

UNIVERSITÉ DE SHERBROOKE

Faculté de génie

Département de génie électrique et génie informatique

## **RAPPORT APP 1**

Systèmes informatiques répartis

GIF 391

Présenté à :

Palao Munoz, Domingo

Présenté par :

Équipe numéro 4

Bendjeddou Mohamed El Hadi -

Sherbrooke - 04 Septembre

# Table des matières

<b>Introduction</b>	<b>3</b>
<b>Développement</b>	<b>3</b>
Étape 1	3
Étape 2	4
Étape 3	5
Étape 4	6
Étape 5	8
Étape 6	10
Discussion de la Structure, les Avantages et les défis du Système	10
<b>Conclusion</b>	<b>11</b>
<b>Références</b>	<b>12</b>

# Introduction

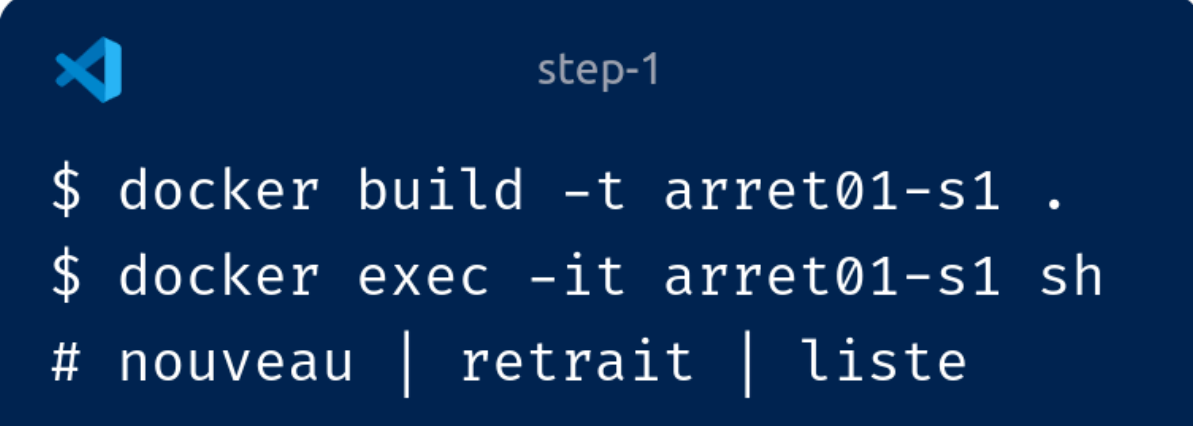
Le projet aborde la problématique de la migration d'une application "To-Do" monolithique et centralisée, conçue pour un utilisateur unique et un serveur unique (échelle réduite), vers une architecture distribuée, multi-utilisateur et multi-serveur, qui interagit avec des instances de bases de données réparties. Cette transformation est particulièrement pertinente dans un contexte où l'évolutivité, la fiabilité et la répartition des charges sont des préoccupations majeures. Ce rapport détaillera les étapes suivies pour mettre en place ce système, les outils utilisés, et les choix technologiques effectués pour parvenir à une version fonctionnelle.

## Développement

La solution mise en œuvre adopte une approche itérative, permettant à l'application "To-Do" d'évoluer et répondre aux exigences fonctionnelles (ce que le système doit accomplir), non fonctionnelles (les qualités et performances attendues du système), et techniques (les aspects technologiques nécessaires à sa réalisation), telles que spécifiées dans le guide de l'étudiant. Ce rapport examine cette transition étape par étape, en détaillant les modifications apportées, les justifications correspondantes, ainsi que les problèmes rencontrés ou potentiels, le cas échéant.

### Étape 1

Cette étape **préliminaire** consiste à compiler une version de base du système qui inclut le composant serveur pour pouvoir après l'accéder et exécuter les commandes de base implémentées par des scripts python et les invoker à travers l'exécution en mode itérative d'un shell linux depuis le conteneur qui est en exécution, tel que indiqué dans la figure 01 ci-dessus



```
step-1  
$ docker build -t arret01-s1 .  
$ docker exec -it arret01-s1 sh  
# nouveau | retrait | liste
```

Figure 1: compiler le serveur et exécuter les scripts  
(\*Le Dockerfile n'est pas été modifié)

Les commandes sont auto-descriptives :

