## TP : Déploiement d'une application Django avec Docker, CI/CD sur GitHub, et déploiement automatique sur Railway



**Objectif**

L'objectif de ce TP est de créer une application Django simple, de la dockeriser, de configurer une pipeline CI/CD avec GitHub Actions et de déployer l'application sur Railway. En plus, nous intégrerons Sentry pour le suivi des erreurs.

### 1. Création d'un projet Django

**1.1 Initialisation du projet**

Nous allons commencer par créer un projet Django. Suivez ces étapes :

1. **Créer un dossier pour votre projet** : Dans votre terminal, créez un nouveau dossier et naviguez à l'intérieur :

mkdir my\_django\_app

cd my\_django\_app

1. **Créer un environnement virtuel et l'activer** : Un environnement virtuel permet d'isoler les dépendances de votre projet. Créez et activez-le avec les commandes suivantes :

python -m venv venv

source venv/bin/activate # macOS/Linux

venv\Scripts\activate # Windows

1. **Installer Django** : Ensuite, installez Django dans cet environnement virtuel :

pip install django

1. **Créer le projet Django** : Utilisez la commande django-admin pour créer un projet Django nommé myproject :

django-admin startproject myproject

cd myproject

1. **Créer une application Django** : Créez une application Django simple à l'intérieur de votre projet pour y ajouter des fonctionnalités :

python manage.py startapp myapp

1. **Ajouter l'application à INSTALLED\_APPS dans settings.py** : Ouvrez le fichier myproject/settings.py et ajoutez votre nouvelle application à la liste des applications installées :

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'myapp', # Ajout de notre application

]

### 2. Création du modèle MyModel

**2.1 Définir un modèle simple**

Dans myapp/models.py, définissez un modèle appelé MyModel avec un champ name pour tester l'ajout d'objets à la base de données :

from django.db import models

class MyModel(models.Model):

name = models.CharField(max\_length=100)

def \_\_str\_\_(self):

return self.name

**2.2 Appliquer les migrations**

Exécutez les migrations pour créer les tables de votre base de données SQLite par défaut (Django utilise SQLite par défaut) :

python manage.py makemigrations

python manage.py migrate

### 3. Création d'une vue et d'un template pour afficher les objets

**3.1 Créer une vue dans views.py**

Dans myapp/views.py, ajoutez une vue qui récupère tous les objets du modèle MyModel et les rend dans un template :

from django.shortcuts import render

from .models import MyModel

def my\_model\_list(request):

MyModel.objects.create(name="Objet de test")

objects = MyModel.objects.all() # Récupérer tous les objets MyModel

return render(request, 'myapp/my\_model\_list.html', {'objects': objects})

**3.2 Créer le template HTML**

Créez un fichier de template dans myapp/templates/myapp/my\_model\_list.html pour afficher les objets MyModel dans une liste :

<!DOCTYPE html>

<html>

<head>

<title>MyModel List</title>

</head>

<body>

<h1>Liste des objets MyModel</h1>

<ul>

{% for obj in objects %}

<li>{{ obj.name }}</li>

{% endfor %}

</ul>

</body>

</html>

**3.3 Ajouter l'URL dans urls.py**

Dans myapp/urls.py, ajoutez une route pour accéder à la vue my\_model\_list :

from django.urls import path

from . import views

urlpatterns = [

path('mymodel/', views.my\_model\_list, name='my\_model\_list'),

]

Dans myproject/urls.py, incluez les URL de myapp :

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('myapp/', include('myapp.urls')), # Ajouter la route de notre application

]

**3.4. Créer un fichier pour les TEST**

Dans myapp/tests.py, vous pouvez ajouter un test simple pour vérifier que votre modèle MyModel fonctionne correctement. Voici un exemple :

from django.test import TestCase

from .models import MyModel

class MyModelTest(TestCase):

def test\_model\_creation(self):

# Créer un objet MyModel

obj = MyModel.objects.create(name="Test Object")

# Vérifier que l'objet a été créé et que son nom est correct

self.assertEqual(obj.name, "Test Object")

self.assertTrue(MyModel.objects.exists())

**TestCase** : TestCase est une classe fournie par Django pour faciliter l'écriture de tests unitaires. Elle inclut des outils pour tester les modèles, les vues, etc.

**test\_model\_creation** : Ce test vérifie que l'objet MyModel peut être créé et que ses données sont correctes.

**assertEqual** et **assertTrue** : Ce sont des assertions utilisées pour vérifier que les résultats sont ceux attendus. Si ces conditions échouent, le test échoue.

