UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et génie informatique

RAPPORT FINALE

Projet de développement en génie informatique
GEI806

Présenté à
Frédéric Mailhot

Présenté par
Ramzi Benzohra - benr1104

Montréal – 25 Juillet

Table des matières

I. Introduction :

Il y a longtemps, les développeurs utilisaient l'architecture monolithique pour développer des logiciels, mais cette architecture leur posait de nombreux problèmes, étant donné que l'application est assemblée en un seul bloc, ce bloc grandit chaque fois que les développeurs ajoutent des fonctionnalités à l'application. Ainsi, si une partie du code échoue, tout le système échoue également. Ce n'est qu'un des nombreux problèmes de cette architecture.

La solution de cette problématique est l'architecture des micro-services, où les développeurs séparent l'application en plusieurs morceaux, généralement interconnectés. Chaque composant de l'application est indépendant des autres, ce qui signifie qu'en cas de panne, il sera facile de le remplacer par un nouveau. Cela rend l'application plus flexible et plus résistante aux bugs et aux pannes.

Dans ce projet, nous allons créer une application de gestion de tâches en utilisant le langage de programmation Python et la base de données PostgreSQL déployée sur Kubernetes. Bien sûr, nous suivrons l'architecture des micro-services.

Notre application sera divisée en trois composants qui fonctionnent ensemble. Nous avons le serveur API, qui communique avec les autres composants ; le composant base de données, où les données sont stockées ; et enfin le composant application, avec lequel l'utilisateur interagit pour stocker et afficher ses tâches dans la base de données.

Comme nous divisons l'application en trois composants, nous pouvons facilement faire évoluer chaque composant. Par exemple, nous pouvons créer plus d'instances de la base de données pour réduire la charge sur une seule instance. Cela s'applique également au serveur. Ainsi, en ayant plus d'instances du serveur et de la base de données, l'application devient plus flexible, offrant ainsi une meilleure expérience utilisateur.

II. Problématique :

La problématique est de fournir la flexibilité et la disponibilité du service pour un composant de l'application. Cela signifie que votre service doit être disponible et fonctionne sans problème pour les utilisateurs à tout moment. Dans cet exemple, nous nous concentrons sur la base de données PostgreSQL.

Imaginez que vous êtes responsable d'un projet avec un seul employé sous votre responsabilité. Chaque fois que vous lui donnez une tâche à accomplir, il doit terminer la première avant de commencer la suivante. Après quelques tâches, l'employé se fatigue et ne peut plus travailler efficacement. Vous devez alors le remplacer ou embaucher d'autres employés pour gérer la charge de travail plus efficacement.

La même logique s'applique à la base de données. Si une seule instance de la base de données fonctionne sur un serveur, elle a une capacité limitée de requêtes qu'elle peut traiter en fonction des ressources du serveur. Cette instance peut également cesser de fonctionner ou planter en raison d'un bug ou d'une mauvaise entrée. Sans une solution adéquate, nous ne pourrons pas garantir la disponibilité du service, ce qui va à l'encontre de notre objectif.

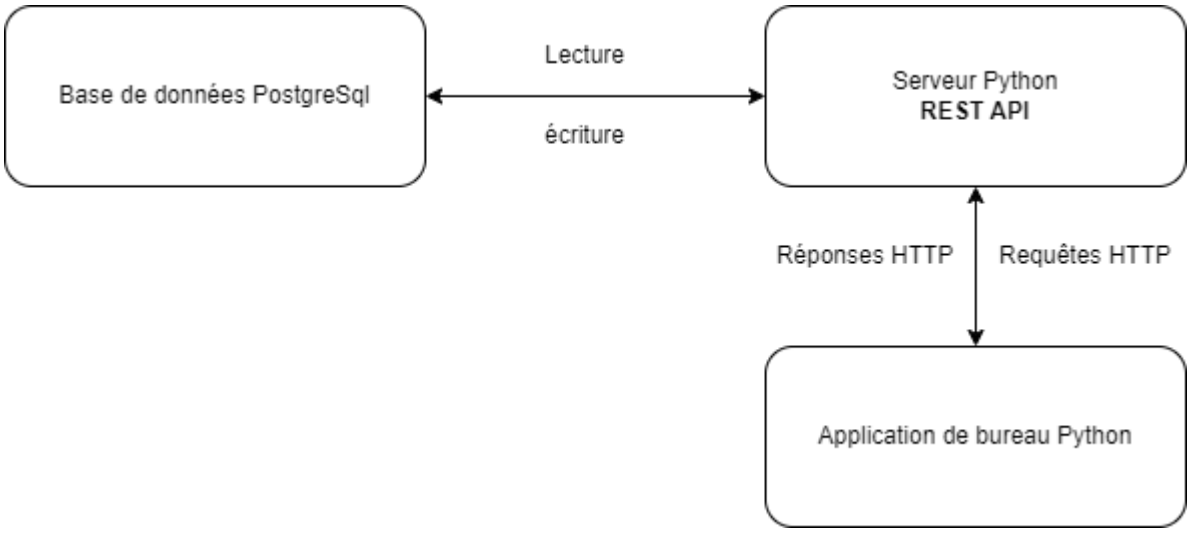
III. Solution :

La solution classique à ce problème est l’augmentation de la capacité du serveur. Comme l'application web est hébergée sur un serveur, plus nous ajoutons de RAM et de CPU à ce serveur, plus il sera rapide. Cela permet de résister à une grande quantité de requêtes et de réduire le temps de traitement. Cependant, cette approche est également limitée : il y a une limite aux ressources que nous pouvons ajouter, et cela devient de plus en plus coûteux avec le temps.