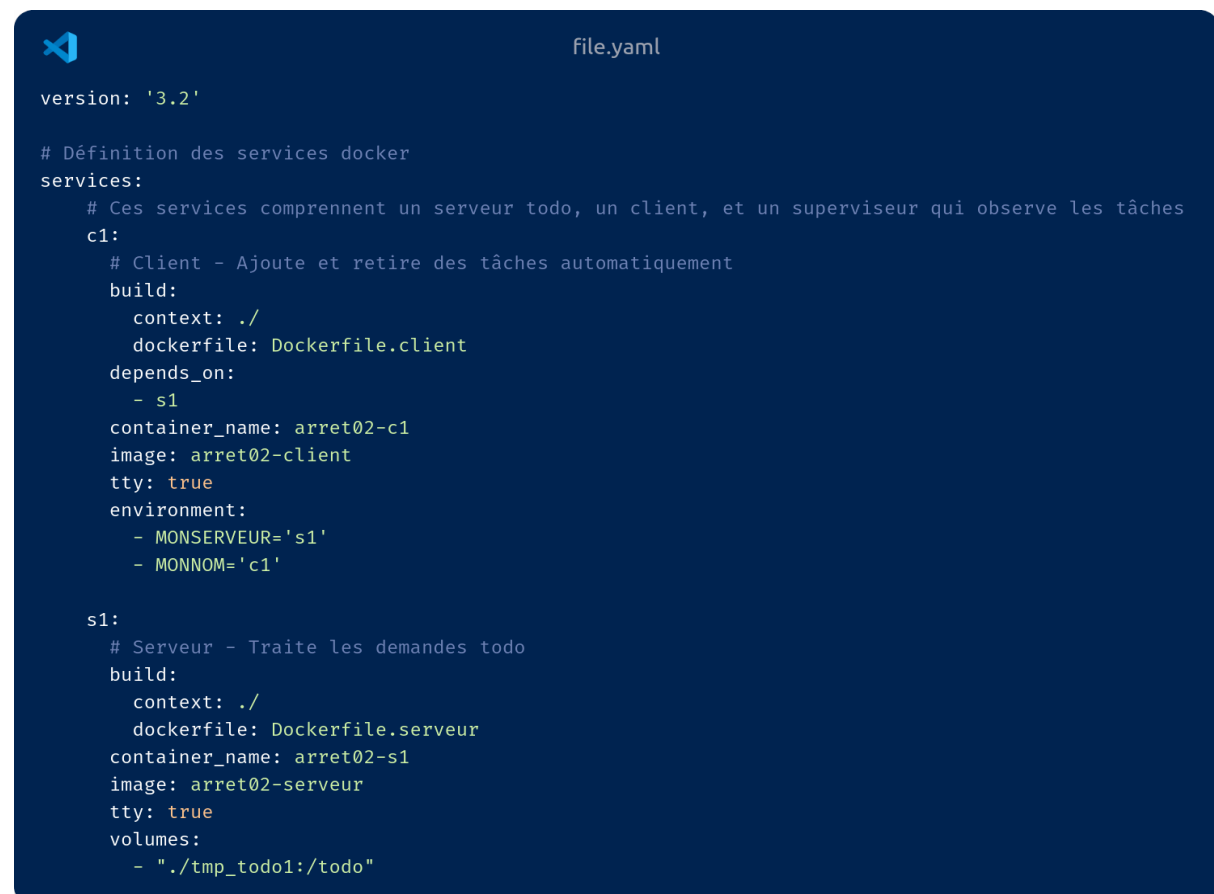
- **nouveau** : Permet d'ouvrir le fichier "todo.txt" en mode écriture et d'ajouter les tâches passées en tant que second argument au shell, avec la définition le nom de l'agent responsable de ces écritures.

- **retrait** : Permet de retirer des tâches du fichier, ou un ensemble de tâches, selon le paramètre N passé au shell.

- **liste** : Permet d'afficher toutes les entrées dans le fichier texte, ou uniquement celles spécifiques à un agent.

## Étape 2

Dans cette étape, le but est d'établir une communication entre les composants client et serveur, lui permettant ainsi d'envoyer des requêtes périodiques à ce dernier (faire infiniment des insertions et des retrait aléatoires séparés par un délai passer en argument). Avant cela, il est nécessaire de **modifier le fichier 'docker-compose'** en ajoutant les sources de construction pour les images Docker, en configurant la dépendance du client sur le serveur, et en définissant les variables d'environnement permettant au client de s'authentifier avant d'envoyer ses requêtes, tel que indiqué par la figure 2 ci-dessus.