### 4. Configuration de Docker et Docker Compose

**4.1 Créer un fichier Dockerfile**

Dans le dossier myproject, créez un fichier Dockerfile pour construire l'image Docker de votre application Django :

# Utiliser l'image officielle de Python

FROM python:3.9-slim

# Définir le répertoire de travail

WORKDIR /app

# Copier le fichier requirements.txt et installer les dépendances

COPY requirements.txt /app/

RUN pip install -r requirements.txt

# Copier tout le code de l'application dans le conteneur

COPY . /app/

# Exposer le port 8000 pour Django

EXPOSE 8000

# Commande par défaut pour démarrer l'application Django

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]

**4.2 Créer un fichier docker-compose.yml**

Créez un fichier docker-compose.yml dans le dossier myproject pour configurer les services de votre application Django avec Docker :

version: '3'

services:

app:

build: .

volumes:

- .:/app

ports:

- "8000:8000"

    command: python manage.py runserver 0.0.0.0:8000  *# Ajout de la commande pour démarrer le serveur*

**4.3 Créer un fichier requirements.txt**

Listez les dépendances nécessaires pour votre projet dans un fichier requirements.txt :

pip freeze > requirements.txt

**4.4 Construire et démarrer les conteneurs Docker Compose**

Exécutez la commande suivante pour construire et démarrer vos conteneurs Docker Compose :

docker compose up --build

puis celle-ci pour checker que les test passent bien (sur un autre CMD) :

docker compose exec app python manage.py test

### 5. Configuration de Github CI

**1. Initialisation du dépôt Git local**

Commencez par initialiser votre dépôt local, ajouter vos fichiers et effectuer un premier commit, comme suit :

**Étapes pour initialiser votre dépôt Git local :**

1. **Initialiser le dépôt Git localement** : Dans le terminal, allez dans le dossier de votre projet et exécutez :

git init

1. **Ajouter tous les fichiers au dépôt Git** : Ajoutez tous les fichiers du projet au dépôt :

git add .

1. **Faire un premier commit** : Ensuite, vous devez faire un premier commit. Ce commit contient les fichiers de votre projet, mais aussi le fichier de workflow GitHub Actions pour que le workflow soit activé dès le premier push.

git commit -m "Initial commit to activate CI workflow"

**2. Ajouter le fichier GitHub Actions pour la CI**

Avant de pousser quoi que ce soit vers GitHub, vous devez créer un fichier de configuration pour **GitHub Actions** afin d’activer l’intégration continue (CI) dès le premier push. Voici les étapes :

**2.1 Créer le fichier de workflow ci.yml**

1. **Créer un dossier .github/workflows/** : Ce dossier contiendra les fichiers de configuration de votre workflow GitHub Actions.

mkdir -p .github/workflows

1. **Créer un fichier ci.yml dans .github/workflows/** : Ouvrez un éditeur de texte et créez un fichier nommé ci.yml dans le dossier .github/workflows/. Voici un exemple de fichier ci.yml pour un workflow CI utilisant Docker Compose avec Django :

name: Django CI/CD Pipeline with Docker Compose

on:

push:

branches:

- develop # Le workflow sera exécuté lorsqu'un push est effectué sur 'develop'

- master # Le workflow sera exécuté lorsqu'un push est effectué sur master

pull\_request:

branches:

- master # Le workflow sera aussi exécuté lors des pull requests vers 'main'

jobs:

build:

runs-on: ubuntu-latest

services:

docker:

image: docker:19.03.12

options: --privileged

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Set up Docker

uses: docker/setup-buildx-action@v1

- name: Build Docker image

run: |

docker compose -f docker-compose.yml up --build -d

- name: Wait for Django app to be ready

run: |

until curl -s http://localhost:8000 > /dev/null; do

echo "Waiting for Django to start..."

sleep 5

done

- name: Run tests

run: |

docker compose exec app python manage.py test

- name: Stop Docker containers

run: |

docker compose down

1. **Ajouter ce fichier de workflow à votre dépôt** : Après avoir créé le fichier de workflow, vous devez l'ajouter à votre dépôt local.

git add .github/workflows/ci.yml

git commit -m "Add GitHub Actions workflow for CI"

**3. Pousser le premier commit vers GitHub**

Maintenant que le fichier de workflow est prêt, vous pouvez pousser le commit vers GitHub. Cela déclenchera immédiatement le workflow CI sur GitHub.

**3.1 Pousser le commit initial vers GitHub**

1. **Ajouter le dépôt distant GitHub** (si ce n’est pas déjà fait) :

Si vous n'avez pas encore ajouté votre dépôt GitHub distant, exécutez la commande suivante pour l'ajouter (Evidemment vous devez au préalable construire une Repo Github) :

git remote add origin https://github.com/votre-utilisateur/votre-repository.git

1. **Pousser le commit sur la branche main** (ou master si vous l'utilisez) :

Poussez le commit initial qui contient le fichier de workflow vers GitHub. Assurez-vous de pousser vers la branche par défaut (main ou master).

git push -u origin master

GitHub va exécuter le workflow dès que vous poussez ce commit.