La solution moderne à ce problème consiste à ajouter plus d'instances du composant au serveur, c'est ce qu'on appelle la mise à l'échelle horizontale. Si une instance échouée, le trafic peut être redirigé vers les autres instances. Nous pouvons également équilibrer la charge pour chaque instance. Si une instance n'est pas capable de gérer ses requêtes, une autre instance disponible peut les prendre en charge. Ainsi, nous pourrons gérer un volume élevé de requêtes de manière efficace afin d’assurer la disponibilité continue du service.

Dans cette problématique, nous allons faire évoluer le composant PostgreSQL de notre application en utilisant une technologie appelée Kubernetes.

1. L’architecture de l’application :



Nous utiliserons une API REST Python Flask comme composant intermédiaire entre l'application de bureau et la base de données PostgreSQL.

L'application de bureau est une application de gestion de tâches où l'utilisateur peut afficher ses tâches à faire, aussi d’ajouter ou supprimer des tâches de sa liste.

L'API REST est écrite en utilisant le Framework Flask. La base de données est une API PostgreSQL qui reçoit les demandes de lecture et d'écriture de l'API (la base de données est dans son état naturel).

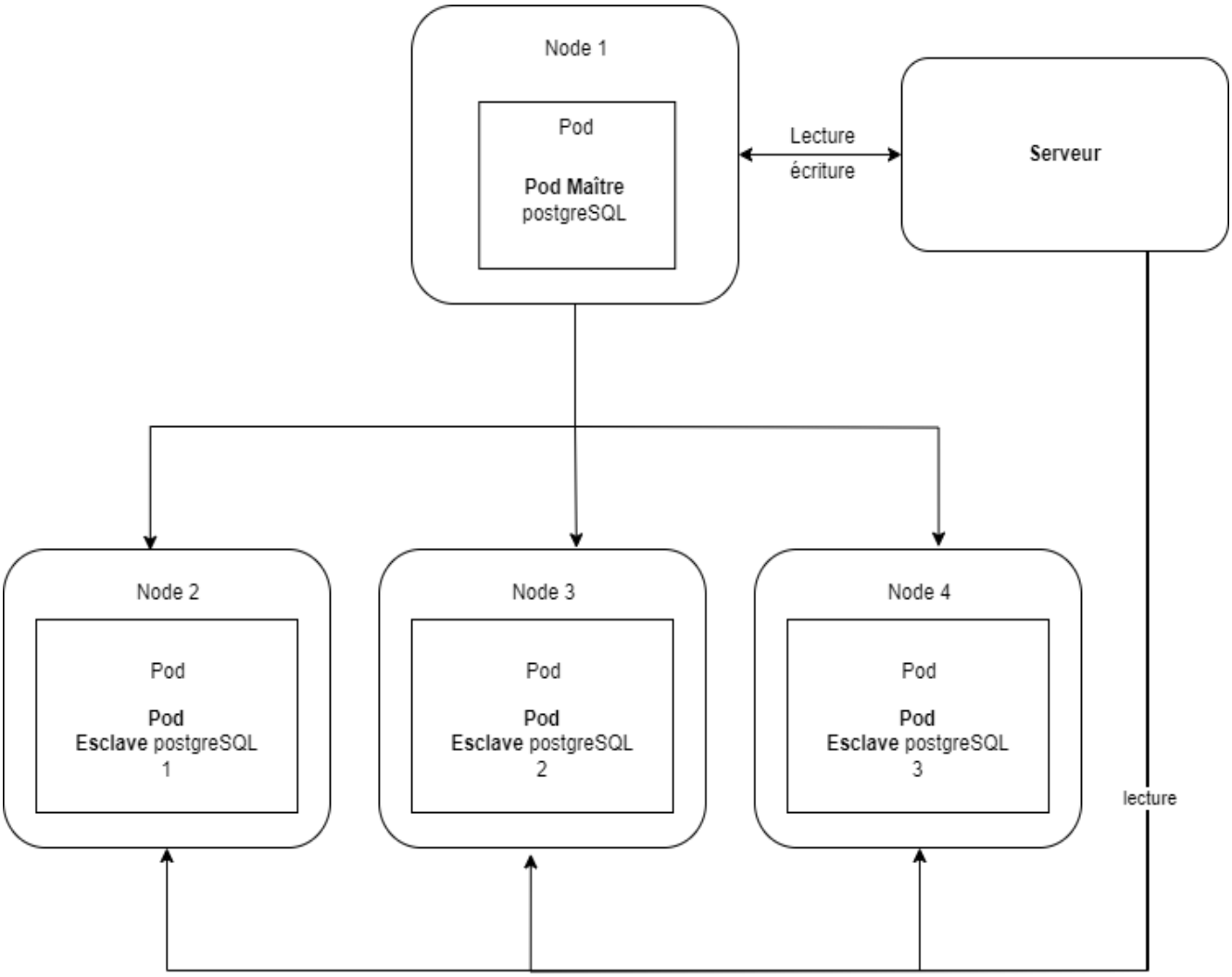
2. PostgreSQL sur Kubernetes :

- Kubernetes et Docker :

Docker est une technologie de conteneurisation qui permet de créer, expédier et exécuter des applications en utilisant des conteneurs, tandis que Kubernetes est une plateforme open-source d'orchestration de conteneurs qui automatise le déploiement, la mise à l'échelle et la gestion des applications conteneurisées.