A screenshot of a code editor with a dark blue background. The editor shows a Docker Compose file named 'file.yaml'. The file contains YAML configuration for two services: 'c1' (client) and 's1' (server). The 'c1' service depends on 's1' and has environment variables 'MONSERVEUR' and 'MONNOM'. The 's1' service has a volume mount for 'tmp\_todo1/todo'.

```
version: '3.2'

# Définition des services docker
services:
  # Ces services comprennent un serveur todo, un client, et un superviseur qui observe les tâches
  c1:
    # Client - Ajoute et retire des tâches automatiquement
    build:
      context: ./
      dockerfile: Dockerfile.client
    depends_on:
      - s1
    container_name: arret02-c1
    image: arret02-client
    tty: true
    environment:
      - MONSERVEUR='s1'
      - MONNOM='c1'

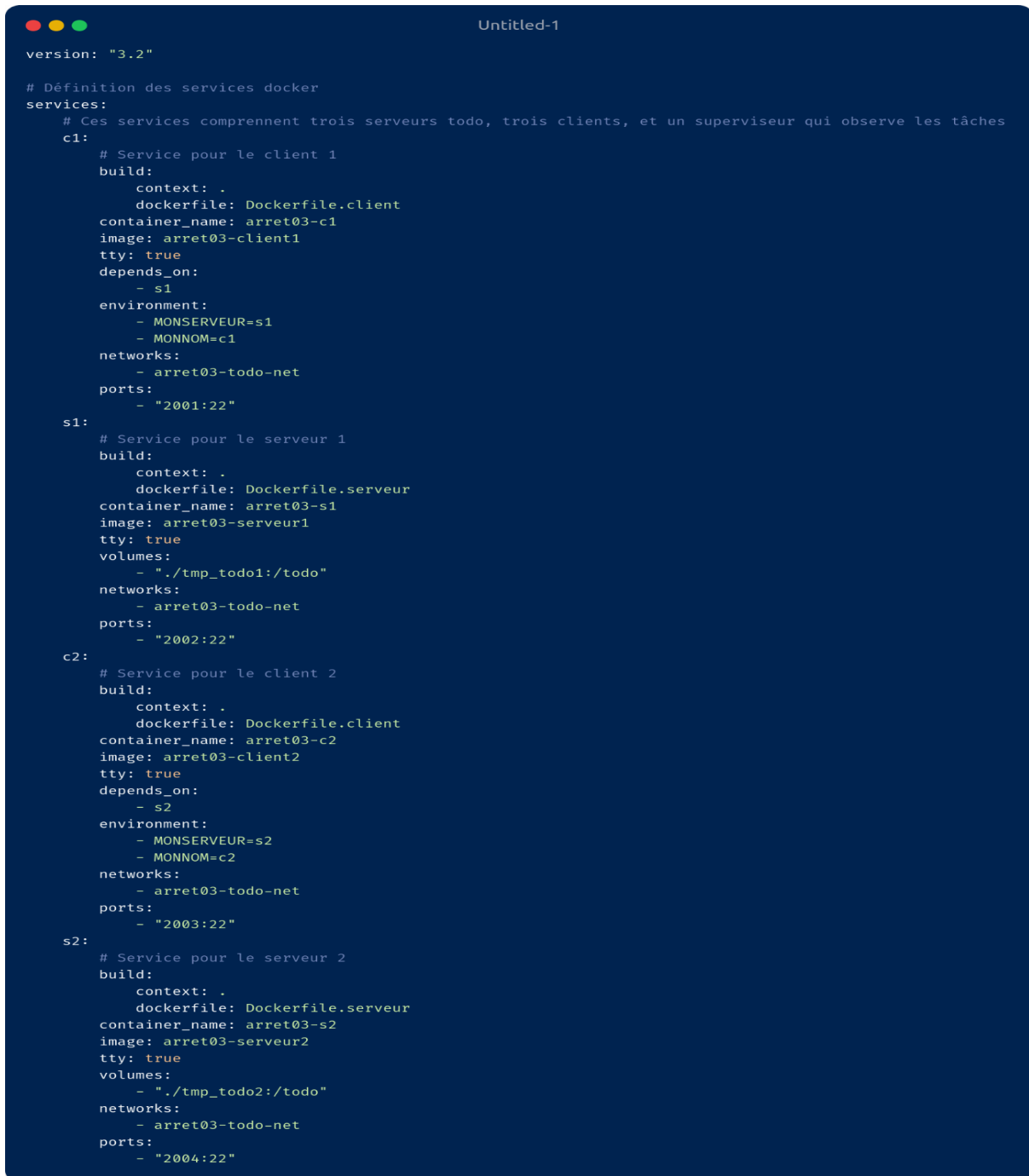
  s1:
    # Serveur - Traite les demandes todo
    build:
      context: ./
      dockerfile: Dockerfile.serveur
    container_name: arret02-s1
    image: arret02-serveur
    tty: true
    volumes:
      - "./tmp_todo1:/todo"
```

