



UNIVERSITE PARIS CITE  
UFR MATHEMATIQUES ET INFORMATIQUE

---

## **Projet Programmation distribuée**

---

Master 1 Vision Machine Intelligente

Abed Nada Fatima-Zahra

Rebai Mohamed Younes

Benmehrez Dima Sabrine

# Table des matières

<b>INTRODUCTION</b> .....	3
<b>1. Développement Initial et Déploiement d'un Service Unique</b> .....	3
1.1 Coder une mini application avec Django & Créer une image Docker .....	4
a) Familiarisation avec l'interface et les taches de notre application .....	5
i. Ajouter une tâche .....	5
ii. Modifier l'état d'une tâche .....	6
iii. Suppression d'une tâche .....	7
1.2 Publier l'image Docker sur le Docker Hub .....	8
1.3 Créer un déploiement Kubernetes .....	8
1.4 Créer un service Kubernetes .....	9
<b>2. Ajouter une gateway en local</b> .....	10
<b>3. Ajouter un deuxième service en local</b> .....	11
3.1 Ajouter un deuxième service en local .....	11
3.2 Relier les services entre-eux .....	13
<b>4. Ajouter une base de données</b> .....	15
<b>CONCLUSION</b> .....	15
<b>Partie LABS</b> .....	16

# INTRODUCTION

Dans le cadre de ce projet, nous allons explorer la création et le déploiement d'un service web en utilisant des technologies et patterns modernes. Ce projet met l'accent sur l'intégration de conteneurs Docker, la gestion multi-conteneurs avec Kubernetes, l'utilisation d'une base de données PostgreSQL hébergée sur Heroku, et l'implémentation de micro-services.

Nous commencerons par développer une application Django, que nous conteneuriserons avec Docker. L'image Docker sera publiée sur Docker Hub et déployée sur un cluster Kubernetes, facilitant la gestion des déploiements et des ressources. Pour la gestion des données, nous utiliserons PostgreSQL, connecté à notre application et hébergé sur Heroku.

Nous mettrons également en œuvre une architecture de micro-services pour une meilleure maintenabilité et évolutivité. Ce projet nous permettra de maîtriser les concepts clés des conteneurs, de l'orchestration avec Kubernetes, de la gestion des bases de données et de l'implémentation de micro-services, tout en développant une application robuste et évolutive.

## 1. Développement Initial et Déploiement d'un Service Unique

Tout d'abord, pour notre première application, nous avons développé une application Django qui fonctionne comme une to-do list. Cette application est intégrée avec une base de données PostgreSQL hébergée sur Heroku. Grâce à cette configuration, nous pouvons ajouter des tâches, modifier leur état et les supprimer, tout en mettant à jour la base de données en temps réel. De plus, nous avons utilisé Retool comme interface de visualisation de notre base de données, ce qui nous permet de surveiller et de gérer facilement les données de notre application.

## 1.1 Coder une mini application avec Django & Créer une image Docker

La commande ``docker build -t front-end-app .`` est une étape cruciale dans le processus de conteneurisation d'une application avec Docker. Cette commande indique à Docker de construire une image à partir des instructions spécifiées dans un fichier Dockerfile. Le fichier Dockerfile doit être situé dans le répertoire courant, représenté par le point (``.``). En utilisant l'option ``-t`` (ou ``--tag``), nous assignons un nom à l'image, ici ``front-end-app``, ce qui permet de facilement identifier et gérer cette image, surtout lorsqu'on travaille avec plusieurs versions ou différentes images. Le contexte de construction (``.``) indique que Docker doit utiliser tous les fichiers du répertoire courant pour construire l'image. En somme, cette commande compile l'application en une image Docker prête à être exécutée dans un conteneur.

```
PS E:\Projet Prog D\progd_projet> docker build -t front-end-app .
[+] Building 24.4s (11/11) FIMISHED
=> [internal] load build definition from Dockerfile
=> transferring dockerfile: 503B
=> [internal] load metadata for docker.io/library/python:3.9-slim
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> transferring context: 34B
=> [internal] load build context
=> transferring context: 12.15kB
=> [1/5] FROM docker.io/library/python:3.9-slim@sha256:088d9217202188598aac37f8db0929345e124a82134ac66b8bb50ee9750b045b
=> resolve docker.io/library/python:3.9-slim@sha256:088d9217202188598aac37f8db0929345e124a82134ac66b8bb50ee9750b045b
=> sha256:088d9217202188598aac37f8db0929345e124a82134ac66b8bb50ee9750b045b 1.86kB / 1.86kB
=> sha256:b9266f45b5809cafacc3853e946f8d249a9080d802cbbdb0becf7f490ee90299 1.37kB / 1.37kB
=> sha256:4602238ffdcf66f436edfb46e21c9521ab4a996051b1a051004fa5a70f3f42 6.90kB / 6.90kB
=> sha256:09f376ebb190216b0459f470e71bec7b5dfa11d66bf008492b40dc5f1d8eae 29.15MB / 29.15MB
=> sha256:276709cbcd1f168290ee408fca2af2aacfeb4f922ddca125e9e8047f9841479 3.51MB / 3.51MB
=> sha256:4e7363ac3b6fb61a9310bb00e385beaa54c712a9633c01de34cc7d8b0823dba 11.89MB / 11.89MB
=> sha256:1f1e6fb6a4a52a77049d55697db79164d7d0e5a78ae115c657699f4471398fc0 244B / 244B
=> sha256:bf8f57a642c477da4e61c92dc0c0fd036a8d7e3d3951df39b88c3dd73bf3d5af 3.13MB / 3.13MB
=> extracting sha256:09f376ebb190216b0459f470e71bec7b5dfa11d66bf008492b40dc5f1d8eae
=> extracting sha256:276709cbcd1f168290ee408fca2af2aacfeb4f922ddca125e9e8047f9841479
=> extracting sha256:4e7363ac3b6fb61a9310bb00e385beaa54c712a9633c01de34cc7d8b0823dba
=> extracting sha256:1f1e6fb6a4a52a77049d55697db79164d7d0e5a78ae115c657699f4471398fc0
=> extracting sha256:bf8f57a642c477da4e61c92dc0c0fd036a8d7e3d3951df39b88c3dd73bf3d5af
[2/5] WORKDIR /app
=> [3/5] COPY requirements.txt .
=> [4/5] RUN pip install --no-cache-dir -r requirements.txt
=> [5/5] COPY . /app
=> exporting to image
=> exporting layers
=> writing image sha256:f3a6d7949bf5a1f532411718566aebb6d3712ca32a4283be32f9263607a3526f
=> naming to docker.io/library/front-end-app
View build details: docker-desktop://dashboard/build/default/default/71k5h9iohcnhkc3gcip41dc
```

Figure 1.1 Résultat de la commande "docker build -t front-end-app ."

