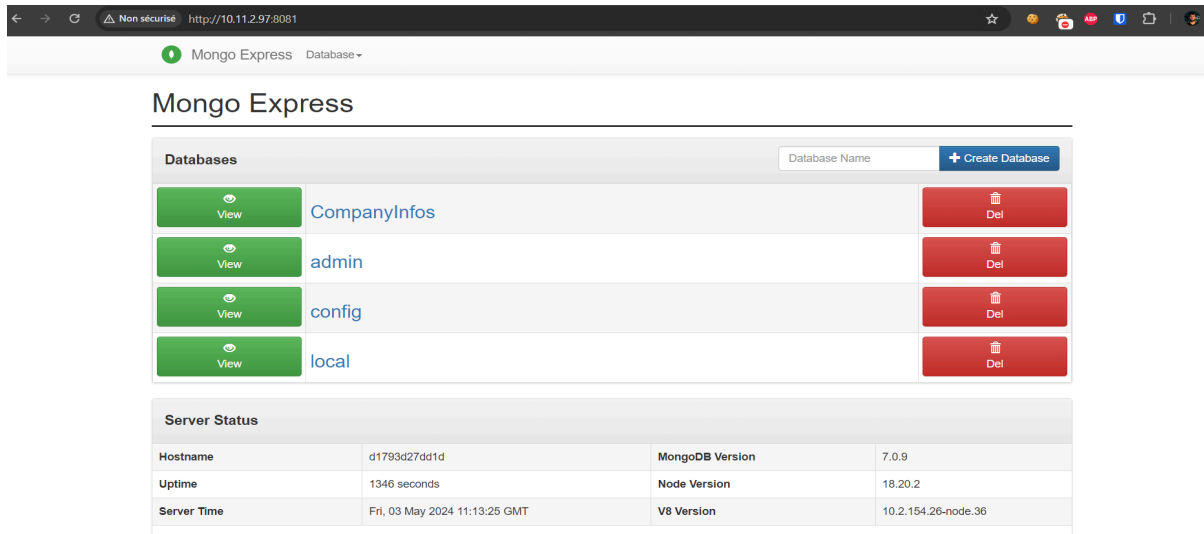
Cette commande va construire les images Docker nécessaires pour les services définis dans le fichier Docker Compose et démarrer les conteneurs correspondants. L'option `--build` est utilisée pour forcer la reconstruction des images Docker, ce qui garantit que toutes les modifications récentes apportées aux fichiers Dockerfile sont prises en compte.

Une fois que la commande est exécutée, Docker Compose va créer les conteneurs Docker pour chaque service défini dans le fichier Docker Compose, en respectant les dépendances entre les services. Par exemple, il s'assurera que la base de données est démarrée avant de lancer le serveur ou le client.

Vous pouvez consulter la base de données en tapant votre ip avec le port 8081(par exemple 10.11.2.97:8081). Les identifiants de la base de données mongoDB sont ceux par défaut (admin, pass).