**4. Protéger la branche main sur GitHub**

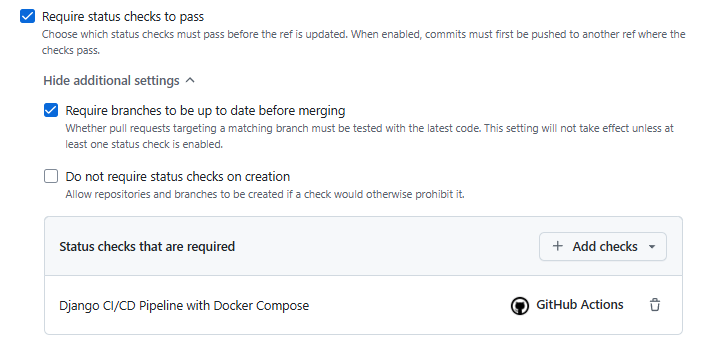
Une fois que le workflow a été exécuté avec succès, vous pouvez configurer la protection de la branche main. Cette étape garantit que les modifications ne peuvent être fusionnées dans main qu'après une revue de code et après avoir passé tous les tests.

**4.1 Accéder à la section "Branch protection rules" de GitHub**

1. Allez dans votre dépôt sur GitHub.
2. Cliquez sur **Settings** (paramètres).
3. Dans le menu latéral gauche, cliquez sur **Branches**.
4. Dans la section **Branch protection rules**, cliquez sur **Add rule**.

**4.2 Configurer la protection pour main**

1. **Branch name pattern** : Entrez main (ou master si c’est votre branche par défaut).
2. Cochez les options suivantes :
   * **Require pull request reviews before merging** : Cela garantit qu'une revue de code est effectuée avant de fusionner une PR dans main.
   * **Require status checks to pass before merging** : Sélectionnez ici le test GitHub Actions pour vous assurer que les tests réussissent avant la fusion dans main. Normalement il s’appelle « Django CI/CD Pipeline with Docker Compose »



1. Cliquez sur **Create** ou **Save changes** pour enregistrer la règle.

Cela signifie que maintenant, toute tentative de push direct sur main sera rejetée et qu'il faudra passer par un pull **request** pour fusionner des modifications, avec la garantie que les tests passent et qu'une revue de code soit effectuée.

**5. Créer et pousser la branche develop**

Après avoir configuré la protection de la branche main, vous pouvez créer une branche develop pour effectuer des modifications et tester avant de fusionner.

1. **Créer la branche develop localement** :

git checkout -b develop

1. **Pousser la branche develop vers GitHub** :

Modifier Une ligne de votre code pour créer une nouvelle version ( par exemple «     name = models.CharField(*max\_length*=100) du fichier models.py peut devenir     name = models.CharField(*max\_length*=150) )

Poussez la branche develop vers GitHub pour qu'elle soit disponible dans votre dépôt distant :

git add .

git commit -m "New Code"

git push -u origin develop

1. **Créer une pull request pour fusionner develop dans main** : Une fois la branche develop poussée, vous pouvez ouvrir une **pull request** sur GitHub pour fusionner les changements dans main.
   * Allez sur votre dépôt sur GitHub.
   * Cliquez sur **Pull Requests**.
   * Cliquez sur **New Pull Request**.
   * Sélectionnez develop comme branche source et main comme branche de destination.
   * Vérifiez les modifications et créez la PR.