L'image "fornt-end-app:latest" dans l'interface de Docker après création.

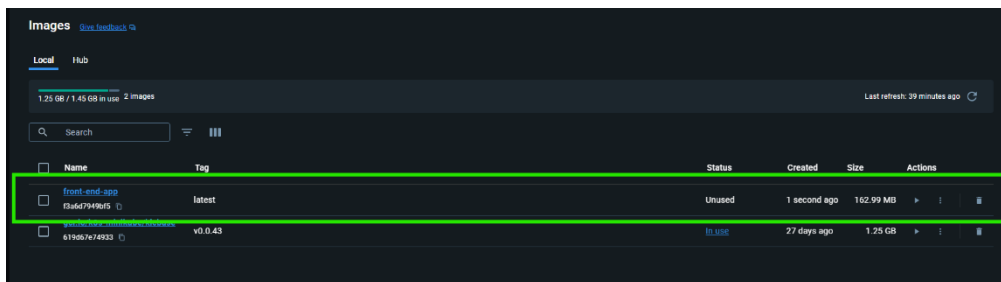


Figure 1.2 Fornt-end-app:latest

Résultat de l'exécution de la commande "docker run -p 8000:8000 front-end-app:latest", le conteneur a bien été créé.

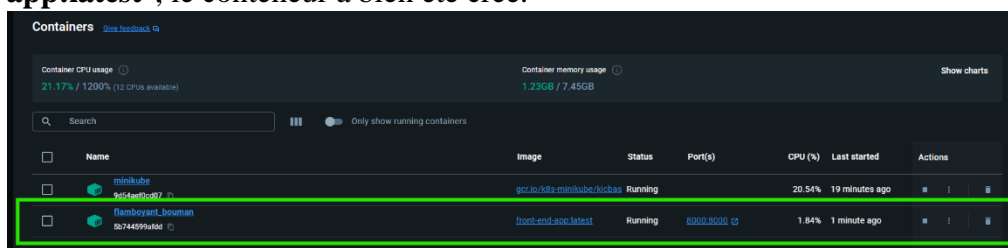
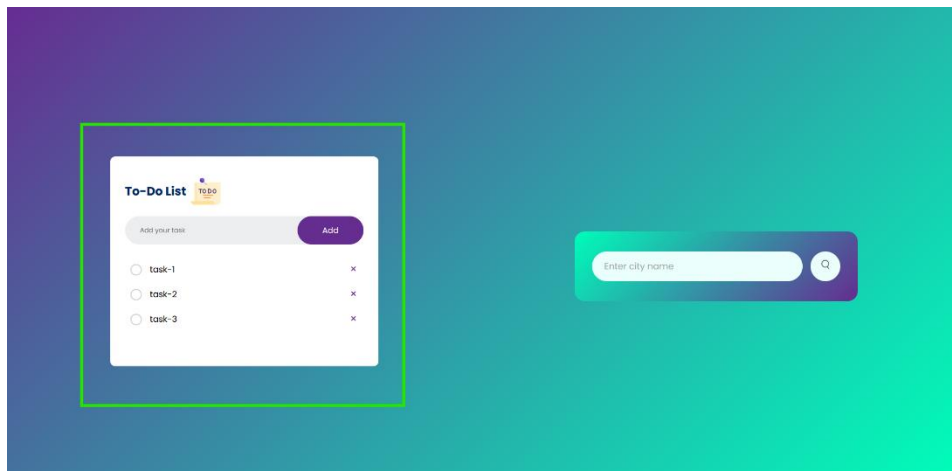


Figure 1.3 Résultat de l'exécution de la commande "docker run -p 8000:8000 front-end-app:latest"

**a) Familiarisation avec l'interface et les tâches de notre application**

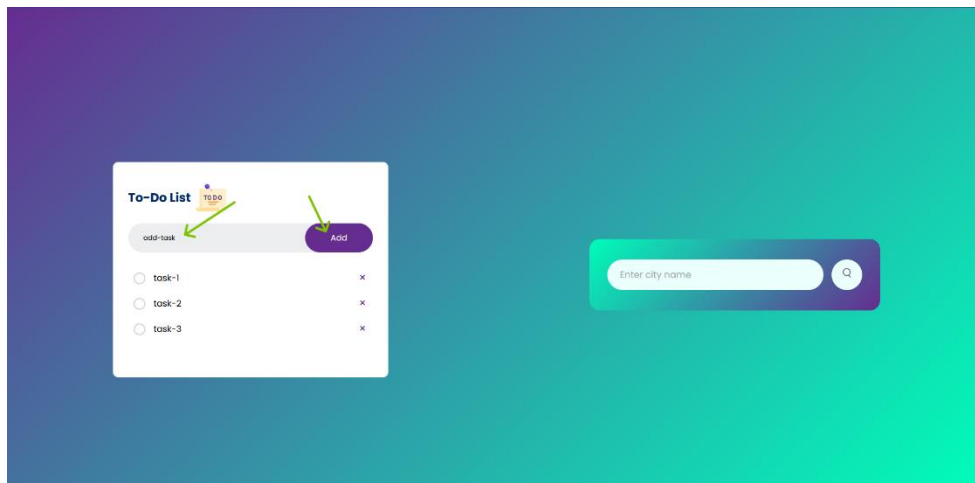
En accédant à "<http://localhost:8000/>", nous arrivons sur une page web affichant notre application. Cette interface nous permet d'interagir avec notre première application, développée et conteneurisée dans une image Docker. En ouvrant cette URL, nous pouvons vérifier que notre application fonctionne correctement et que l'image Docker que nous avons créé précédemment a été construite et déployée avec succès. Cela constitue une étape essentielle pour s'assurer que toutes les fonctionnalités de l'application, telles que la gestion des tâches, la connexion à la base de données PostgreSQL, et l'affichage via l'interface utilisateur, sont opérationnelles et qu'il n'y a pas de problèmes de configuration ou de bugs dans l'environnement de développement.