V. Résultats et fonctionnement

A. Démonstration du fonctionnement du système



Mongo Express

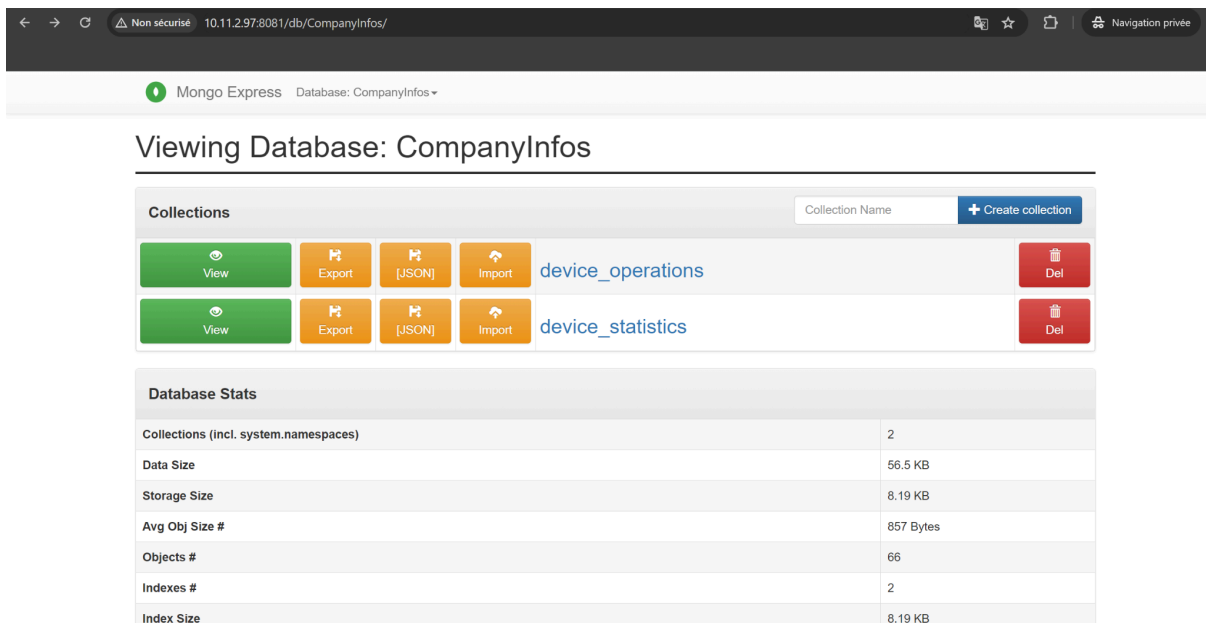
Databases

Database Name	View	Del
CompanyInfos	View	Del
admin	View	Del
config	View	Del
local	View	Del

Server Status

Hostname	d1793d27dd1d	MongoDB Version	7.0.9
Uptime	1346 seconds	Node Version	18.20.2
Server Time	Fri, 03 May 2024 11:13:25 GMT	V8 Version	10.2.154.26-node.36

Figure 4 : Le serveur de db MongoDB



Viewing Database: CompanyInfos

Collections

Collection Name	View	Export	[JSON]	Import	Del
device_operations	View	Export	[JSON]	Import	Del
device_statistics	View	Export	[JSON]	Import	Del

Database Stats

Collections (incl. system.namespaces)	2
Data Size	56.5 KB
Storage Size	8.19 KB
Avg Obj Size #	857 Bytes
Objects #	66
Indexes #	2
Index Size	8.19 KB

Figure 5 : Les différentes tables dans la base des données

Navigation privée

Mongo Express Database: CompanyInfos → Collection: device_statistics

1 2 3 > >>



















_id	device_name	total_operations	failed_operations
  6641b85705da8b2afa58c1de	c1153f7a-b060-4215-bf22-601e8f8e704c	15	3
  6641b85c05da8b2afa58c1e0	971645e6-6870-4db6-9e6b-817227d8f338	30	6
  6641b85c05da8b2afa58c1e2	547ec23d-e97f-4a5d-8488-95c6b423933e	45	9
  6641b85c05da8b2afa58c1e4	f632d672-2be0-4f09-ac40-1eaa5d913b23	60	12
  6641b85c05da8b2afa58c1e6	0b3939ec-06d4-48e0-ade9-d06e48fd4fe0	75	15
  6641b85c05da8b2afa58c1e8	d79f5841-f7d9-4b4d-a226-baf69b2ebb0d	90	18
  6641b85f05da8b2afa58c1ea	c1153f7a-b060-4215-bf22-601e8f8e704c	0	0
  6641b85f05da8b2afa58c1ec	971645e6-6870-4db6-9e6b-817227d8f338	0	0
  6641b85f05da8b2afa58c1ee	547ec23d-e97f-4a5d-8488-95c6b423933e	0	0

Figure 6 : Machines stockées avec leurs informations

The screenshot shows the Mongo Express web interface. At the top, the browser address bar displays the URL: `10.11.2.97:8081/db/CompanyInfos/device_operations`. The interface header shows the database name **CompanyInfos** and the collection name **device_operations**.