Dans ce projet, nous utiliserons l'image officielle Docker de PostgreSQL déployée sur Kubernetes. Kubernetes créera plusieurs répliques de la base de données et répartira la charge entre elles.

- Voici le schéma de la base de données PostgreSQL hébergée dans l'environnement Kubernetes :



Ce schéma montre le plan de notre base de données hébergée dans Kubernetes, où nous avons une instance maître et deux ou plusieurs instances esclaves. Pour répartir la charge entre elles, le pod maître est utilisé pour les requêtes de lecture et d'écriture, tandis que les pods esclaves sont utilisés uniquement pour les opérations de lecture.

Dans le cas où ces répliques ne sont pas capables de gérer la charge, nous pouvons ajouter plus de répliques (instances). Également, si l'une d'elles tombe en panne ou ne répond pas, Kubernetes va la détruire et recréera d'autres répliques saines.

3. La mise en place de PostgreSQL dans Kubernetes :

Pour configurer notre base de données afin qu'elle soit hébergée sur Kubernetes pour activer la réplication entre les instances maître et esclave, nous devons suivre les étapes suivantes :

Tout d'abord nous devons écrire le fichier YAML afin de définir les étapes pour observer comment nous voulons configurer notre base de données dans Kubernetes, ci-dessous un exemple de configuration de base :