*Figure 1.4 Affichage de l'application*

**i. Ajouter une tâche**

Pour ajouter une tâche, cliquez sur "**add-task**", remplissez le champ de texte, puis cliquez sur "**ADD**".



*Figure 1.5 Ajout d'une tâche*

La tâche "**add-task**" a bien été affichée et ajoutée à la base de données.

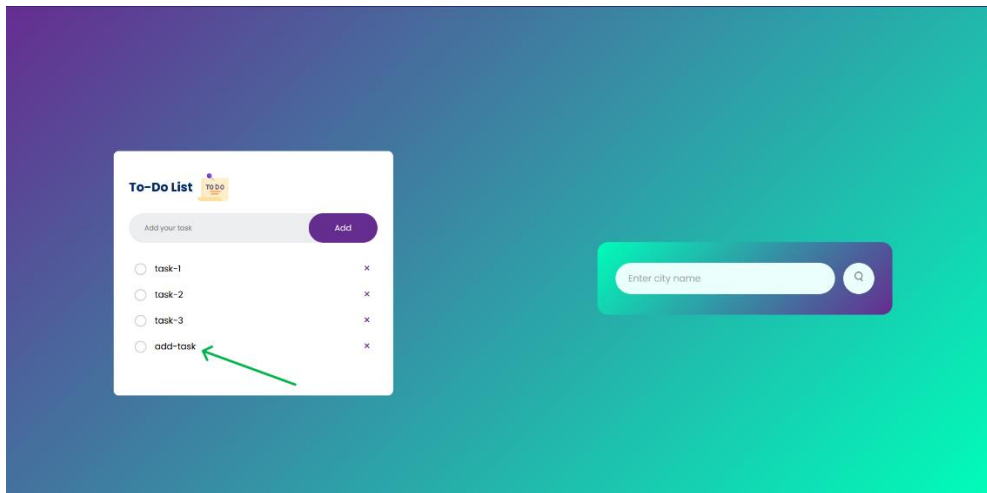


Figure 1.6 La tâche a bien été affichée

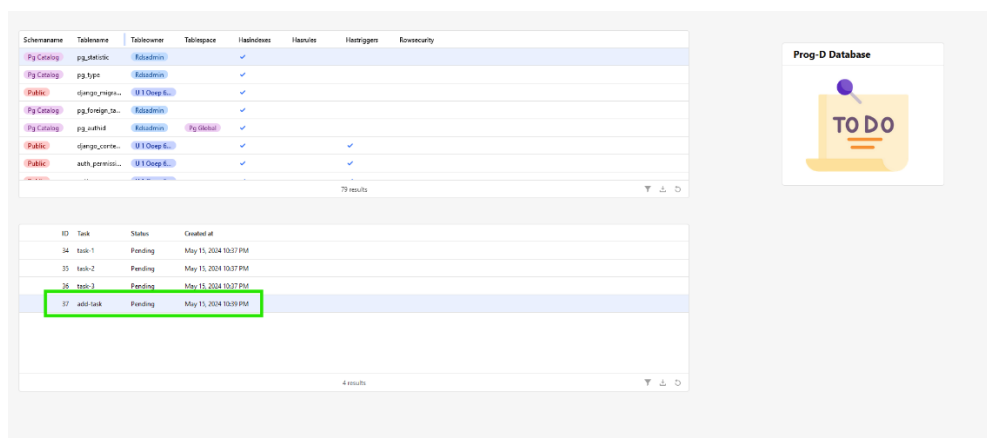


Figure 1.7 La tâche a bien été ajoutée à la base de données

## ii. Modifier l'état d'une tâche

Pour modifier l'état d'une tâche de "Pending" à "Done", il suffit de cocher la case située à gauche de la tâche dans l'interface utilisateur de l'application. Cette action envoie une requête au serveur, qui met à jour l'état de la tâche dans la base de données PostgreSQL hébergée sur Heroku. L'interface utilisateur se met alors à jour pour refléter ce changement, indiquant visuellement que la tâche est terminée. Cela permet aux utilisateurs de suivre facilement l'avancement de leurs tâches et de gérer leur liste de manière efficace.

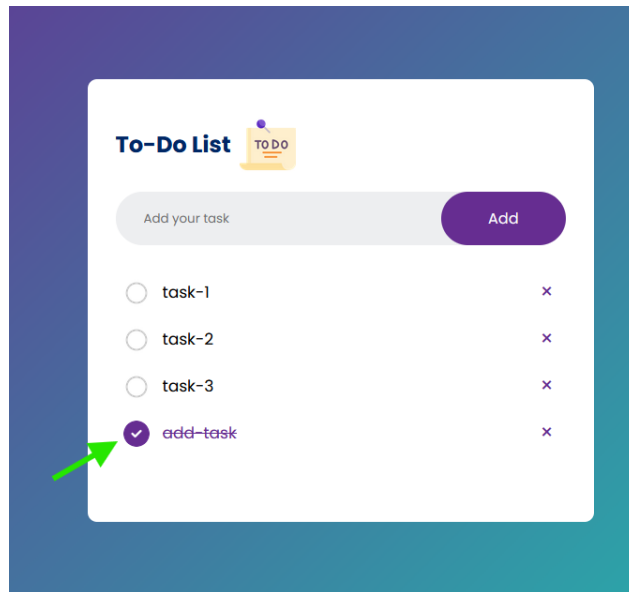


Figure 1.8 Modification de l'état d'une tâche de "Pending" a "Done"

ID	Task	Status	Created at
34	task-1	Pending	May 15, 2024 10:37 PM
35	task-2	Pending	May 15, 2024 10:37 PM
36	task-3	Pending	May 15, 2024 10:37 PM
37	add-task	Done	May 15, 2024 10:39 PM

4 results

Figure 1.9 Etat de la tâche bien modifié

### iii. Suppression d'une tâche

Suppression d'une tâche (en cliquant sur le "x" à droite de la tâche)