The main content area displays two REST client requests:

- Request 1:** A DELETE request to the `device_operations` collection. The request body is:


```
{
  "type": "DELETE",
  "hasSucceeded": false
},
{
  "type": "CREATE",
  "hasSucceeded": true
},
[
  {2 items},
  {2 items},
  {2 items},
  {2 items},
  {2 items},
  {2 items},
  {2 items},
  {2 items},
  {2 items},
  {2 items},
  {2 items},
  {2 items},
  {2 items}
]
```
- Request 2:** A DELETE request to the `device_operations` collection. The request body is:


```
{
  "type": "DELETE",
  "hasSucceeded": false
}
```

Figure 7 : Machines stockées avec leurs opérations

VI. Conclusion

Dans le cadre du projet Go gRPC, nous avons exploré plusieurs aspects essentiels, notamment la mise en place d'un système de communication à distance robuste et efficace

grâce à gRPC, ainsi que la manipulation de données JSON et leur stockage dans une base de données MongoDB.

Communication distante avec gRPC : En utilisant gRPC, nous avons mis en place un système de communication à distance moderne et performant. Les services gRPC permettent de définir des méthodes et des types de données à échanger entre le client et le serveur de manière efficace et sécurisée.

Manipulation de données JSON : le projet comprend la lecture de fichiers JSON côté client, ainsi que l'analyse et le traitement de ces données côté serveur. Cela démontre une bonne compréhension des opérations de manipulation de données JSON en Go.

Intégration avec MongoDB : L'utilisation de MongoDB comme base de données pour stocker les données traitées dans le système montre une approche moderne et flexible pour la gestion des données. L'intégration avec MongoDB nous permet de stocker et de récupérer des données de manière efficace et évolutive.

Tests unitaires : Les tests unitaires que nous avons mis en place démontrent votre engagement envers la qualité du code. Les tests couvrent différents aspects du code, y compris la lecture de fichiers, l'ouverture de fichiers et le traitement des données, ce qui renforce la robustesse et la fiabilité de votre application.

Déploiement avec Docker : L'utilisation de Docker pour encapsuler l'application et ses dépendances facilite le déploiement et la gestion du système dans différents environnements. Docker Compose permet une configuration et un déploiement simples et reproductibles de l'application sur des machines virtuelles ou des serveurs.

En conclusion, le projet Go gRPC nous aide à gagner une solide compréhension des principes fondamentaux du développement logiciel, y compris la communication distante, la manipulation de données, l'intégration avec des bases de données et la mise en œuvre de tests unitaires. Avec une attention portée à la qualité du code, à la robustesse et à la scalabilité.