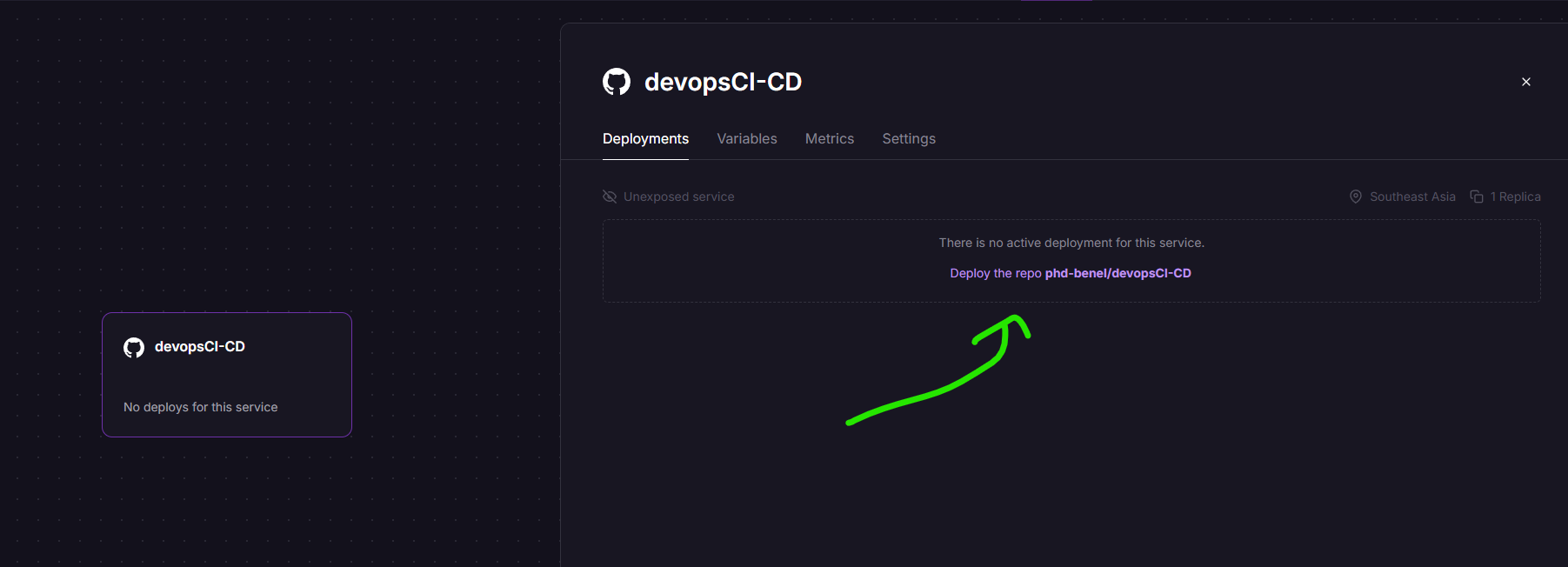
La PR sera soumise pour revue, et les tests GitHub Actions seront exécutés pour vérifier que tout fonctionne correctement.

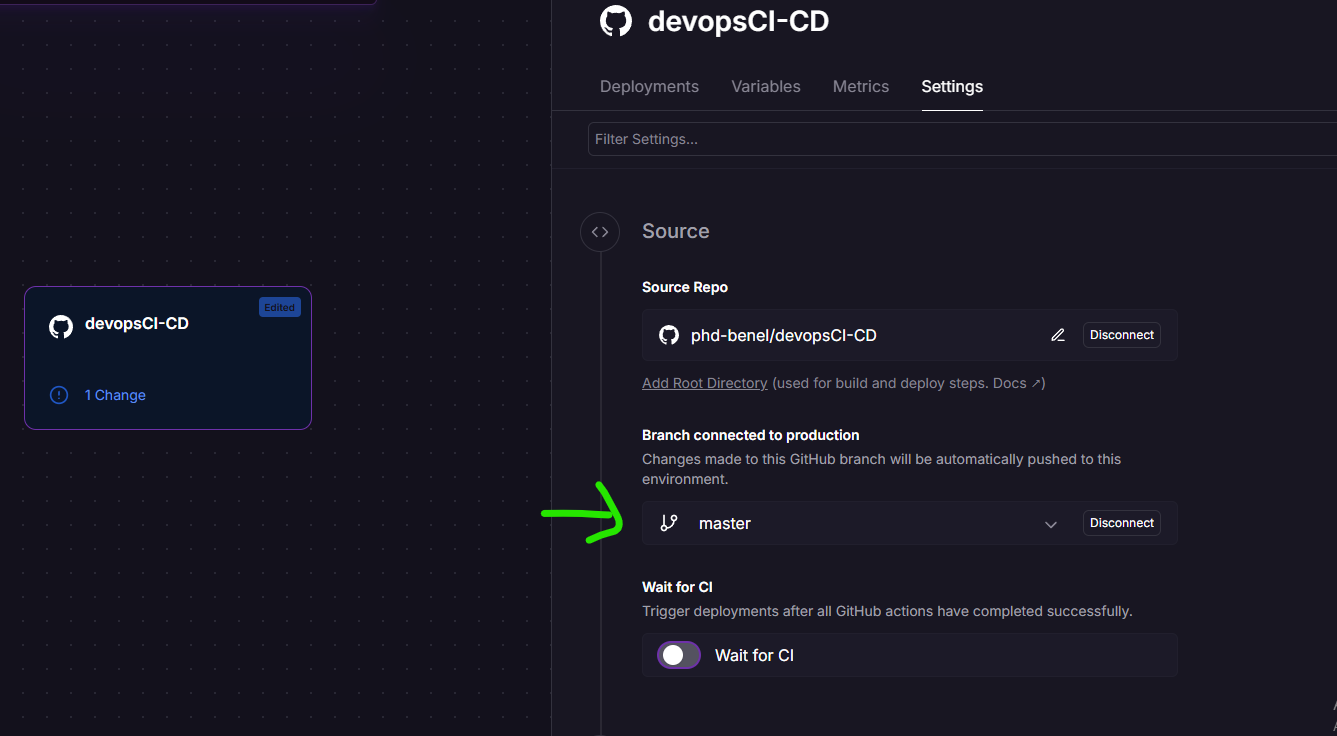
### 5. Configuration de Railway Pour le CD