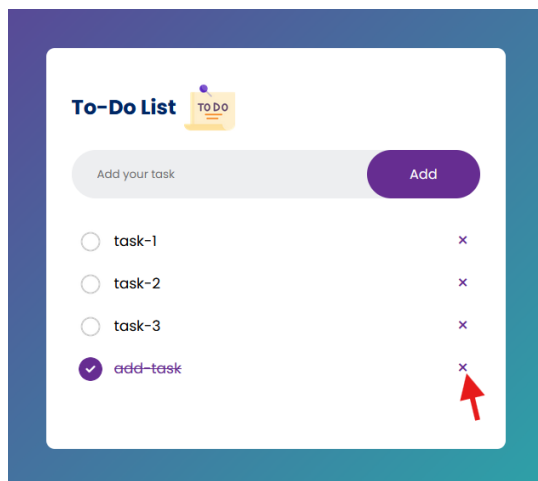


Figure 1.10 Suppression d'une tâche

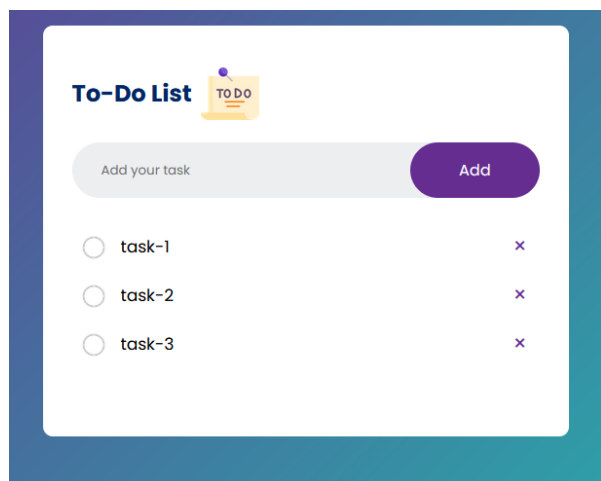


Figure 1.11 Tâche supprimée de l'interface

ID	Task	Status	Created at
34	task-1	Pending	May 15, 2024 10:37 PM
35	task-2	Pending	May 15, 2024 10:37 PM
36	task-3	Pending	May 15, 2024 10:37 PM

Figure 1.12 Tâche supprimée de la base de données

## 1.2 Publier l'image Docker sur le Docker Hub

Pour publier une image Docker sur Docker Hub, nous utilisons deux commandes essentielles : `docker tag` et `docker push`. La commande `docker tag front-end-app:latest younesrebai01/front-end-app:latest` crée un alias pour l'image Docker locale en la renommant avec un nouveau nom, correspondant à notre compte Docker Hub. Cela permet de préparer l'image pour la publication en alignant son nom avec le format requis par Docker Hub. Ensuite, la commande `docker push younesrebai01/front-end-app:latest` envoie l'image marquée vers Docker Hub. Cette opération transfère les couches de l'image au registre Docker Hub, rendant l'image accessible pour téléchargement et utilisation publique. Ainsi, nous pouvons facilement partager et déployer l'image sur d'autres machines ou services, facilitant la distribution et le déploiement de notre application.

```
PS E:\Projet Prog D\progd_projet> docker tag front-end-app:latest younesrebai01/front-end-app:latest
PS E:\Projet Prog D\progd_projet> docker push younesrebai01/front-end-app:latest
>>
The push refers to repository [docker.io/younesrebai01/front-end-app]
540fea426d6d: Pushed
abbfe031c8bb: Pushed
d60e125b68ef: Pushed
bd4c762dd5c0: Pushed
ae96698df02c: Mounted from library/python
bai01/front-end-app:latest
>>
deployment.apps/front-end-service created
```

Figure 1.13 Exécution des deux commandes "docker tag front-end-app:latestyounesrebai01/front-end-app:latest" et "docker push younesrebai01/front-end-app:latest"

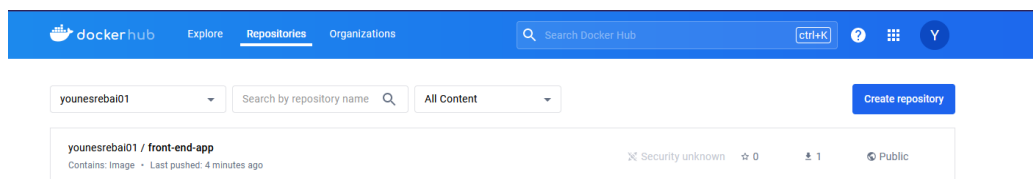


Figure 1.14 L'image a bien été ajoutée

## 1.3 Créer un déploiement Kubernetes

L'exécution de la commande `kubectl create deployment front-end-service --image=younesrebai01/front-end-app:latest` joue un rôle crucial dans le déploiement d'applications sur un cluster Kubernetes. Cette commande indique à Kubernetes de créer un nouveau déploiement nommé `front-end-service` en utilisant l'image Docker spécifiée `younesrebai01/front-end-app:latest`. Le déploiement est une abstraction de Kubernetes qui gère un ensemble de pods, assurant leur disponibilité et leur mise à jour de manière contrôlée. En utilisant cette commande, Kubernetes télécharge l'image Docker depuis Docker Hub, puis crée et lance les pods nécessaires pour exécuter



l'application. Cela permet de gérer automatiquement le nombre de réplicas, de redémarrer les pods en cas de défaillance, et de déployer des mises à jour de l'application sans interruption de service. En somme, cette commande simplifie et automatise le processus de déploiement, garantissant que l'application est déployée de manière fiable et scalable au sein du cluster Kubernetes.

```
PS E:\Projet Prog D\progD_projet> kubectl create deployment front-end-service --image=younesrebai01/front-end-app:latest
>>
deployment.apps/front-end-service created
```

Figure 1.15 Créer un déploiement nommé "front-end-service" à partir de l'image Docker "younesrebai01/front-end-app:latest"

## 1.4 Créer un service Kubernetes

L'exécution des deux commandes suivantes permet de rendre accessible un service déployé sur un cluster Kubernetes via Minikube. La première commande, ``kubectl expose deployment front-end-service --type=NodePort --port=8000``, expose le déploiement nommé ``front-end-service`` en tant que service NodePort. En spécifiant ``--type=NodePort`` et ``--port=8000``, Kubernetes configure le service pour qu'il soit accessible sur le port 8000 sur chaque nœud du cluster. Le type NodePort attribue un port unique à chaque nœud, permettant ainsi l'accès au service de l'extérieur du cluster via l'adresse IP du nœud et le port attribué.

