

- [Discord bot](#)
 - [Les étapes de création du bot](#)
 - [1\) Local](#)
 - [2\) Déployer sur une VM](#)
 - [3\) Ajouter le linting](#)
 - [Demo du bot](#)
- [Le bot étape 1](#)
 - [Creez le bot sur le portaitl dev de discord](#)
 - [le DISCORD_TOKEN](#)
 - [L'environnement local](#)
 - [Le bot dans main.py](#)
 - [A vous](#)
 - [Le Dockerfile](#)
 - [Pourquoi un docker compose.yaml alors qu'il n'y a qu'un service ?](#)
 - [Le compose.yaml](#)
 - [docker compose -f compose.yaml config](#)
 - [Lancer le bot](#)
 - [Inviter le bot sur le server discord](#)
 - [Exemples de services complémentaires](#)
 - [Déploiement avec GitHub Actions](#)
 - [Set up secrets in your GitHub repository:](#)
 - [Version 3: Linting et validation](#)
 - [Then deploy to host](#)
 - [Running Locally with Docker](#)

Discord bot

Le plan de l'après midi: **construire un bot Discord**

D'abord en local, puis dans le cloud sur un VM sur digital ocean.

Ce qui nous permettra d'illustrer et de pratiquer les points suivant

- Déploiement with github actions (CI/CD)
- Gestion des *secrets* avec Github secrets
- Lint : comment améliorer la qualité de son code, appliqué aux dockerfiles compose.yaml et

Les étapes de création du bot

1) Local

1. créer une application dans Discord Developer Portal:
<https://discord.com/developers/applications>, ouvrir un compte, remplir des pages web.
2. prendre le script python dans la repo github : `main.py`
3. écrire un `Dockerfile`
4. écrire un `compose.yaml` pour docker compose

Lancer le bot avec `docker compose up -d`

2) Déployer sur une VM

- créer une repo github avec le code python du bot et un `readme.md`
- récupérer le fichier `deploy.yaml` sur le github
 - comprendre `deploy.yaml` : deploy, ssh keys
 - ajouter les secrets dans github
 - récupérer le SSH key Private du server
- push to master / main => et voir le déploiement

Jouer avec le bot!

3) Ajouter le linting

Ajouter des étapes de Linting et de validation des fichiers dockerfile, compose.yaml et python au fichier de déploiement.

Demo du bot

quelques commandes

- `/hello`
- `/ping`
- `/inspire`
- `/roll 2d6`

Le bot étape 1

Sur votre local créez un nouveau repertoire

```
mkdir discord-bot  
cd discord_bot
```

Creez le bot sur le portail dev de discord

Créer votre bot sur le portail Discord

1. Allez sur le Portail Développeur Discord : <https://discord.com/developers/applications>
2. Cliquez sur "New Application" (Nouvelle Application) et donnez-lui un nom
3. Allez dans l'onglet "Bot" et cliquez sur "Add Bot" (Ajouter un Bot)
4. Sous le nom d'utilisateur du bot, cliquez sur "Copy" (Copier) pour copier le DISCORD_TOKEN de votre bot

le DISCORD_TOKEN

Il faut garder le `DISCORD_TOKEN` secret!

Pour cela dans le répertoire `discord-bot`, créez un fichier `.env` et ajoutez la ligne

```
DISCORD_TOKEN=<le token>
```

Gérer le token pour qu'il reste secret et que le fichier `.env` ne soit pas copié dans le container va être un fil rouge du projet.

L'environnement local

Nous n'avons pas besoin d'installer python puisque nous avons docker!

Mais nous avons quand même besoin du script python du bot : `main.py`

- Récupérez le depuis <https://github.com/SkatAI/ynov-docker/blob/master/apps/discord-bot/src/main.py>
- Par convention on met les fichiers python dans un sous répertoire `src`.

Dans un fichier `requirements.txt` ajoutez la ligne

```
discord.py==2.3.2
```

Le contenu de votre répertoire projet `discord-bot` doit ressembler à

```
.
├── .env
├── requirements.txt
└── src
    └── main.py

2 directories, 3 files
```

Le bot dans main.py

Dans le fichier main.py

Une seule commande est définie : `/hello`

ChatGPT est très bon pour proposer d'autres actions.

A vous

Vous allez maintenant écrire les fichiers suivants

- le `Dockerfile` qui permet de construire votre image et de runner le container
- le `compose.yaml` qui permet lancer le bot avec `docker compose up`
- le `.dockerignore` pour éviter que `.env` ne se retrouve dans le container

Le Dockerfile

Un Dockerfile classique

le plus simple est

- de partir de l'image `python:3.12-slim`
- le workdir est `'/app'`
- de copier le `requirements.txt` et d'installer les librairies avec `pip install`
- de copier le code `main.py`
- et de l'exécuter avec `python main.py` de préférence avec une syntaxe **Exec** (définit la commande et ses arguments sous forme d'array)

Pourquoi un docker compose.yaml alors qu'il n'y a qu'un service ?

- Gestion des variables d'environnement
 - Docker Compose fournit un moyen propre d'injecter la variable d'environnement `DISCORD_TOKEN`.
 - Vous pouvez utiliser un fichier `.env` avec Compose automatiquement, alors qu'avec Docker seul, vous devriez les spécifier en ligne de commande ou utiliser `--env-file`.
- Commandes standardisées

- Au lieu de se souvenir des commandes Docker build/run avec tous leurs paramètres, vous pouvez utiliser :

```
docker compose up --build    # Construire et démarrer
docker compose down          # Arrêter et supprimer les conteneurs
```

Bash

- Scalabilité future
 - Si vous décidez plus tard d'ajouter d'autres services (comme une base de données ou un cache), vous n'aurez pas besoin de refactorer.
 - Il suffit d'ajouter de nouveaux services dans le même fichier compose.