**Étape 1 : Création et configuration de l'application**

**5.1 Création d'un compte et d'un projet sur Railway**

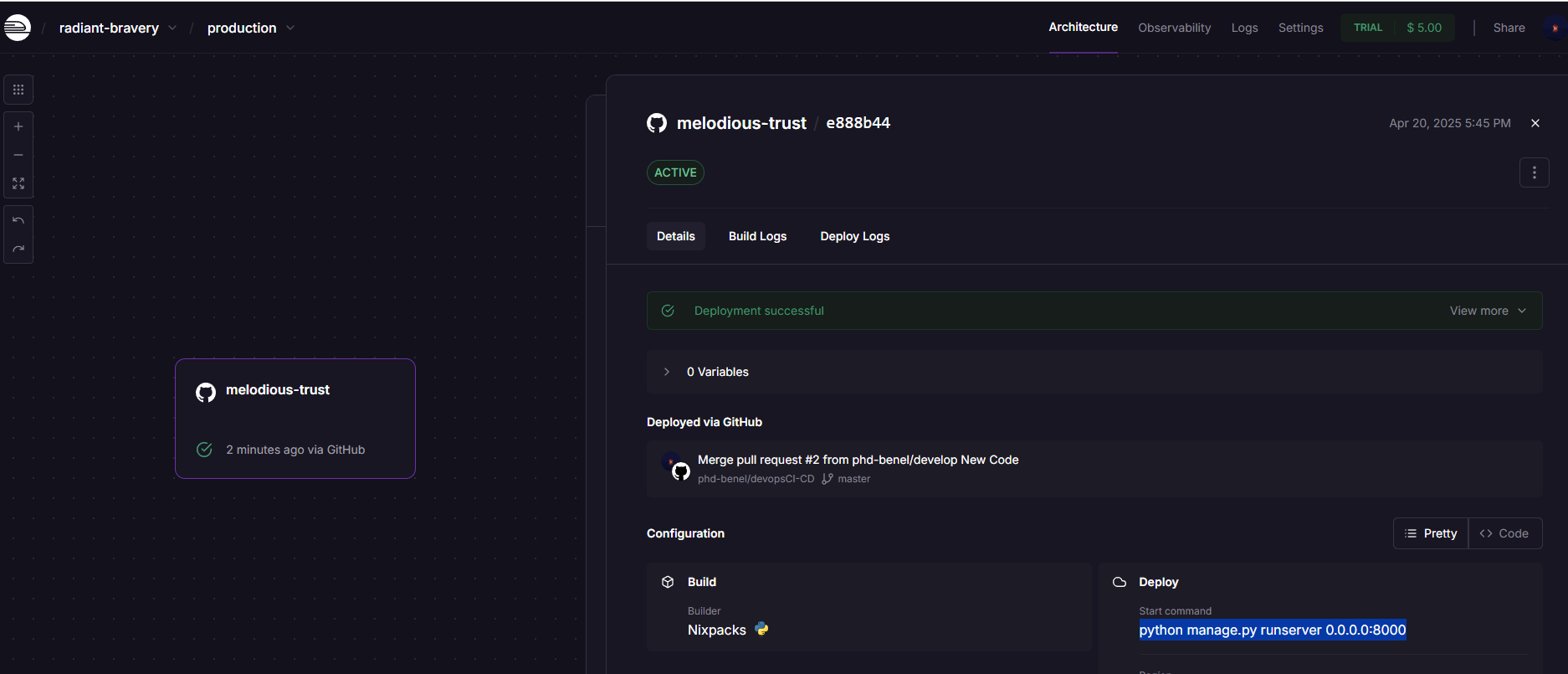
Rendez-vous sur [Railway.app](https://railway.app) et créez un compte. Une fois connecté, cliquez sur **"New Project"** et sélectionnez **"Deploy from GitHub Repo"**. Connectez votre compte GitHub et autorisez Railway à accéder à vos dépôts. Choisissez le dépôt contenant votre projet Django.​