La seconde commande, ``minikube service front-end-service --url``, est spécifique à Minikube et est utilisée pour obtenir l'URL à laquelle le service ``front-end-service`` est accessible. Minikube expose le service localement, facilitant ainsi le développement et les tests en fournissant une URL directe. Par exemple, après l'exécution de cette commande, l'application peut être accessible via une URL comme ``http://127.0.0.1:64190``.

```
PS E:\Projet Prog D\progD_projet> kubectl expose deployment front-end-service --type=NodePort --port=8000
service/front-end-service exposed
PS E:\Projet Prog D\progD_projet> minikube service front-end-service --url
>>
W0516 01:15:15.340917 4788 main.go:291] Unable to resolve the current Docker CLI context
"default": context "default": context not found: open C:\Users\Timgad informatique\.docker\contexts\meta\37a8eec1ce19687d132fe29051dca629d164e2c4958ba141d5f4133a33f0688f\meta.json: The
system cannot find the path specified.
http://127.0.0.1:64190
! Because you are using a Docker driver on windows, the terminal needs to be open to run i
t.
```

Figure 1.16 Déploiement "front-end-service" sur un port spécifié dans tous les nœuds du cluster Kubernetes en tant que service NodePort

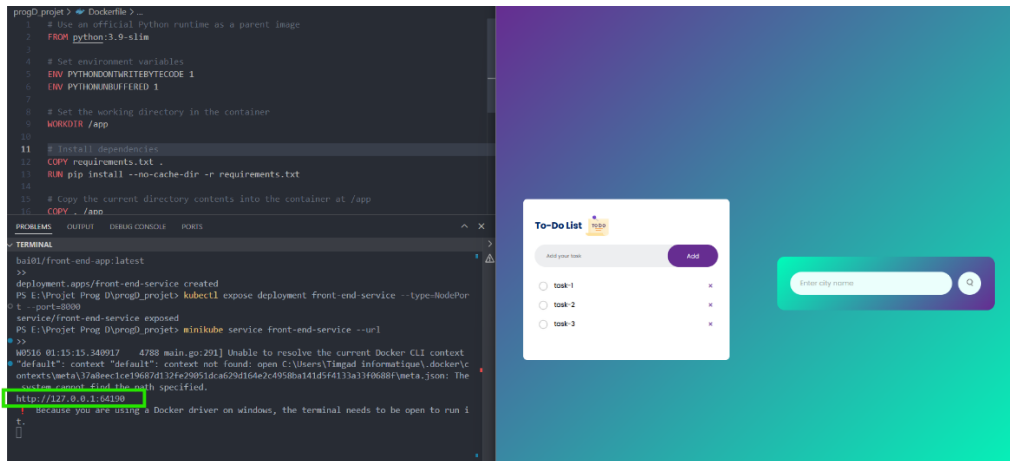


Figure 1.17 Accès à l'application via : <http://127.0.0.1:64190>

## 2. Ajouter une gateway en local

L'exécution des commandes "**kubectl apply -f ingress.yml**" et "**kubectl get ingress**" constitue une étape cruciale dans la configuration du routage du trafic sur un cluster Kubernetes via un Ingress Controller.

La première commande applique la configuration définie dans un fichier YAML nommé "**ingress.yml**" sur le cluster, spécifiant les règles de routage du trafic entrant vers les services appropriés. Cela met à jour la configuration de l'Ingress Controller, permettant un routage correct du trafic vers les services déployés.

La seconde commande affiche tous les objets Ingress actuellement déployés sur le cluster. Cela permet de vérifier que l'Ingress pour notre service, défini dans le fichier "**ingress.yml**", a été correctement créé et qu'il est opérationnel, assurant ainsi que l'application est accessible depuis l'extérieur du cluster.

Ces commandes permettent de configurer et de valider le routage du trafic vers les services déployés sur un cluster Kubernetes, garantissant que l'application est correctement exposée et accessible aux utilisateurs finaux.

```
PS E:\Projet Prog D\progD_projet> kubectl apply -f ingress.yml
Warning: annotation "kubernetes.io/ingress.class" is deprecated, please use 'spec.ingressClassName' instead
ingress.networking.k8s.io/front-ingress created
● PS E:\Projet Prog D\progD_projet> kubectl get ingress
NAME          CLASS    HOSTS          ADDRESS          PORTS    AGE
front-ingress <none>   frontendapp.info 80              18s
```

Figure 2.1 Afficher tous les objets Ingress déployés dans notre cluster Kubernetes

```
● PS E:\Projet Prog D\progD_projet> minikube addons enable ingress-dns
W0516 01:38:05.213675 13456 main.go:291] Unable to resolve the current Docker CLI context "default": context "default":
7a8eccc19687d132fe29051dca629d164e2c4958ba141d5f4133a33f0688f\meta.json: The system cannot find the path specified.
💡 ingress-dns is an addon maintained by minikube. For any concerns contact minikube on GitHub.
You can view the list of minikube maintainers at: https://github.com/kubernetes/minikube/blob/master/OWNERS
💡 After the addon is enabled, please run "minikube tunnel" and your ingress resources would be available at "127.0.0.1"
   Using image gcr.io/k8s-minikube/minikube-ingress-dns:0.0.2
🌟 The 'ingress-dns' addon is enabled
```

Figure 2.2 Exécution de la commande "minikube addons enable ingress-dns"

La commande "**minikube tunnel**" est utilisée pour créer un tunnel réseau entre notre machine locale et le cluster Kubernetes hébergé par Minikube. Ce tunnel permet d'exposer les services

du cluster Kubernetes localement sur notre machine, donc quand on accède à "http://frontendapp.info/" notre application s'affiche comme sur la figure 20 ci-dessous.