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres-gei806
spec:
  replicas: 2
  selector:
    matchLabels:
      app: postgres-gei806
  template:
    metadata:
      labels:
        app: postgres-gei806
    spec:
      containers:
      - name: postgres-gei806
        image: postgres:latest
        ports:
        - containerPort: 5432
        env:
        - name: POSTGRES_PASSWORD
          value: docker_gei806
        volumeMounts:
        - name: postgres-data
          mountPath: /var/lib/postgres_gei806/data
  volumeClaimTemplates:
  - metadata:
      name: postgres-data
    spec: # These fields must be nested under 'spec'
      accessModes: ["ReadWriteOnce"]
```

Dans ce fichier, j'ai défini un StatefulSet Kubernetes, nommé "postgres-gei806". Ce StatefulSet me permet de déployer deux réplicas du conteneur PostgreSQL, le premier étant le maître et l'autre l'esclave. J'utilise l'image officielle de PostgreSQL sur Docker Hub "postgres:latest" et j'expose le port 5432 pour permettre les connexions à la base de données. Également, j'ai défini le mot de passe de la base de données dans la variable d'environnement "POSTGRES_PASSWORD" et lui ai donné la valeur 'docker_gei806'.

Afin d’assurer la persistance des données, j'ai défini un volume persistant appelé "postgres-data" qui stocke toutes les informations importantes. Ce volume est configuré en mode d'accès "ReadWriteOnce" avec une capacité de stockage minimale de 1 Go.

Ce fichier ne créera que deux instances dans deux Pod et dans un seul Node. Nous en utiliserons une comme maître et l'autre sera configurer pour synchroniser les données du maître.

- Comment accéder à cette configuration depuis notre machine locale ?

Il y a deux façons de le faire : l'une pour la conception et les tests, et l'autre pour la production et le développement :

Pour tester la connexion avec les bases de données, nous devrons transférer le trafic de l'environnement Kubernetes vers notre machine locale. Tout d'abord, nous devons créer deux services : un service Headless pour exposer le Pod maître, et un service ClusterIP pour gérer les demandes de lecture entre le Pod maître et l'esclave. Ces deux services fonctionnent à l'intérieur du cluster Kubernetes, nous devons donc, transférer le trafic depuis eux vers notre machine locale en utilisant les deux commandes dans deux fenêtres PowerShell :

- Pour exposer le Pod maitre : kubectl port-forward postgres-gei806-0 5432:5432
- Pour exposer le service : kubectl port-forward service/postgres-service 5433:5432

Maintenant, nous pouvons nous connecter aux instances PostgreSQL depuis notre machine locale. Nous pouvons nous connecter à `localhost:5432` pour accéder au pod maître et à `localhost:5433` pour les demandes de lecture.

Mais à des fins de développement, il existe un type de service appelé Ingress. Avec ce service, nous pouvons configurer Kubernetes pour exposer les Pods et les services via des requêtes HTTP. Cela simplifié grandement l'accès aux applications depuis notre machine locale. En effet, il agit comme un point d'entrée unique pour tout le trafic externe, et Kubernetes se charge de rediriger les requêtes vers le bon Pod ou service en fonction des règles définies. De plus, l'Ingress permet de gérer les certificats TLS/SSL, ce qui facilite la mise en place d'une connexion HTTPS sécurisée.

Cependant, Ingress ne fonctionne pas sous Windows, donc nous devons transférer tout le travail vers Linux.

- Synchronisation des données des Pod maîtres vers les Pod esclaves :

La synchronisation des données n'est pas quelque chose liée à Kubernetes, c'est purement une affaire PostgreSQL. Jusqu'à présent, notre configuration Kubernetes fonctionne bien mais sans synchronisation des données.