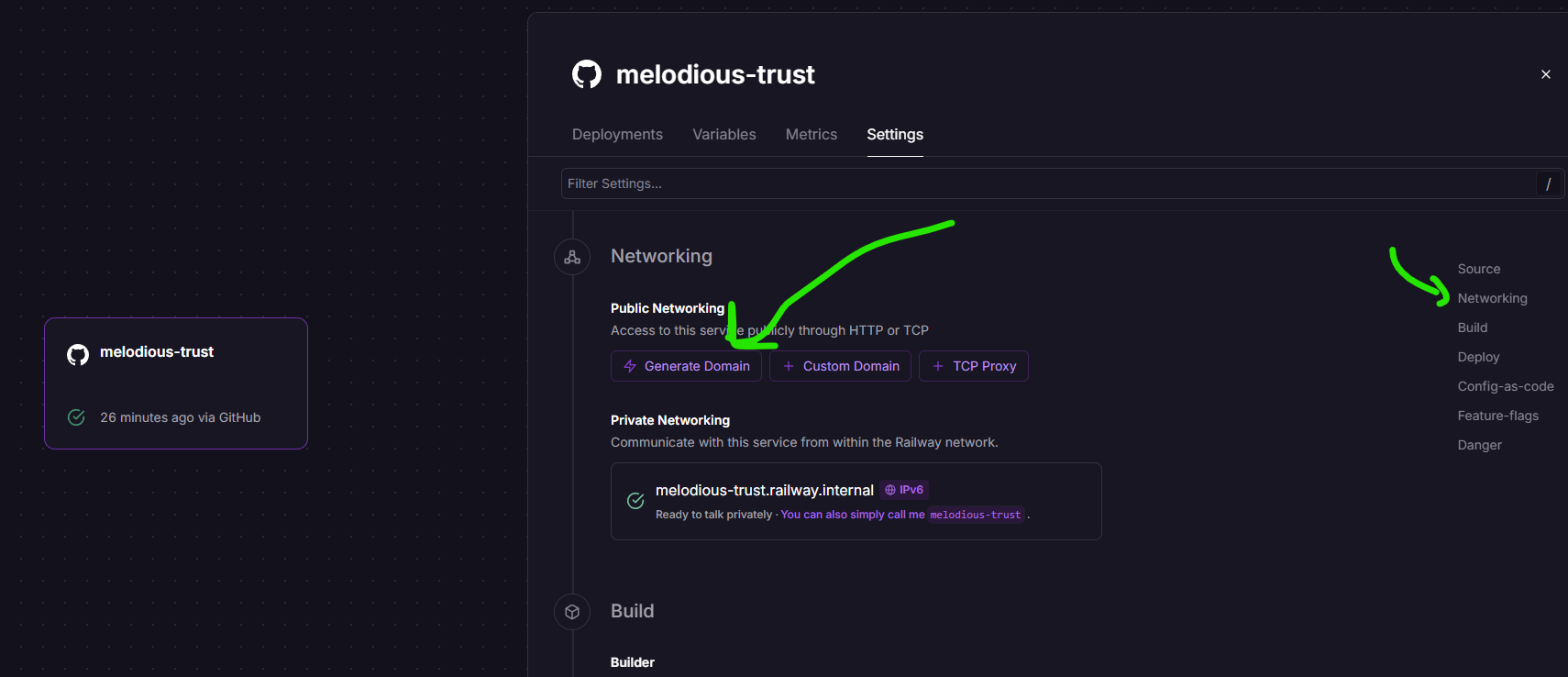


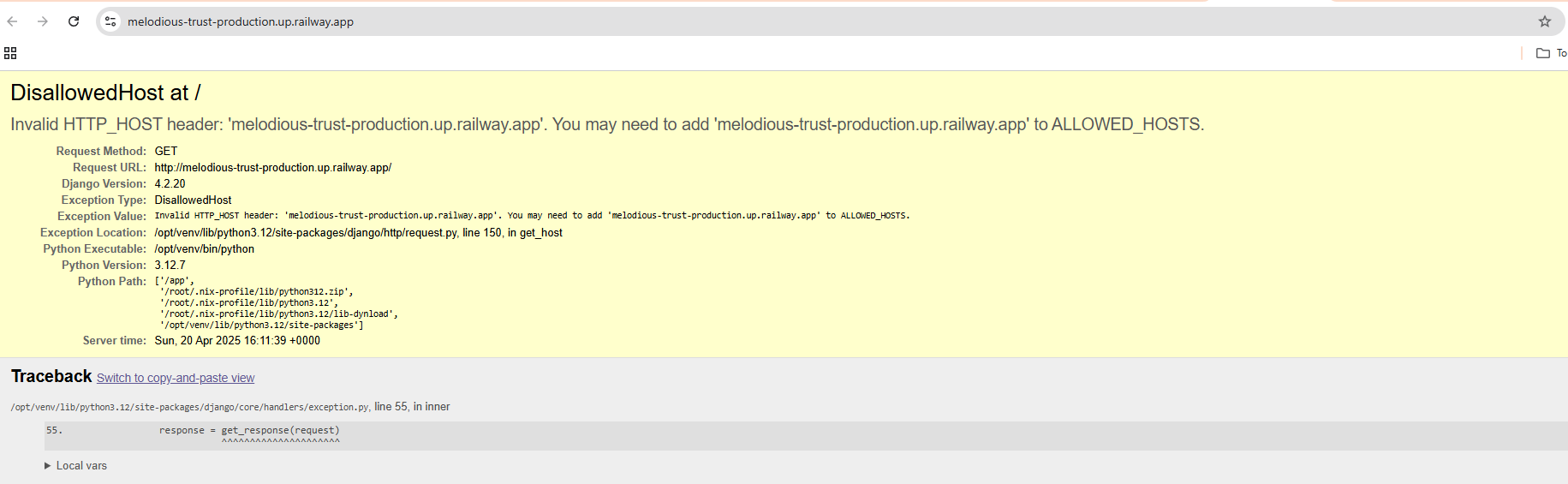
**5.2 Déploiement**

Une fois le projet configuré, Railway déploie automatiquement votre application. Vous pouvez suivre le processus de déploiement en temps réel depuis le tableau de bord Railway.​



Aller sur Settings > Networking > Generate Domain





L'erreur **DisallowedHost** que vous rencontrez se produit lorsque l'application Django reçoit une requête avec un HTTP\_HOST non autorisé. Cela peut se produire si vous ne spécifiez pas les hôtes autorisés dans les paramètres de votre projet Django, en particulier lorsqu'il est déployé sur un environnement de production comme Railway.

**Solution**

Vous devez ajouter l'URL de votre application **melodious-trust-production.up.railway.app** à la liste des hôtes autorisés dans votre fichier settings.py de Django.

1. **Ouvrez votre fichier settings.py** dans le répertoire de votre projet Django.
2. **Trouvez la variable ALLOWED\_HOSTS**. Cette variable est une liste qui définit les hôtes autorisés pour votre application Django.

Ajoutez votre URL Railway à la liste ALLOWED\_HOSTS. Par exemple :

ALLOWED\_HOSTS = ['localhost', '127.0.0.1', 'melodious-trust-production.up.railway.app']

1. **Sauvegardez le fichier settings.py** après avoir ajouté l'URL.
2. **Poussez les modifications vers GitHub** et laissez Railway redéployer votre application avec les modifications.

git add .

git commit -m "Update ALLOWED\_HOSTS to include Railway URL"

git push origin main # Ou la branche appropriée

1. **Vérification** :