Figure 2: fichier docker compose pour le pair client-server arrêt 2

\*La communication inter-processus est établie via le **ssh** (Secure Shell Protocol), et les demandes du client sont servies par l'appel **subprocess.run()**, dans une boucle infinie que le serveur exécute en permanence, ce qui le surcharge et le rend non réactif !

## Étape 3

À cet arrêt, l'objectif est d'élargir le système pour qu'il prenne en charge l'utilisation indépendante par trois développeurs, en permettant au gestionnaire d'accéder en lecture à toutes les données. Il est nécessaire de réutiliser les images Docker existantes pour établir trois paires client-serveur distinctes, ainsi qu'une image pour le gestionnaire. Le fichier **'docker-compose' doit être modifié** pour gérer ces services, en configurant correctement les dépendances, les variables d'environnement et les accès nécessaires, tel que indique par la figure 3 ci-dessus.

A screenshot of a terminal window with a dark blue background and white text. The window title is 'Untitled-1'. It displays a Docker Compose file for version '3.2'. The file defines four services: 'c1' (client 1), 's1' (server 1), 'c2' (client 2), and 's2' (server 2). Each service is built from a local Dockerfile and runs on the 'arret03-todo-net' network. 'c1' and 'c2' are clients that depend on their respective servers ('s1' and 's2'). 's1' and 's2' are servers that mount a local volume ('./tmp\_todo1/todo' and './tmp\_todo2/todo' respectively) and expose port 22. The environment variables 'MONSERVEUR' and 'MONNOM' are set for each service to match their names (e.g., 's1' for 'MONSERVEUR=s1').

```
version: "3.2"

# Définition des services docker
services:
  # Ces services comprennent trois serveurs todo, trois clients, et un superviseur qui observe les tâches
  c1:
    # Service pour le client 1
    build:
      context: .
      dockerfile: Dockerfile.client
    container_name: arret03-c1
    image: arret03-client1
    tty: true
    depends_on:
      - s1
    environment:
      - MONSERVEUR=s1
      - MONNOM=c1
    networks:
      - arret03-todo-net
    ports:
      - "2001:22"

  s1:
    # Service pour le serveur 1
    build:
      context: .
      dockerfile: Dockerfile.serveur
    container_name: arret03-s1
    image: arret03-serveur1
    tty: true
    volumes:
      - "./tmp_todo1:/todo"
    networks:
      - arret03-todo-net
    ports:
      - "2002:22"

  c2:
    # Service pour le client 2
    build:
      context: .
      dockerfile: Dockerfile.client
    container_name: arret03-c2
    image: arret03-client2
    tty: true
    depends_on:
      - s2
    environment:
      - MONSERVEUR=s2
      - MONNOM=c2
    networks:
      - arret03-todo-net
    ports:
      - "2003:22"

  s2:
    # Service pour le serveur 2
    build:
      context: .
      dockerfile: Dockerfile.serveur
    container_name: arret03-s2
    image: arret03-serveur2
    tty: true
    volumes:
      - "./tmp_todo2:/todo"
    networks:
      - arret03-todo-net
    ports:
      - "2004:22"
```

Figure 3: fichier docker compose pour les pair client-server arrêt 3  
(\*Les autres services ont été omis en raison de la taille de l'image)

Le principal problème de conception dans ce scénario est lié à la duplication inutile des ressources et à une mauvaise évolutivité à savoir :