Pour son activation, nous devons configurer le Pod principal pour accepter les connexions de tous les hôtes via le port 5432, et par la suite créer les utilisateurs de réplication. Ensuite, nous configurons le fichier de configuration du Pod esclave pour spécifier l'adresse IP du Pod principal, le nom d'utilisateur et le mot de passe de l'utilisateur de réplication. Une fois ces configurations effectuées, le Pod esclave peut se connecter au Pod principal et commencer à répliquer les données, Mais avant cela, nous devons redémarrer le serveur pour que ces modifications soient appliquées. Cependant, dans mon cas, chaque fois que j'essaie de le redémarrer, le serveur plante, puis le Pod génère une erreur. Dans ce cas, Kubernetes le remplace par un nouveau Pod sain, ce qui détruit toutes les modifications que j'ai apportées.

Cette situation m'a poussé à chercher d'autres alternatives pour mettre cela en place.

4. Le projet ‘Kubegres’

Le processus de déploiement de PostgreSQL sur Kubernetes, suivi de l'ajout de la réplication, peut parfois être long, surtout lorsque des erreurs surviennent et que l'on perd beaucoup de temps à essayer de les résoudre. La solution alternative la plus simple est d'utiliser un projet PostgreSQL prêt à l'emploi, avec la réplication déjà configurée. Dans ce cas, en quelques minutes seulement, nous aurons PostgreSQL sur Kubernetes prêt à être utilisé comme base de données pour notre projet. Ce projet s'appelle "Kubegres".

Selon sa documentation officielle : ‘Kubegres est un opérateur Kubernetes open source permettant de déployer un cluster d'instances PostgreSQL avec réplication de données activée par défaut. Il apporte de la simplicité lors de l'utilisation de PostgreSQL, compte tenu de la complexité de la gestion du cycle de vie des ‘StatefulSets’ et de la réplication des données avec Kubernetes’

- Et il a les fonctionnalités suivantes :
  - **Création d'un cluster avec réplication des données activée** : Il crée un pod PostgreSQL principal et un certain nombre de pods PostgreSQL réplicas, répliquant la base de données du primaire en temps réel vers les pods réplicas.
  - **Gestion du fail-over** : Si le PostgreSQL principal tombe en panne, il promeut automatiquement un PostgreSQL réplica en tant que primaire.

- **Option de sauvegarde des données** : Il permet de sauvegarder régulièrement les données PostgreSQL dans un volume spécifié.
- **Configuration simplifiée** : Il fournit un fichier YAML très simple avec des propriétés spécialisées pour PostgreSQL.
- **Résilience et fiabilité** : Il est testé avec plus de 55 cas de tests automatisés et a été utilisé en production.
- **Compatibilité avec les conteneurs PostgreSQL officiels** : Il fonctionne avec les conteneurs PostgreSQL créés par l'équipe Docker Official Images, sans nécessiter d'image Docker personnalisée.

Pour plus d'informations sur 'Kubegres', ouvrez ce lien :

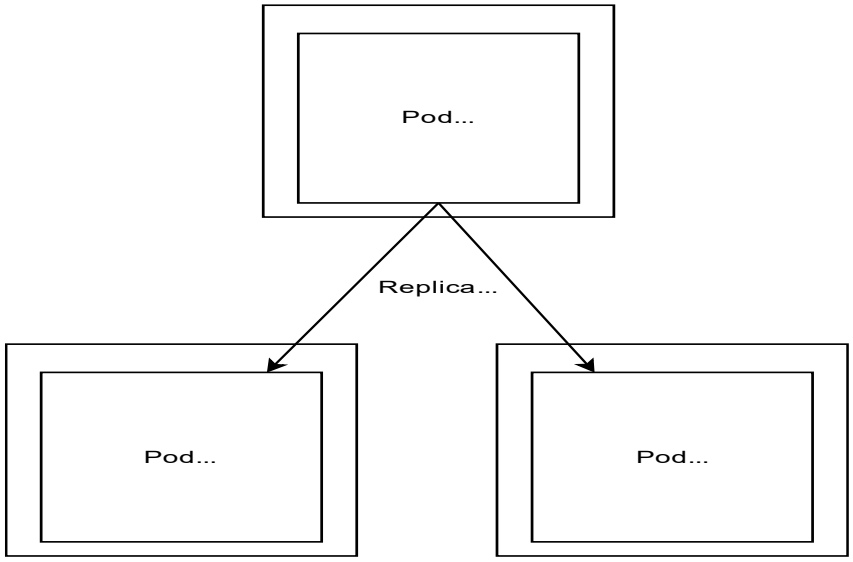
https://www.postgresql.org/about/news/kubegres-is-available-as-open-source-2197/

- L'utilisation de Kubegres :

Kubegres nous offre tout ce dont nous avons besoin pour déployer un PostgreSQL fonctionnel sur Kubernetes. J'ai donc suivi la documentation du projet pour créer la base de données.

Pour créer la base de données et les répliques, nous devons créer le fichier de déploiement où nous définissons le mot de passe de la base de données, le mot de passe de l'utilisateur de réplication, et la version de l'image PostgreSQL que nous souhaitons utiliser. J'ai utilisé le fichier par défaut dans la documentation, car nous voulons une base de données simple qui n'a pas besoin de modifications complexes.

Après avoir suivi le guide, nous aurons un Pod maître et deux répliques, chaque Pod fonctionnant sur son propre nœud de, comme l'image montre :



Viewer does not support full SVG 1.1

Nous pouvons afficher tout le travail effectué par Kubegres en exécutant la commande :

“kubectl get pod,statefulset,svc,pvc -o wide”

L’image ci-dessous montre le résultat de la commande :

```
PS F:\postgres_gei806> kubectl get pod,statefulset,svc,pvc -o wide
NAME                   READY   STATUS    RESTARTS      AGE   IP             NODE       NOMINATED NODE   READINESS GATES
pod/mypostgres-1-0     1/1     Running   2             12d   10.244.0.43    minikube   <none>           <none>
pod/mypostgres-3-0     1/1     Running   2 (11m ago)   12d   10.244.0.42    minikube   <none>           <none>
pod/mypostgres-4-0     1/1     Running   1 (11m ago)   12d   10.244.0.46    minikube   <none>           <none>