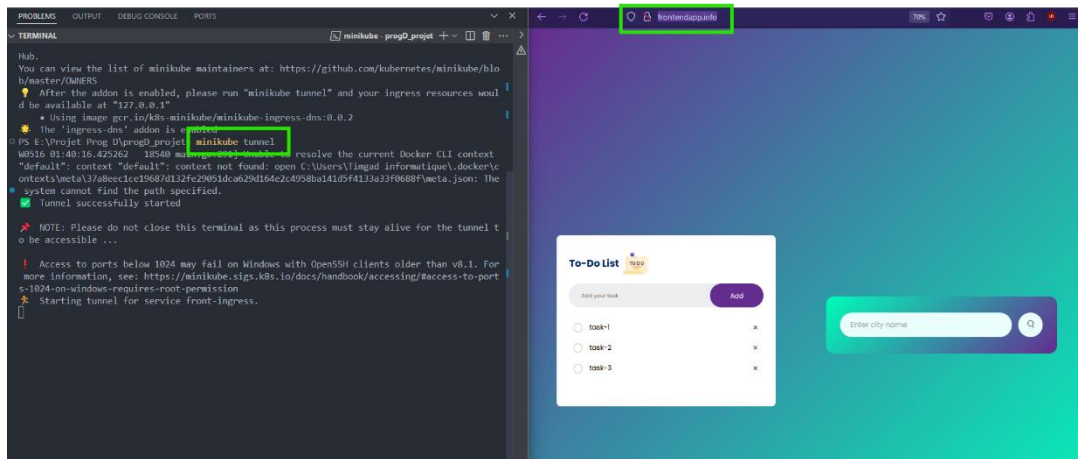


Figure 2.3 Créer un tunnel réseau entre notre machine locale et le cluster Kubernetes hébergé par Minikube

### 3. Ajouter un deuxième service en local

Notre deuxième service est un service météo qui permet d'obtenir les prévisions météorologiques de n'importe quelle ville saisie dans le champ de recherche.

Nous utilisons une API pour récupérer les données. Pour cela, nous avons développé une deuxième application Django qui ne dispose pas d'interface utilisateur. Cette application reçoit des requêtes de la première application, avec le nom de la ville recherchée en tant que paramètre, et répond avec un fichier JSON contenant les données météorologiques. Ces données sont ensuite affichées sur l'interface de notre application principale.

#### 3.1 Ajouter un deuxième service en local

Nous avons donc répété les étapes précédentes pour notre deuxième service :

- Créer une image Docker et publier l'image Docker sur Docker Hub.
- Créer un déploiement Kubernetes.
- Créer un service Kubernetes.
- Ajouter une gateway en local.

### a. Créer une image Docker et publier l'image Docker sur le Docker Hub.

```
PS E:\Projet Prog D\weather_service\weather_service>
>> docker build -t weather-service .
>> docker tag weather-service younesrebai01/weather-service:latest
>> docker push younesrebai01/weather-service:latest
[+] Building 12.8s (11/11) FINISHED                                docker:default
=> [internal] load build definition from dockerfile                 0.0s
=> => transferring dockerfile: 503B                                0.0s
=> [internal] load metadata for docker.io/library/python:3.9-slim  1.0s
=> [auth] library/python:pull token for registry-1.docker.io       0.0s
=> [internal] load .dockerignore                                   0.0s
=> => transferring context: 2B                                       0.0s
=> [1/5] FROM docker.io/library/python:3.9-slim@sha256:088d9217202188598aac37f8db092 0.0s
=> [internal] load build context                                   0.0s
=> => transferring context: 1.39kB                                   0.0s
=> CACHED [2/5] WORKDIR /app                                       0.0s
=> [3/5] COPY requirements.txt .                                    0.1s
=> [4/5] RUN pip install --no-cache-dir -r requirements.txt        10.6s
=> [5/5] COPY . /app                                              0.1s
=> => exporting to image                                           0.8s
=> => exporting layers                                             0.8s
=> => writing image sha256:4ef224db6804e69ch28436a9d206520be34a85e78f1e35d5dc51ac9bb 0.0s
=> => naming to docker.io/library/weather-service                 0.0s

View build details: docker-desktop://dashboard/build/default/default/q44kduf5e80dbabxh3ro14io

What's Next?
View a summary of image vulnerabilities and recommendations + docker scout quickview
The push refers to repository [docker.io/younesrebai01/weather-service]
f01013ada16b: Pushed
e21bb63e5081: Pushed
afd02cd1be9d: Pushed
bd4c762dd5c0: Mounted from younesrebai01/front-end-app
ae96698df02c: Mounted from younesrebai01/front-end-app
e555c0055a9b: Mounted from younesrebai01/front-end-app
205262265e50: Mounted from younesrebai01/front-end-app
146826fa3ca0: Mounted from younesrebai01/front-end-app
5d442706decc: Mounted from younesrebai01/front-end-app
latest: digest: sha256:cf3c9085def186c0f5ff362baa3231acd4c44381de30fd71eff12ee3ea3f8ed7 size: 2203
PS E:\Projet Prog D\weather_service\weather_service>
```

Figure 3.1 Créer une image Docker, publier l'image Docker sur le Docker Hub

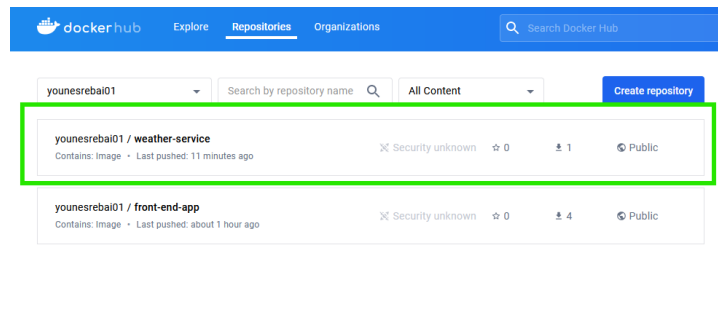


Figure 3.2 L'image a bien été ajoutée

### b. Créer un déploiement Kubernetes

Créer un déploiement nommé "**weather-service**" à partir de l'image Docker "**younesrebai01/weather-service:latest**"

```
PS E:\Projet Prog D\weather_service\weather_service>
>> kubectl create deployment weather-service --image=younesrebai01/weather-service:latest
>> kubectl expose deployment weather-service --type=NodePort --port=8000
deployment.apps/weather-service created
service/weather-service exposed
```

Figure 3.3 Créer un déploiement nommé "weather-service"

### c. Créer un service Kubernetes

Exécution des deux commandes "**kubectl apply -f ingress.yml**" Cette commande est utilisée pour appliquer la configuration définie dans un fichier YAML appelé "**ingress.yml**" sur le cluster Kubernetes.