1. **Duplication des ressources** : Créer trois paires distinctes client-serveur pour chaque développeur entraîne une duplication des services et des ressources, ce qui n'est pas nécessairement efficace
2. **Scalabilité limitée** : Cette approche n'est pas scalable. Si l'équipe de développeurs grandit, le système nécessiterait la création d'une nouvelle paire client-serveur pour chaque nouveau développeur, augmentant la complexité
3. **Manque de coordination** : L'architecture adoptée rend difficile la gestion des données et complique le partage des données entre les développeurs
4. **Problèmes de cohérence des données** : Avec des serveurs distincts pour chaque développeur et un gestionnaire accédant aux trois serveurs, il y a un risque accru de lectures inconsistantes (phénomènes de lecture, par exemple: en lisant un enregistrement juste avant qu'il ne soit supprimé par un client, ou en ne parvenant pas à lire un enregistrement qui vient d'être ajouté par un client). Cela peut entraîner des incohérences dans les données visibles par le gestionnaire

## Étape 4

L'étape 4 consiste à modifier l'architecture existante pour qu'elle repose sur un seul serveur partagé desservant trois développeurs clients (c1, c2, c3), avec un gestionnaire en lecture seule au serveur. Cette étape vise à tester la faisabilité de centraliser les opérations sur une ressource partagée, en réduisant la duplication des serveurs (simplifiant l'infrastructure), et en maintenant l'accès multi-utilisateurs au système de gestion des tâches. Mais, en réalité, cette étape présente des défis significatifs en termes de gestion de la concurrence et de synchronisation.

En effet, il est important de considérer

1. **Concurrence et Synchronisation** : Si plusieurs développeurs tentent d'accéder ou de modifier le fichier `todo.txt` simultanément, cela peut entraîner des conflits, des pertes de données, ou des résultats imprévisibles, car le serveur ne garantit pas l'atomicité des opérations, les opérations critiques (comme l'ajout ou la suppression d'une tâche) peuvent être interrompues ou exécuté partiellement, laissant le fichier dans un état incohérent
2. **Performance et Scalabilité** : Un seul serveur pour gérer les requêtes de plusieurs développeurs peut devenir un bottleneck<sup>1</sup>, réduisant les performances globales du système. À mesure que le nombre de développeurs augmente, le serveur unique pourrait ne pas être en mesure de traiter efficacement toutes les requêtes, ce qui compromettrait la réactivité et l'efficacité du système
3. **Complexité de Mise en Œuvre** : Assurer la cohérence des données et éviter les conflits pour plusieurs clients exige une implémentation complexe (incluant des verrous, des transactions, ou d'autres mécanismes)

---

<sup>1</sup> un bottleneck se produit lorsque la capacité et le flux d'une application sont restreintes par un seul composant

Néanmoins, pour y arriver il faut **modifier le 'docker-compose'** et enlever les 2 autres serveurs, corriger les dépendances, ainsi que utiliser un **mécanisme de synchronisation** dans les fichiers `todo.py` et `callTodo.py`. La solution présentée utilise la bibliothèque `fcntl` (**Kerrisk**) en Python qui fournit des interfaces pour les appels système POSIX, principalement utilisés pour manipuler les descripteurs de fichiers sur les systèmes et dans ce cas pour gérer des verrous de fichiers. Ce comportement est implémenté par la class `Locker` qui expose deux méthodes **`acquire()`** et **`release()`** qui seront utilisés dès les opérations I/O dans le code, tel qu'indiqué dans les figures 4 et 5 ci-dessus.

```
locker.py

# locker.py

import fcntl

class Locker:
    """
    Class for file locking using fcntl module.
    Attributes:
        file: A file object to be locked.
    Methods:
        acquire: Acquires an exclusive lock on the file.
        release: Releases the lock on the file.
    """
    def __init__(self, file):
        self.file = file

    def acquire(self):
        fcntl.flock(self.file.fileno(), fcntl.LOCK_EX)

    def release(self):
        fcntl.flock(self.file.fileno(), fcntl.LOCK_UN)
```

Figure 4: La class `Locker`

```
Untitled-1

import argparse
from datetime import date, datetime
import subprocess
import re
import os
from locker import Locker

class Todo:
    # Previous code omitted
    def add_todo(self, new_msg, user):
        with open(self.TODO, 'a+', encoding='utf-8') as f:
            lock = Locker(f)
            lock.acquire()
            today = date.today()
            now = datetime.now()
            current_time = now.strftime("%H:%M:%S")
            f.write(f"TODO<{self.todo_num}> : [{user}], ({today}:{current_time}) : {new_msg}\n")
            self.increment_latest_todo_num()
            lock.release()
```