NAME                                READY   AGE   CONTAINERS     IMAGES
statefulset.apps/mypostgres-1       1/1     12d   mypostgres-1   postgres:16.1
statefulset.apps/mypostgres-3       1/1     12d   mypostgres-3   postgres:16.1
statefulset.apps/mypostgres-4       1/1     12d   mypostgres-4   postgres:16.1

NAME                                  TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE   SELECTOR
service/kubernetes                    ClusterIP   10.96.0.1      <none>        443/TCP    21d   <none>
service/mypostgres                    ClusterIP   None           <none>        5432/TCP   12d   app=mypostgres,replicationR
ole=primary
service/mypostgres-replica            ClusterIP   None           <none>        5432/TCP   12d   app=mypostgres,replicationR
ole=replica
service/postgres-reading-service      ClusterIP   10.111.60.73   <none>        5433/TCP   21d   app=postgres-gei806
service/postgres-service              ClusterIP   None           <none>        5432/TCP   21d   app=postgres-gei806

NAME                                                    STATUS   VOLUME                                     CAPACITY   A
CCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE   VOLUMEMODE
persistentvolumeclaim/mongo-data-mongo-gei806-0         Bound    pvc-d055a1dd-9baa-429a-904a-ed62f3b0d2c9   1Gi        R
WO            standard       <unset>                 14d   Filesystem
persistentvolumeclaim/mongo-data-mongo-gei806-1         Bound    pvc-cc47e88c-f26f-4ba5-884b-80482895129c   1Gi        R
WO            standard       <unset>                 14d   Filesystem
persistentvolumeclaim/postgres-data-postgres-gei806-0   Bound    pvc-c4e5276a-2503-46e6-92a4-e47fddff4b91   1Gi        R
WO            standard       <unset>                 16d   Filesystem
persistentvolumeclaim/postgres-data-postgres-gei806-1   Bound    pvc-a3a5e598-cc74-4d9e-89d0-70c50b2526ea   1Gi        R
WO            standard       <unset>                 16d   Filesystem
persistentvolumeclaim/postgres-db-mypostgres-1-0        Bound    pvc-e6abad58-ead1-41c0-bceb-9d55a3e729c7   200Mi      R
WO            standard       <unset>                 12d   Filesystem
persistentvolumeclaim/postgres-db-mypostgres-2-0        Bound    pvc-3324033a-3ee9-41e5-8fe2-208f13548a75   200Mi      R
WO            standard       <unset>                 12d   Filesystem
persistentvolumeclaim/postgres-db-mypostgres-3-0        Bound    pvc-23c4ce5c-00da-408a-8e15-188defec57a5   200Mi      R
WO            standard       <unset>                 12d   Filesystem
persistentvolumeclaim/postgres-db-mypostgres-4-0        Bound    pvc-bf8546ce-04da-4e11-ae82-5354570c9b04   200Mi      R
```

Ici, nous observons que Kubegres a créé trois Pods : mypostgres-1-0, mypostgres-3-0, mypostgres-4-0, et trois StatefulSets (nœuds) : mypostgres-1, mypostgres-3, mypostgres-4. Il a également créé deux services : mypostgres et mypostgres-replicas. De plus, pour chaque Pod, il a créé son propre volume.

- Connexion aux répliques et aux Pods maîtres :

Maintenant, nous avons la base de données PostgreSQL complète prête à accepter les connexions. Il nous suffit de l'exposer à notre machine locale. J'en ai parlé dans une section précédente. De la même manière, Kubegres crée deux services pour les connexions :

- "mypostgres" pour se connecter au pod maître, pouvant être utilisé pour les requêtes de lecture et d'écriture.
- "mypostgres-replica" est utilisé uniquement pour les requêtes de lecture.

Les commandes suivantes sont celles que j'ai utilisées pour les exposer à ma machine locale :

```
PS F:\postgres_gei806> kubectl port-forward service/mypostgres-replica 5433:5432
Forwarding from 127.0.0.1:5433 -> 5432
Forwarding from [::1]:5433 -> 5432
```

```
PS F:\postgres_gei806> kubectl port-forward service/mypostgres 5432:5432
Forwarding from 127.0.0.1:5432 -> 5432
Forwarding from [::1]:5432 -> 5432
```

5. Le côté serveur :

Après avoir parlé du premier composant, la "base de données", nous abordons le deuxième composant, le serveur, qui agit comme intermédiaire entre la base de données et l'interface utilisateur. Ce serveur est codé en Python en utilisant le Framework Flask, dédié à la création de serveurs et d'API REST.

Avec Flask, j'ai créé cinq points de terminaison :

- "tasks" pour afficher les tâches présentes dans la base de données.
- "user" pour afficher les tâches d'un utilisateur spécifique.
- "delete_tasks" pour supprimer une tâche spécifique en utilisant son ID.
- "new_task" pour enregistrer une nouvelle tâche dans la base de données.

- "new_fortune" pour enregistrer l'ID du type de tâche "fortune". Cela nous permet de pouvoir supprimer toutes les fortunes dans la base de données des tâches générées par l'auto-générateur.
- "delete_fortune" pour supprimer toutes les tâches de type "fortune" générées par l'auto-générateur.

Concernant les connexions à la base de données, j'ai créé deux connexions : une pour la lecture et l'autre pour les requêtes d'écriture. Les points de terminaison, comme "new_task", "new_fortune", ou "delete_fortune", nécessitent une connexion d'écriture. En revanche, pour afficher les tâches ou les tâches d'un utilisateur spécifique, nous utilisons une connexion de lecture.

L'image montre les deux connexions :