   * Une fois le déploiement effectué (faite le Pull Request sur github), essayez d'accéder à votre application via l'URL de Railway : https://melodious-trust-production.up.railway.app/myapp/mymodel/
   * L'application devrait maintenant accepter les requêtes provenant de cette URL sans générer l'erreur DisallowedHost.

**Étape 2 : Test de l'application**

**5.3 Modification de l'interface**

Pour tester le bon fonctionnement de votre déploiement, modifiez l'interface de votre application. Par exemple, changez la couleur de fond de la page d'accueil dans votre fichier HTML.​

**5.4 Visualisation des changements**

Une fois les modifications effectuées, poussez-les vers votre dépôt GitHub. Railway détectera automatiquement ces changements et redéploiera l'application. Accédez à l'URL fournie par Railway pour visualiser les modifications en ligne.​

### 6. **Intégration de Sentry pour le suivi des erreurs (Monitoring)**

**6. Intégration de Sentry pour le suivi des erreurs (Monitoring)**

**Sentry** est une plateforme de gestion des erreurs en temps réel, permettant de capturer, d'analyser et de suivre les exceptions dans votre application. Il fournit des alertes détaillées, des informations sur les erreurs et leur contexte, ce qui aide à résoudre rapidement les problèmes en production.

Bien que **Railway** propose un monitoring de base, en se concentrant principalement sur les logs d'application et la disponibilité, il reste limité par rapport à Sentry. En effet, **Sentry** offre un suivi plus précis des erreurs, avec des alertes en temps réel et un diagnostic plus détaillé des problèmes.