Le compose.yaml

Donc écrivez un `compose.yaml` avec

- un seul service appelé `discord-bot`
- construit à partir du Dockerfile (et qui ne part pas d'une image) (`build .`)
- avec les volumes adéquats. Comme il n'y a que 2 fichiers (`src/main/py` et `requirements.txt`), on peut les monter un à un.
- et la variable d'environnement `DISCORD_TOKEN`

La commande suivante permet de vérifier que le fichier est bien écrit

```
docker compose -f compose.yaml config
```

Bash

docker compose -f compose.yaml config

Lorsque vous exécutez `docker compose -f compose.yaml config`, Docker Compose :

- Analyse le fichier et vérifie la syntaxe et la structure.
- Valide les services, volumes, réseaux, et configurations.
- Combine les configurations si plusieurs fichiers Compose sont utilisés.
- Affiche la configuration finale.

En cas d'erreur (syntaxe, champs manquants, configurations non prises en charge), la commande les signalera.

En résumé, cette commande valide le fichier et affiche la configuration finale, en indiquant les éventuelles erreurs.

Lancer le bot

Et vous pouvez bien entendu lancer le bot avec

```
docker compose up
```

Bash

Inviter le bot sur le server discord

Pour que le bot devienne actif, il faut l'autoriser et l'inviter

1. Retournez sur le Portail Développeur Discord
2. Allez dans l'onglet "OAuth2", puis "URL Generator"
3. Sélectionnez "bot" sous "Scopes"
4. Choisissez les permissions que vous souhaitez donner à votre bot
5. Copiez l'URL générée et copiez la dans le channel #bot-playground
6. A ce stade il faut que j'admette le bot sur le server

Exemples de services complémentaires

Le bot est simple mais on peut ajouter deds serviue en les definissant dans le compose.yaml.

Voici quelques exemples potentiels.

```
# For music bots
lavalink:
  image: fredboat/lavalink:dev
  ports:
    - "2333:2333"

# For image processing/AI features
ai-service:
  build: ./ai-service
  environment:
    - OPENAI_API_KEY=${OPENAI_API_KEY}

# For handling background tasks
celery-worker:
  build: .
  command: celery -A tasks worker
  depends_on:
    - redis
    - discord-bot

# Message queue for task distribution
rabbitmq:
  image: rabbitmq:3-management
  ports:
    - "5672:5672"
    - "15672:15672"
```

YAML

Déploiement avec GitHub Actions

On va déployer le bot sur le serveur

Créez le fichier `.github/workflows/deploy.yml` avec le contenu:

YAML

```
name: Deploy Discord Bot

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Login to DockerHub
        uses: docker/login-action@v1
        with:
          username: ${ secrets.DOCKERHUB_USERNAME }
          password: ${ secrets.DOCKERHUB_TOKEN }

      - name: Build and push Docker image
        uses: docker/build-push-action@v2
        with:
          context: .
          push: true
          tags: yourdockerhubusername/discord-bot:latest

      - name: Deploy to server
        uses: appleboy/ssh-action@master
        with:
          host: ${ secrets.SERVER_HOST }
          username: ${ secrets.SERVER_USERNAME }
          key: ${ secrets.SERVER_SSH_KEY }
          script: |
            docker pull yourdockerhubusername/discord-bot:latest
            docker stop discord-bot || true
            docker rm discord-bot || true
            docker run -d --name discord-bot -e DISCORD_TOKEN=${ secrets.DISCORD_TOKEN }
```

Set up secrets in your GitHub repository:

- DOCKERHUB_USERNAME
- DOCKERHUB_TOKEN
- DISCORD_TOKEN
- SERVER_HOST
- SERVER_USERNAME

- SERVERSSHKEY

la clef SERVERSSHKEY est dans le document

https://docs.google.com/document/d/1BUbD0M-3UkMHywpC3a7wgjDt-Bamj7dWZLDrk5o_dYc/edit?tab=t.0

le SERVER_HOST est : 161.35.90.136

SERVER_USERNAME: root

Version 3: Linting et validation

dans deploy.yml

This workflow provides several layers of validation:

1. Docker Compose Validation:
 - Validates the syntax of your compose file
 - Ensures all referenced services and volumes are properly defined
2. Dockerfile Linting with Hadolint:
 - Checks for best practices
 - Identifies potential issues
 - Ensures consistency
3. Build Testing:
 - Verifies that the Dockerfile can build successfully
4. Security Scanning:
 - Uses Trivy to scan for configuration issues
 - Uses Snky to check for vulnerabilities (requires SNYK_TOKEN)
5. Compose Integration Test:
 - Tests if all services can start together
 - Checks for proper container orchestration

To enhance this further, you could:

1. Add custom test scripts to verify service connectivity: `yml - name: Test Service Connectivity run: | docker compose exec discord-bot ping -c 1 redis docker compose exec discord-bot ping -c 1 db`
2. Add specific linting for compose files: ```yaml - name: Install compose-linter run: npm install -g compose-linter`
 - name: Lint compose file run: `compose-linter docker-compose.yml ```

3. Add container structure tests: `yml - name: Container Structure Test uses: plexsystems/container-structure-test-action@v1 with: image: test-image config: container-structure-tests.yml`

Then deploy to host

Running Locally with Docker

To run your bot locally using Docker:

1. Build the Docker image:

```
docker build -t discord-bot .
```

Bash

1. Run the container:

```
docker run -e DISCORD_TOKEN=your_token_here discord-bot
```

Bash

Or use Docker Compose:

```
DISCORD_TOKEN=your_token_here docker-compose up
```

Bash