La commande "**kubectl get ingress**" Cette commande est utilisée pour afficher tous les objets Ingress actuellement déployés dans notre cluster Kubernetes on peut constater que notre "**weather-ingress**" est bien dedans.

```
PS E:\Projet Prog D\weather_service\weather_service>
>> kubectl apply -f ingress.yaml
>> kubectl get ingress
Warning: annotation "kubernetes.io/ingress.class" is deprecated, please use 'spec.ingressClassName' instead
ingress.networking.k8s.io/weather-ingress created
NAME          CLASS    HOSTS          ADDRESS          PORTS    AGE
front-ingress <none>    frontendapp.info 192.168.49.2     80      58m
weather-ingress <none>    weather.example.com 80      0s
```

Figure 3.4 Afficher tous les objets Ingress actuellement déployés dans notre cluster Kubernetes

#### d. Ajouter une gateway en local

Créer deux tunnels qui nous permettent d'exposer les deux services du cluster Kubernetes localement sur notre machine, on remarque que les deux tunnels sont en marche. L'accès à `'http://weather.example.com/weather/?city=Paris'` permet changer les paramètres city pour n'importe quelle ville.

La sortie est une réponse Json ce qui signifie que le service 2 fonctionne parfaitement.

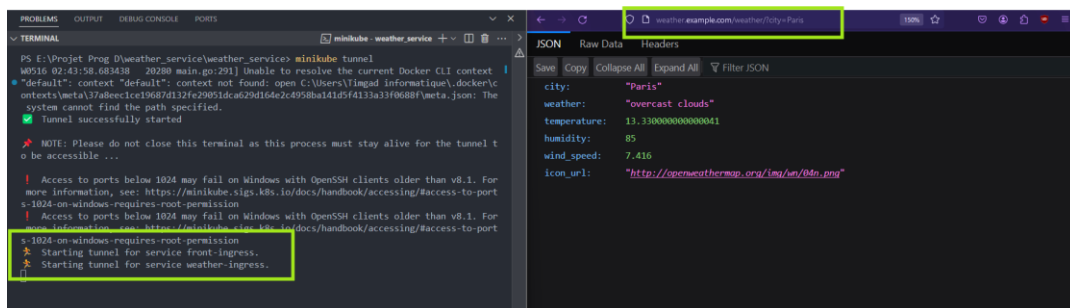


Figure 3.5 Les deux tunnels sont en marche

### 3.2 Relier les services entre-eux

Le premier service envoie une requête contenant le nom de la ville saisie en tant que paramètre, tandis que le deuxième service reçoit cette requête et traite les données pour générer une réponse au format JSON. Cette réponse est ensuite utilisée par l'application pour afficher les informations météorologiques demandées par l'utilisateur.

```

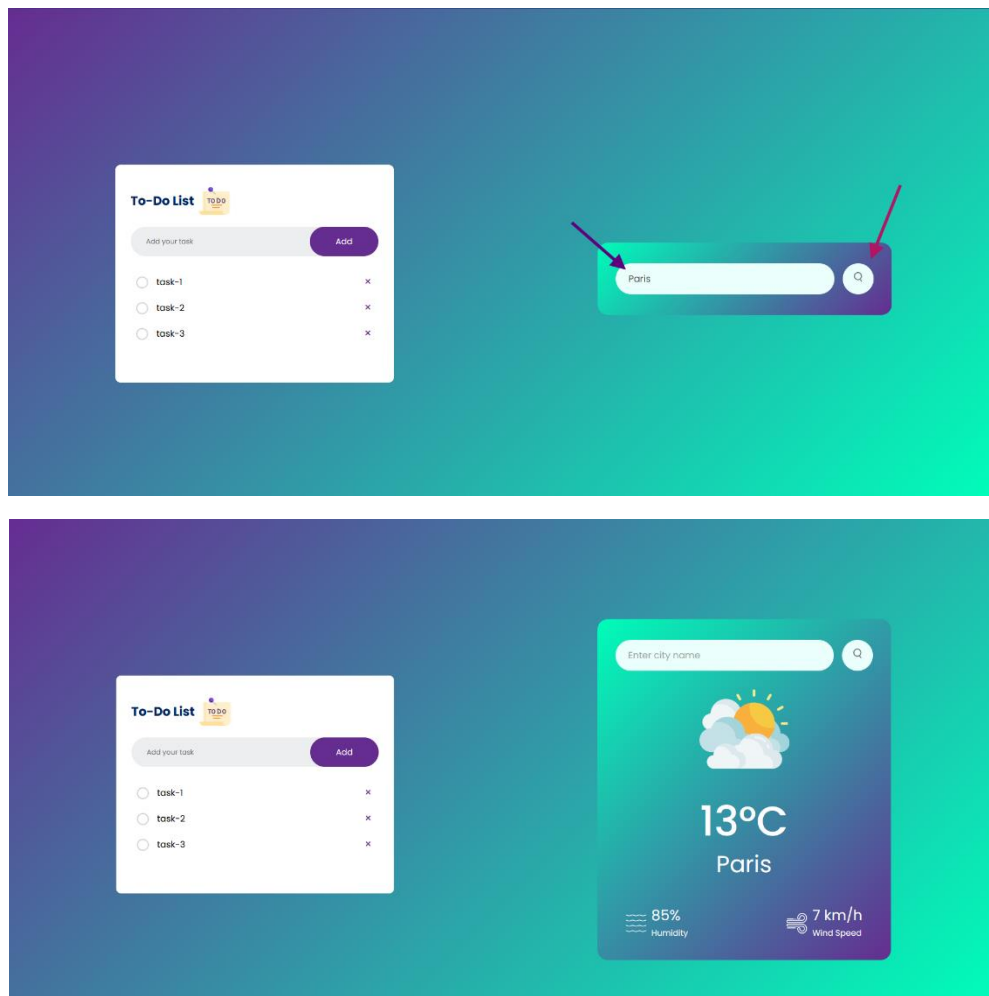
function searchWeather() {
  // Retrieve the value entered by the user
  let city = document.getElementById('city-input').value;

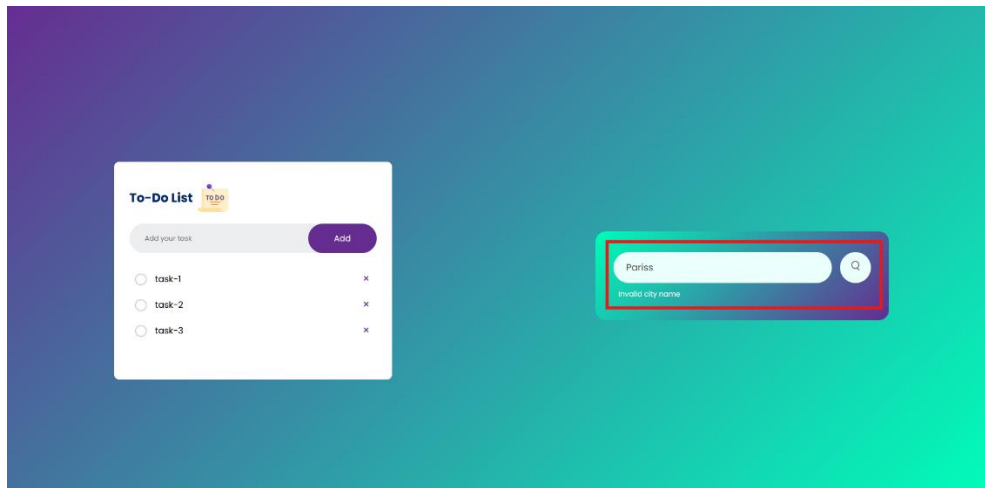
  // Fetch weather data for the entered city
  fetch('http://weather.example.com/weather/?city=${city}')
    .then(response => response.json())
    .then(data => {
      document.querySelector('.weather').style.display = "block";
      // Update the UI with the received weather data
      document.getElementById('city-name').innerText = capitalizeFirstLetter(data.city);
      // Round temperature to the nearest integer
      document.getElementById('temperature').innerText = `${Math.round(data.temperature)}°C`;
      // Round humidity to the nearest integer
      document.getElementById('humidity').innerText = `${Math.round(data.humidity)}%`;
      // Round wind speed to the nearest integer
      document.getElementById('wind-speed').innerText = `${Math.round(data.wind_speed)} km/h`;
      // Update weather icon
      const weatherIcon = getWeatherIcon(data.weather);
      document.querySelector('.weather-icon').src = `static /icons/ ${weatherIcon}`;
      // Clear the input field after search
      document.getElementById('city-input').value = '';
      document.querySelector('.error').style.display = "none";
    })
    .catch(error => {
      document.querySelector('.error').style.display = "block";
      document.querySelector('.weather').style.display = "none";
      console.error('Error fetching weather data:', error);
    });
}