```
reading_db_connection = psycopg2.connect(host="localhost",
                                         database="postgres", user="postgres", password="postgresSuperUserPsw",
                                         port="5433")
##postgresReplicaPsw
writing_db_connection = psycopg2.connect(host="localhost",
                                         database="postgres", user="postgres", password="postgresSuperUserPsw",
                                         port="5432")
```

Ici, j'ai créé une connexion à la base de données que j'ai exposée à ma machine locale : une connexion de lecture au port 5433 pour se connecter directement au service de réplication, qui va rediriger la requête vers une réplique, et une connexion d'écriture au port 5432 pour se connecter au service maître, qui va rediriger la requête vers l'instance maître.

Voici une explication du point de terminaison "new_task" :

```python
@app.post("/v1/new_task")
def create_task():
    try:
        data = request.get_json()
        description = data.get('description')
        username = data.get('username')
        if not description:
            return jsonify({'error': 'Description is required'}), 400
        if not description:
            return jsonify({'error': 'User name is required'}), 400

        with writing_db_connection:
            with writing_db_connection.cursor() as db_cur:
                db_cur.execute(CREATE_TASKS_TABLE)
                query = "INSERT INTO tasks (description, username) VALUES (%s, %s);"
                values = (description, username)
                db_cur.execute(query, values)

        return jsonify({'message': 'Task created successfully'}), 201
    except Exception as e:
        print(f" On peut pas ajouter la tache, revien demain || error : {e}")
        return jsonify({'error': 'Failed to create task'}), 500
```

La première chose est que nous prenons le nom d'utilisateur et le texte de la tâche saisi par l'utilisateur. Ensuite, nous vérifions si l'un d'eux est vide. Si c'est le cas, nous renvoyons une erreur. Sinon, nous utilisons la connexion d'écriture pour les ajouter à la base de données. Nous utilisons des requêtes SQL pré-préparées pour éviter le problème d'injection SQL (nous ne pouvons pas injecter les valeurs directement dans la requête).

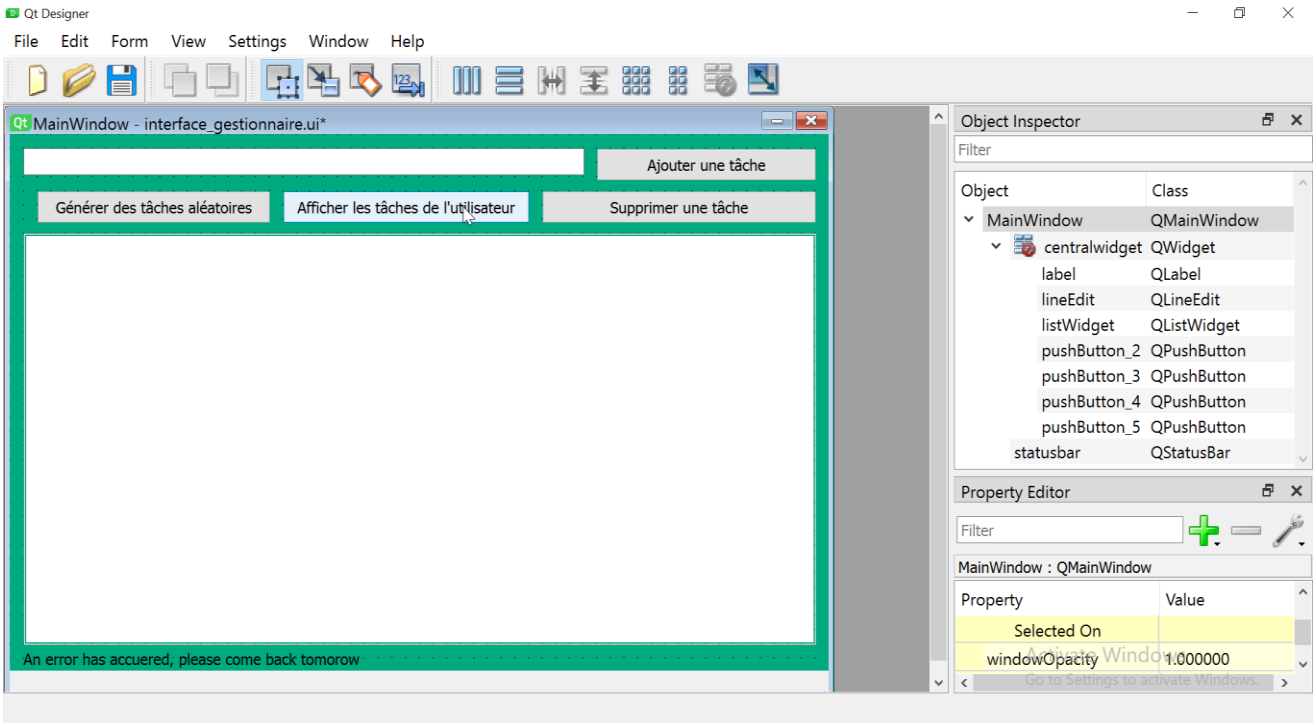
6. L'interface d'utilisateur :

Maintenant, nous sommes arrivés au dernier composant de notre application : l'interface utilisateur. C'est ici que l'utilisateur interagit avec l'application, où il peut afficher toutes les tâches et les supprimer en un clic.

L'interface utilisateur est conçue avec la bibliothèque PyQt5, avec l'aide de Qt Designer. Je parlerai ci-dessous en détail des étapes de conception.

Tout d'abord, nous avons besoin de deux applications : une pour le gestionnaire d'applications, "l'administrateur", et l'autre pour "le client". Le gestionnaire a plus de privilèges que le client. Le client peut uniquement ajouter, supprimer et afficher ses tâches. Cependant, le gestionnaire peut voir les tâches des autres utilisateurs et, de plus, il peut également apporter des modifications à leurs données.

Afin de créer l'interface, j'ai utilisé Qt Designer, qui est un outil très utile. Vous pouvez simplement glisser-déposer des éléments sur votre interface utilisateur. Une fois terminé, vous transformez le fichier en code Python et apportez vos modifications ou ajustements à la mise en page. L'image ci-dessous montre Qt Designer :



Dans mon cas, j'ai créé une seule interface en utilisant Qt Designer, car les autres écrans sont similaires. Par exemple, pour le client, je n'ai besoin de conserver que deux boutons : les boutons "Supprimer" et "Ajouter". De la même manière, j'ai créé les autres écrans en supprimant ou en ajoutant simplement des morceaux de code.

Après cela, je l'ai lié au serveur. Ainsi, lorsque l'utilisateur clique sur un bouton, une requête HTTP est envoyée au point de terminaison approprié, et le serveur renvoie la réponse. Voici un exemple ci-dessous :