Figure 5: L'utilisation des locks pour les I/O

## Étape 5

À ce stade, afin de garantir l'atomicité des requêtes, le serveur intègre une instance de l'image PostgreSQL. La tâche maintenant consiste à combiner les trois clients, le gestionnaire, le serveur et la base de données PostgreSQL dans un fichier docker-compose fonctionnel. Les images du client, serveur et gestionnaire dépendent maintenant d'une image **todo-base** qui doit être construite **avant**. Il faut aussi ajuster les **variable d'environnement** qui configurent la connexion avec la base de donnée (notamment DB\_NAME, DB\_USER, DB\_PASS), et de corriger ses **mauvais directoires** pour les volumes montées, tel qu'indiqué dans la figure 6 et 7 ci-dessus.



```
version: "3.2"

services:
  todo-base:
    build:
      context: .
      dockerfile: Dockerfile.todo-base
    container_name: arret05-todo-base
    image: todo-base
  s1:
    build:
      context: .
      dockerfile: Dockerfile.serveur
    container_name: arret05-s1
    image: arret05-serveur1
    tty: true
    networks:
      - arret05-todo-net
    ports:
      - "2002:22"
    environment:
      - DB_HOST=db
      - DB_PORT=5432
      - DB_NAME=postgres
      - DB_USER=postgres
      - DB_PASS=postgres
    depends_on:
      - todo-base
      - db
```

Figure 6: fichier docker compose pour le server arrêt 5



```
db:
  container_name: arret05-db
  image: postgres:15
  environment:
    - POSTGRES_PASSWORD=postgres
    - POSTGRES_USER=postgres
    - POSTGRES_DB=postgres
  volumes:
    - todo-db:/var/lib/postgresql/data
    - ./init_db/sql:/docker-entrypoint-initdb.d/
  networks:
    - arret05-todo-net
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres"]
    interval: 10s
    timeout: 5s
    retries: 5
  expose:
    - "5432:5432"

networks:
  arret05-todo-net:

volumes:
  todo-db:
```

Figure 7: fichier docker compose pour postgres arrêt 5

Untitled-2					
id	td_user	init_time	last_update_time	done_time	task
1		2024-09-03 08:15:34.859796+00	2024-09-03 08:15:34.859796+00		The fashion wears out more apparel than the man. – William Shakespeare
2		2024-09-03 08:15:34.981635+00	2024-09-03 08:15:34.981635+00		There is a time and a place for everything: 8 o'clock, my place, everything.
3		2024-09-03 08:15:35.051598+00	2024-09-03 08:15:35.051598+00		Never stand between a fire hydrant and a dog.
4		2024-09-03 08:15:35.389346+00	2024-09-03 08:15:35.389346+00		Firmness of delivery dates is inversely proportional to tightness of schedule.
5		2024-09-03 08:15:35.542657+00	2024-09-03 08:15:35.542657+00		Love is a much stronger drug than LSD. – Ram Dass
6		2024-09-03 08:15:35.63893+00	2024-09-03 08:15:35.63893+00		A sentence is worth a thousand words.
7		2024-09-03 08:15:35.906248+00	2024-09-03 08:15:35.906248+00		All is not gold that glitters.
8		2024-09-03 08:15:36.131925+00	2024-09-03 08:15:36.131925+00		All people are born alike – except Republicans and Democrats. – Groucho Marx
9		2024-09-03 08:15:36.214832+00	2024-09-03 08:15:36.214832+00		Your body is the greatest instrument you'll ever own. – Mary Schmich
10		2024-09-03 08:15:36.423279+00	2024-09-03 08:15:36.423279+00		Reality often astonishes theory. – Tom Magliozzi

Figure 8: résultat d'une requête dans le conteneur du postgres (SELECT\* FROM todo.element LIMIT 10)

## Étape 6

Finalement, pour faire évoluer le système (1000 développeurs) il faut dupliquer la base de données et la répartir en instances pour éviter la congestion. Le déploiement se fera dans des pods Kubernetes pour tirer parti de multiples machines, avec l'utilisation de Minikube pour simplifier la configuration et la gestion des environnements Kubernetes. **Cependant**, malgré ces préparations, la mise à l'échelle à grande a rencontré des difficultés significatives liées à la configuration de Kubernetes et à la gestion des pods. Ces défis empêchent la réalisation complète du déploiement à grande échelle.