VII. Annexes

A. Schémas de conception

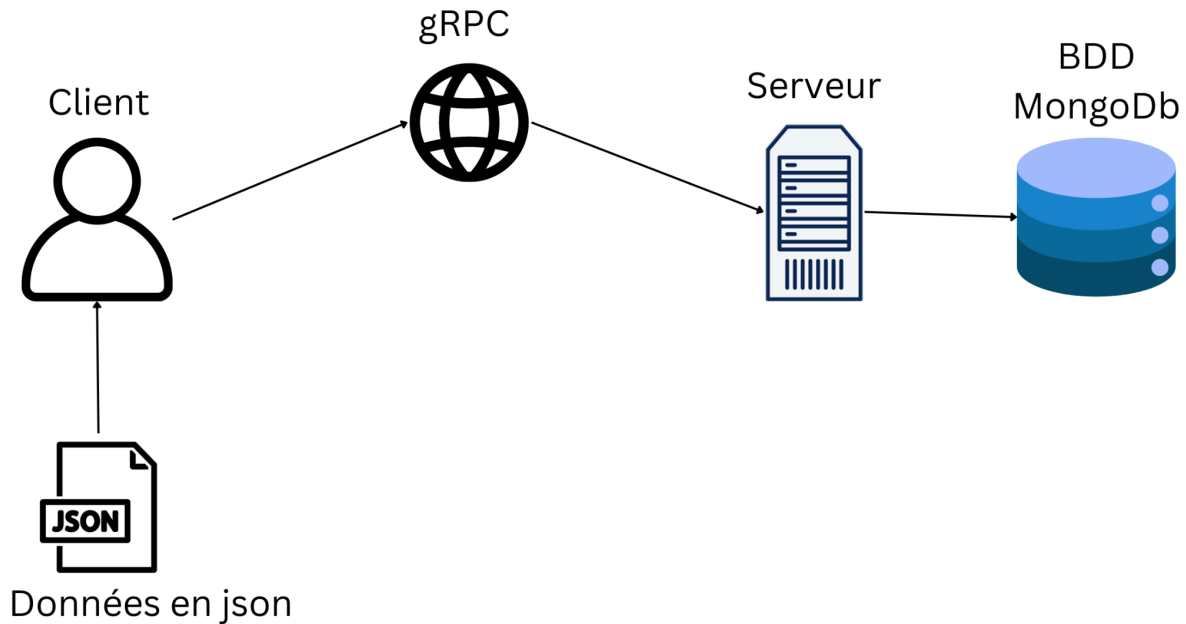


Figure 6 : Schéma explicatif de l'architecture globale du projet

B. Exemples de fichiers JSON

```
[{"device_name":"c1153f7a-b060-4215-bf22-601e8f8e704c","operations":[{"type":"DELETE","has_succeeded":false}, {"type":"CREATE","has_succeeded":true}, {"type":"DELETE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}, {"type":"DELETE","has_succeeded":true}, {"type":"UPDATE","has_succeeded":false}, {"type":"CREATE","has_succeeded":true}, {"type":"UPDATE","has_succeeded":true}, {"type":"DELETE","has_succeeded":true}, {"type":"UPDATE","has_succeeded":true}, {"type":"DELETE","has_succeeded":false}, {"type":"UPDATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}, {"type":"UPDATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}], {"device_name":"971645e6-6870-4db6-9e6b-817227d8f338","operations":[{"type":"DELETE","has_succeeded":false}, {"type":"CREATE","has_succeeded":true}, {"type":"DELETE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}, {"type":"DELETE","has_succeeded":true}, {"type":"UPDATE","has_succeeded":false}, {"type":"CREATE","has_succeeded":true}, {"type":"UPDATE","has_succeeded":true}, {"type":"DELETE","has_succeeded":false}, {"type":"UPDATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}, {"type":"UPDATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":false}, {"type":"CREATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}, {"type":"DELETE","has_succeeded":true}, {"type":"DELETE","has_succeeded":true}, {"type":"CREATE","has_succeeded":false}, {"type":"UPDATE","has_succeeded":true}, {"type":"UPDATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}, {"type":"UPDATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":false}, {"type":"UPDATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}], {"device_name":"547ec23d-e97f-4a5d-8488-95c6b423933e","operations":[{"type":"DELETE","has_succeeded":false}, {"type":"CREATE","has_succeeded":true}, {"type":"DELETE","has_succeeded":true}, {"type":"CREATE","has_succeeded":true}, {"type":"DELETE","has_succeeded":true}]}
```

VIII. Références

A. Sources documentaires utilisées

<https://go.dev/doc/>

<https://grpc.io/docs/protoc-installation/#install-using-a-package-manager>

<https://docs.docker.com/>

<https://doc.ubuntu-fr.org/>