```python
def add_task(self):
    item = self.lineEdit.text()
    if item.strip():
        try:
            self.label.setText("")
            self.label.setText("Adding new task ........")
            new_task = {'description': item,
                        'username': "Gestionnaire"}
            response = requests.post( url: 'http://localhost:5000/v1/new_task', json=new_task)
            response.raise_for_status()

            if response.status_code == 201:
                self.label.setText("New task added ........")
                self.load_tasks()
                self.lineEdit.setText("")

        except requests.exceptions.RequestException as e:
            self.label.setText("error adding task ........")
            print(f"Error adding task: {e}")
    else:
        self.label.setText("Can't add empty task ........")
        print("empty task")
```

L'image montre le code pour ajouter une nouvelle tâche à la base de données. Tout d'abord, nous enregistrons le texte que l'utilisateur a saisi dans une variable appelée "item". Ensuite, nous vérifions si le texte n'est pas vide. Nous envoyons une requête HTTP à ‘http://localhost:5000/v1/new_task’, qui correspond au point de terminaison d'ajout de tâches, en incluant le texte de l'utilisateur et le nom d'utilisateur, ici "Gestionnaire".

Lorsque le serveur reçoit la requête, il se connecte à la base de données, ajoute les données, puis renvoie une réponse. Si la réponse est réussie, cela signifie que les tâches ont été ajoutées avec succès et nous pouvons mettre à jour l'interface utilisateur. En revanche, si le serveur renvoie une exception, nous informerons l'utilisateur qu'il y a eu une erreur.

Un autre exemple, que je trouve très important à expliquer, consiste à ajouter des tâches aléatoires à partir d'une base de données de fortunes :

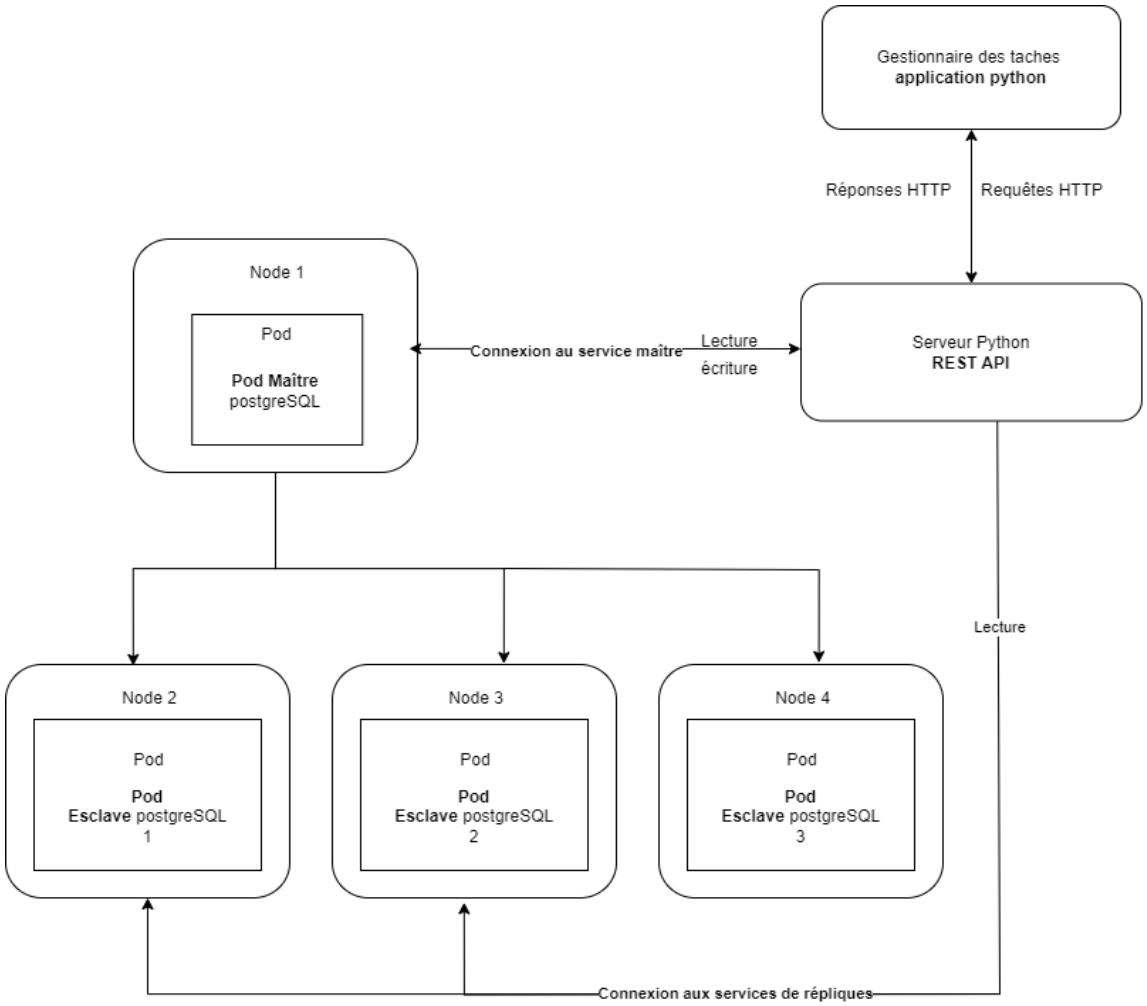
```python
def generate_random_tasks(self):
    try:
        response = requests.delete('http://localhost:5000/v1/delete_fortune')
        response.raise_for_status()
        if response.status_code == 200:
            print("Fortunes are deleted....ready to add new ones")
            print("starting new seprate thread to add fortune")
            self.running = True
            self.fortune_thread = threading.Thread(target=self.addFortureToDatabase)
            self.fortune_thread.daemon = True
            self.fortune_thread.start()
    except requests.exceptions.RequestException as e:
        print(f"Error deleting fortunes: {e}")
        return False
```

Les tâches d'ajout de tâches toutes les cinq secondes doivent être exécutées sur un autre thread afin de ne pas bloquer le thread principal (le thread d'interface utilisateur).

La première étape consiste à supprimer toutes les tâches ajoutées, générées auparavant par cette fonction. Nous envoyons donc, une requête DELETE à http://localhost:5000/v1/delete_fortune pour nettoyer toutes les fortunes de la base de données des tâches. Si le serveur renvoie une réponse positive, nous exécutons la fonction addFortuneToDatabase() pour envoyer une requête au serveur afin d'ajouter une fortune, de la même manière que l'exemple précédent, mais après avoir récupéré aléatoirement un texte de fortune dans la base de données de fortunes (fortunes_.txt).

De cette façon, nous pouvons continuer à interagir avec l'application en ajoutant ou en supprimant des éléments de la base de données, car l'autre processus s'exécute sur un thread séparé. Pendant ce temps, nous interagissons avec le thread principal.

7. L’architecture complète de projet :



Cette architecture assemble les trois composants de votre système :

- **Base de données** : En utilisant Kubegres, nous avons mis en place une base de données PostgreSQL distribuée et répliquée.
- **Serveur d'application** : Codé en Python avec Flask, le serveur gère les interactions entre la base de données et l'interface utilisateur.
- **Interface utilisateur** : Conçue avec PyQt5, l'interface utilisateur permet aux utilisateurs d'interagir facilement avec le système, d'ajouter et de supprimer des tâches, et de voir leurs données en temps réel.

8. Comment on peut évoluer le system ?

Le système que j'ai créé fonctionne très bien, mais il n'est pas adapté à un environnement de production, car mon ordinateur n'est pas capable de gérer trop de demandes. Nous devons donc, transférer ce système vers un environnement de production et augmenter sa disponibilité. Nous pouvons le faire en suivant ces étapes :

1. Comme nous avons créé de nombreuses instances pour la base de données, nous en créerons d'autres pour l'application et le serveur. Nous pouvons créer une image Docker pour le serveur et l'application, puis les déployer dans le même cluster Kubernetes que la base de données. De cette façon, Kubernetes s'occupera de la disponibilité.
2. Comme tous les composants du système fonctionneront sur le même cluster, nous n'aurons pas besoin d'exposer la base de données à notre machine locale. Nous utiliserons plutôt le service Ingress, où nous créerons des URL HTTPS pour les connexions à la base de données. Par exemple : http://mypostgres.readingdatabase.com ou http://mypostgres.com/reading
   Ainsi, lorsque nous nous connectons à l'une de ces URL, nous nous connectons à la base de données de lecture.
3. Comme nous allons utiliser le service d'Ingress, nous devons déplacer le système vers Ubuntu, car le contrôleur d'Ingress ne fonctionne que dans un environnement Linux. Nous devons donc utiliser un système Linux exécuté sur un serveur à haute capacité, capable de gérer plusieurs requêtes.

IV. Conclusion :

Le système que j'ai créé représente un exemple de la façon de concevoir une application de niveau industriel, capable de supporter de nombreux utilisateurs. Il fournit également une solution pour les applications existantes afin d'augmenter leur disponibilité et de résoudre les problèmes liés à la scalabilité. De plus, il illustre comment déployer une base de données dans Kubernetes.

Notre architecture système, peut servir de modèle à de nombreux autres développeurs. Elle montre comment gérer efficacement les composants backend et frontend d'une application, comment assurer une haute disponibilité et comment tirer parti des fonctionnalités de Kubernetes pour le déploiement et la gestion des services.