**6.1 Création d'un compte Sentry**

1. Rendez-vous sur [Sentry.io](https://sentry.io/) et créez un compte.
2. Créez un nouveau projet et sélectionnez **Django** comme plateforme.
3. Copiez le **DSN** (Data Source Name) fourni par Sentry.

**6.2 Installation du SDK Sentry**

Dans votre environnement virtuel, installez le SDK Sentry en exécutant la commande suivante :

pip install sentry-sdk

À chaque fois que vous installez via pip, n'oubliez pas de faire un pip freeze > requirements.txt pour mettre à jour votre fichier d'installation

**6.3 Configuration de Sentry dans Django**

Dans le fichier settings.py de votre projet Django, ajoutez la configuration suivante pour intégrer Sentry :

import sentry\_sdk

from sentry\_sdk.integrations.django import DjangoIntegration

sentry\_sdk.init(

dsn="votre-dsn", # Remplacez par le DSN copié depuis Sentry

integrations=[DjangoIntegration()],

traces\_sample\_rate=1.0, # Ajustez le taux de collecte des traces

send\_default\_pii=True

)

**6.4 Déploiement des modifications**

Une fois que Sentry est configuré, poussez les modifications vers votre dépôt GitHub. Railway déploiera automatiquement l'application avec Sentry intégré.

**6.5 Vérification des erreurs sur Sentry**

1. Connectez-vous à votre tableau de bord Sentry.
2. Vous devriez voir les exceptions générées dans la section **"Issues"**. Cela confirme que Sentry capte correctement les erreurs de votre application.

**6.6 Simulation d'une exception dans Django pour Sentry**

**6.6.1 Création d'une vue de test**

Dans votre application Django, créez une vue qui génère une exception afin de la capturer dans Sentry :

from django.http import HttpResponseServerError

def trigger\_error(request):

division\_by\_zero = 1 / 0

return HttpResponseServerError("An error occurred.")

**6.6.2 Ajout de l'URL correspondante**

Dans le fichier urls.py de votre application, ajoutez une route pour cette vue de test :

from django.urls import path

from . import views

urlpatterns = [

path('sentry-debug/', views.trigger\_error),

]

**6.6.3 Test de l'erreur**

Une fois que vous avez configuré l'URL pour déclencher l'exception dans votre application Django, vous devez procéder comme suit :

1. **Pousser les modifications vers GitHub** :
   * Commencez par pousser les changements dans votre dépôt GitHub pour activer le déploiement sur Railway. Cela garantit que les modifications effectuées (y compris la création de la vue et l'ajout de l'URL pour générer une erreur) sont prises en compte.

Exécutez ces commandes dans votre terminal :

pip freeze > requirements.txt

git add .

git commit -m "Add Sentry error trigger"

git push origin develop

1. **Vérification sur Railway** :
   * Railway détectera automatiquement les modifications poussées vers GitHub et redéploiera l'application avec la vue d'erreur en place. Une fois le déploiement terminé, accédez à l'URL fournie par Railway pour tester la vue : http://<votre-domaine>/sentry-debug/.
2. **Visualiser l'erreur sur Sentry** :
   * Accédez à votre tableau de bord Sentry pour visualiser l'exception. L'erreur générée par la division par zéro sera capturée et affichée dans la section **"Issues"** de Sentry. Vous pourrez y consulter les détails de l'exception, tels que la trace de la pile et le contexte de l'erreur.

Cela vous permet de vérifier que Sentry capture correctement les erreurs générées en production et d'assurer une gestion adéquate des exceptions dans votre application.

**Conclusion**

Dans ce TP, vous avez mis en place un cycle DevOps complet pour une application Django. Voici les étapes clés que vous avez réalisées :

1. **Création de l'application Django** : Vous avez développé une application Django avec un modèle et une vue pour afficher des données.
2. **Dockerisation** : L'application a été dockerisée, ce qui permet de la déployer facilement dans des environnements différents.
3. **Automatisation des tests avec GitHub Actions** : Un pipeline CI/CD a été configuré pour tester automatiquement le code à chaque modification.
4. **Déploiement sur Railway** : L'application a été déployée automatiquement sur Railway à chaque mise à jour de la branche principale.
5. **Surveillance des erreurs avec Sentry** : L'intégration de Sentry a permis de suivre les erreurs en temps réel, assurant une gestion efficace des problèmes.

Ce cycle DevOps que vous avez mis en place assure non seulement un développement rapide et fiable, mais il est aussi un excellent exemple de la manière dont les **pratiques DevOps** peuvent améliorer la gestion de projet dans un environnement de développement logiciel moderne. En automatisant les tests, le déploiement et le suivi des erreurs, vous réduisez les risques de régressions, assurez une qualité continue du code et garantissez une mise en production sans heurts.