## Discussion de la Structure, les Avantages et les défis du Système

### Quelle est l'architecture logicielle du système ? et pourquoi ?

L'architecture logicielle mise en place repose sur une approche modulaire et distribuée. Chaque composant, qu'il s'agisse des clients, des serveurs ou des bases de données, est encapsulé dans un **conteneur** docker distinct. Cela garantit une isolation stricte des environnements d'exécution et facilite la gestion des dépendances, tout en permettant une évolutivité accrue. L'intégration de **l'orchestration** des conteneurs, notamment à travers Kubernetes, est essentielle pour maximiser leurs avantages. Elle facilite l'automatisation du déploiement, de la mise à l'échelle, et de la haute disponibilité, tout en optimisant la gestion des instances des modules. Cette approche renforce la fiabilité du système et assure une disponibilité élevée dans les environnements critiques.

### Quelles technologies Linux sous-jacentes jouent un rôle fondamental dans la mise en œuvre de cette architecture ?

- Les **namespaces** assurent l'isolation des processus, des utilisateurs, des réseaux et des systèmes de fichiers, permettant ainsi de gérer plusieurs conteneurs indépendamment en garantissant leur sécurité et leur séparation
- Les **control groups** (cgroups) gèrent et limitent l'utilisation des ressources telles que le CPU et la mémoire, garantissant une allocation équitable dans les environnements multi-conteneurs
- Le **système de fichiers à union**, tel que **OverlayFS** qui est au cœur de la gestion du stockage dans Docker en raison de sa **compatibilité** avec le noyau Linux est essentiel pour superposer plusieurs couches de fichiers, offrant une gestion efficace de l'espace disque, un mécanisme de mise à jour incrémentale rapide, et une persistance optimale des données des conteneurs.

### Quel pilote de persistance a été utilisé et pourquoi ?

Le pilote de persistance utilisé est PostgreSQL, une base de données relationnelle conforme aux principes ACID (Atomicité, Cohérence, Isolation, Durabilité). La combinaison de la conformité ACID de PostgreSQL et de la répartition en plusieurs instances constitue une solution robuste pour gérer des environnements de bases de données complexes

## La duplication des ressources est-elle bonne ou pas ?

Oui, c'est un élément clé car elle permet la **disponibilité** à cause de la **redondance**. Elle contribue également à l'augmentation de la capacité de lecture ("What Is Data Replication?"), ce qui permet une meilleure localisation des données (transparence de localisation) et soutient la reprise après l'échec (transparence d'échec)

## C'est quoi la configuration réseau permettant aux différents conteneurs d'interagir ?

Dans Docker, la configuration réseau permet de créer des **réseaux virtuels** isolés pour les conteneurs, garantissant ainsi que seuls les conteneurs faisant partie du même réseau peuvent communiquer directement. Il utilise des **réseaux de pont** pour connecter les conteneurs sur une même machine hôte, et des **réseaux overlay** pour les communications entre conteneurs situés sur différentes machines hôtes ou dans un cluster (via des fonctionnalités tels que les IP assignées dynamiquement, des noms de services résolubles, gestion des ports exposés, etc.)

## Dans quel cas la duplication peut se faire sur une machine réelle unique, dans quel cas elle doit être distribuée sur plusieurs machines réelles ?

Alors que la duplication peut être effectuée sur une seule machine pour des systèmes à petite échelle, les environnements de grande échelle nécessitent une distribution sur plusieurs machines physiques pour assurer une haute disponibilité et éviter les bottlenecks.

# Conclusion

La migration de l'application "To-Do" a constitué un projet complexe, illustrant les défis associés à la transformation d'une solution monolithique locale en une architecture distribuée et scalable. Les étapes initiales (1 à 5) ont été réalisées avec succès, incluant la configuration des conteneurs Docker, la gestion des interactions entre clients et serveurs, la mise en place de systèmes de gestion des bases de données, et l'intégration d'un gestionnaire d'accès. Chaque étape a permis de progresser vers une solution plus robuste et adaptée aux besoins croissants. Cependant, l'étape 6, qui impliquait l'échelle à 1000 développeurs avec Kubernetes, n'a pas été achevée en raison de difficultés techniques.

# Références

Kerrisk, Michael. "fcntl(2) - Linux manual page." *Michael Kerrisk*, 2 5 2024,

<https://man7.org/linux/man-pages/man2/fcntl.2.html>.

"What Is Data Replication?" *IBM*, 13 08 2022, <https://www.ibm.com/topics/data-replication>.