```

Figure 3.6 Relier les services entre-eux

Familiarisation avec l'interface maintenant que le service 2 qui fonctionne, on revient sur <http://frontendapp.info/>.





## 4. Ajouter une base de données

Nous utilisons une base de données PostgreSQL hébergée sur Heroku pour gérer et stocker les données de notre application. Heroku offre un service de base de données fiable et évolutif, ce qui nous permet de bénéficier d'une infrastructure robuste sans avoir à gérer les détails de l'administration de la base de données. Cette configuration nous permet de nous concentrer sur le développement de notre application tout en assurant que les données sont gérées de manière efficace et sécurisée.

En complément, nous utilisons Retool comme interface de visualisation de notre base de données. Retool est un outil puissant qui permet de créer rapidement des interfaces utilisateur personnalisées pour interagir avec les données stockées dans PostgreSQL. Grâce à Retool, nous pouvons visualiser, gérer et analyser nos données facilement, ce qui améliore notre capacité à superviser et à administrer notre base de données de manière intuitive et efficace.

## CONCLUSION

En conclusion, ce projet nous a permis de plonger dans le monde fascinant du développement et du déploiement d'applications modernes en utilisant des technologies avancées telles que les conteneurs Docker, Kubernetes et PostgreSQL hébergé sur Heroku. En parcourant les différentes étapes, de la création d'une application initiale à son déploiement sur un cluster Kubernetes en passant par l'intégration de bases de données et l'implémentation de micro-services, nous avons acquis une compréhension approfondie des principes fondamentaux et des meilleures pratiques de développement d'applications.

Au-delà des aspects techniques, ce projet nous a également permis d'appréhender des concepts cruciaux dans des environnements distribués. En adoptant une approche itérative et collaborative, et en résolvant des défis pratiques tout au long du processus, nous avons renforcé nos compétences en résolution de problèmes.



Partie LABS

Nada Fatima Zohra ABED

Date d'abonnement : 2024

280 points

Votre profil n'est pas public ni accessible. Rendre le profil public

Parcours de formation

Activités

Classement

Badges

Cours

Atelier

Quiz

Jeu

En cours

Terminée

Activité	Type	Date de début	Date de fin	Score	Réussie
Infrastructure as Code avec Terraform	Atelier	il y a 8 jours	il y a 8 jours	Assessment: 100%	✓
Développement d'applications : déployer l'application dans Kubernetes Engine – Python	Atelier	il y a 8 jours	il y a 8 jours	Assessment: 70%	
Présentation des ateliers pratiques Google Cloud	Atelier	il y a 8 jours	il y a 8 jours	Assessment: 100%	✓

Mohamed Younes REBAI

Member since 2024

280 points

Your profile is not public and accessible. Make profile public

Paths

Activities

Leaderboard

Badges

Course

Lab


Quiz

Game

In progress

Finished

Activity	Type	Date started	Date finished	Score	Passed
Infrastructure as Code with Terraform	Lab	8 days ago	8 days ago	Assessment: 100%	✓
App Dev: Deploying the Application into Kubernetes Engine - Python	Lab	8 days ago	8 days ago	Assessment: 70%	
A Tour of Google Cloud Hands-on Labs	Lab	8 days ago	8 days ago	Assessment: 100%	✓



dima Benmehrez

Date d'abonnement : 2024

250 points

Votre profil n'est pas public ni accessible. Rendre le profil public

Parcours de formation

Activités

Classement

Badges

Cours

Atelier

Quiz

Jeu

En cours

Terminée

Activité	Type	Date de début	Date de fin	Score	Réussie
Présentation des ateliers pratiques Google Cloud	Atelier	24 avr. 2024	24 avr. 2024	Assessment: 100%	✓
Infrastructure as Code avec Terraform	Atelier	24 avr. 2024	24 avr. 2024	Assessment: 100%	✓