

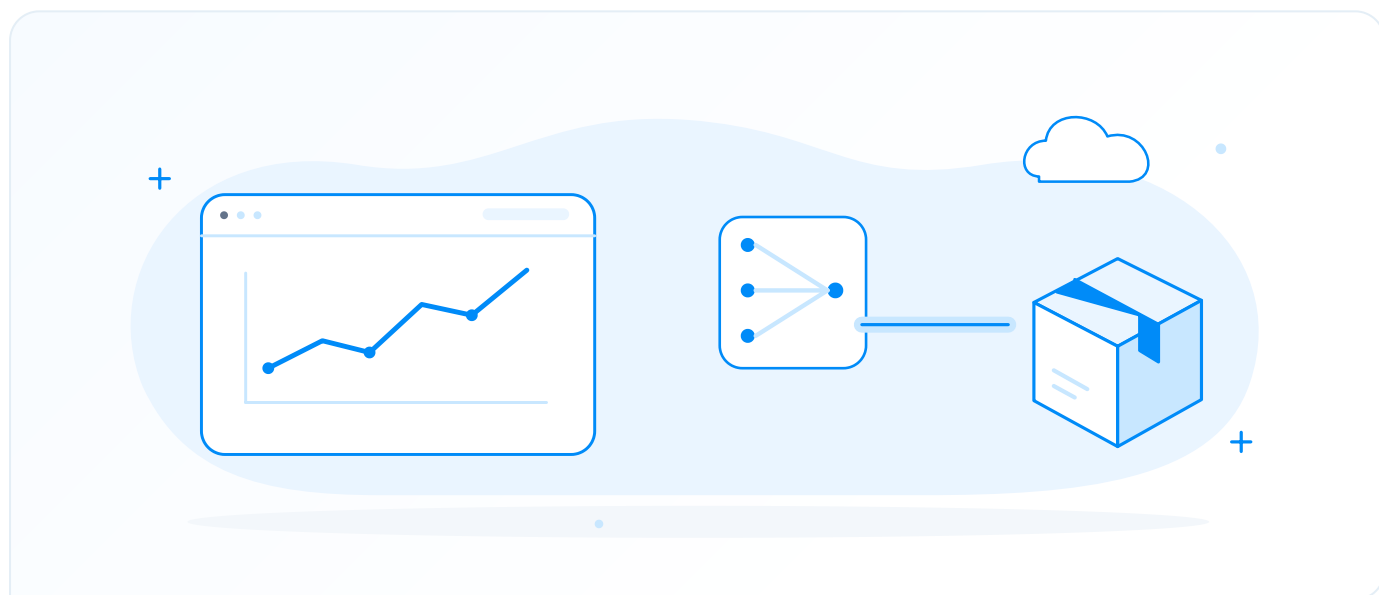
Mini-série

MLOps · API · Conteneurisation

Servir un modèle avec MLflow, FastAPI et Docker

Entraîner un modèle, conserver la trace du run MLflow, exposer une prédiction validée par FastAPI et emballer le service dans une image Docker.

Auteur **Youness Boutyour**

**DURÉE**

3 épisodes · 45 min /
épisode

NIVEAU

Intermédiaire

FORMAT

Mini-série · 3 épisodes

MLOps · API · Conteneurisation · publié le 2026-04-13 · local

Objectifs et livrables

Un modèle entraîné dans un notebook devient vraiment utile lorsqu'une autre application peut l'interroger et retrouver sa provenance. Dans cette mini-série, nous allons accompagner un petit modèle de classification Iris jusqu'à un service HTTP exécuté dans Docker.

Le parcours commence par l'entraînement et l'évaluation. Nous enregistrerons les paramètres, les métriques et l'artefact dans MLflow, puis nous chargerons cet artefact dans une API FastAPI. Le service exposera `/health` pour vérifier son état, `/predict` pour soumettre une observation et `/docs` pour explorer son contrat. Nous construirons ensuite l'image Docker et testerons le conteneur sur le port 8000.

À chaque étape, vous verrez comment conserver le lien entre l'expérience et le modèle servi. Le `run_id` et les métriques permettront d'identifier l'entraînement ; les réponses HTTP et le conteneur permettront de vérifier le fonctionnement du service. Une entrée incorrecte sera également testée pour observer le rejet `422`.

À la fin, vous disposerez d'une chaîne complète que vous pourrez reproduire sur votre machine. Les sorties présentées accompagnent les commandes ; vos propres identifiants et mesures serviront à vérifier quel artefact est effectivement utilisé par l'API.

RÉSULTATS DES ÉTAPES · MLOPS

01 · RUN MLFLOW

```
run_id: 8e4c...91a
accuracy: 0.973
artifact: model.joblib
```

✓ run enregistré
artifacts/model-info.json

02 · API

```
GET /health
status: ok

POST /predict
class_id: 0
```

✓ invalid → HTTP 422

03 · DOCKER

```
docker build
iris-fastapi:1.0

port: 8000
```

✓ healthcheck PASS
image inspect à conserver

Étapes de construction et contrôles du service de prédiction.

RÉSULTATS DE RÉFÉRENCE · IRIS

Run → modèle → API → conteneur

MLFLOW

```
run_id: 8e4c...91a
accuracy: 0.973
artifact: model.joblib
preuve de l'entraînement
```

FASTAPI

```
GET /health
{ "status": "ok" }
POST /predict → class 0
entrée invalide → HTTP 422
```

DOCKER

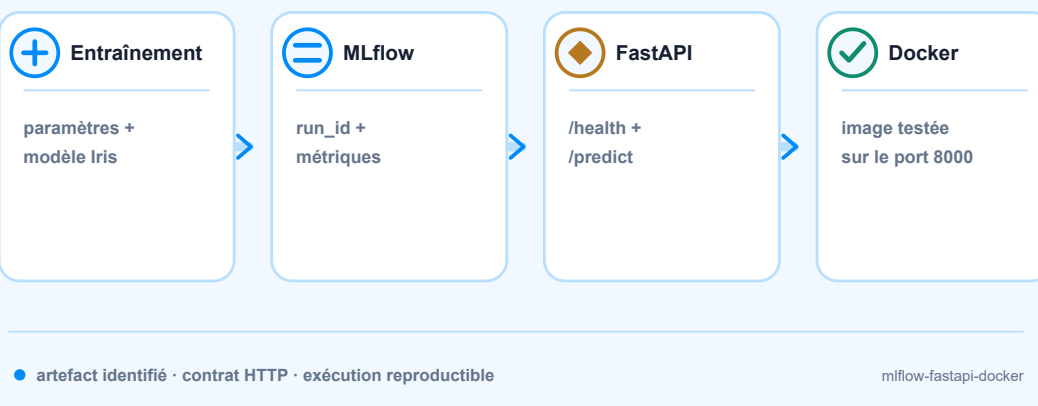
```
image: iris-fastapi:1.0
port: 8000
healthcheck: PASS
artefact reconstruisible
```

Exemple de sortie de validation — conserver les valeurs réellement observées dans le rapport.

Résultats attendus pour relier le modèle, l'API et l'image livrée.

Du modèle entraîné au service livrable

Schéma du lien entre le code d'entraînement, le run MLflow, le contrat FastAPI et l'image Docker.



Chaque étape conserve le lien entre l'artefact produit, le contrat exposé et le service livré.

Lire le flux avant de coder

Le schéma montre une chaîne de responsabilité : le script produit un modèle, MLflow conserve la preuve de l'entraînement, FastAPI expose un contrat et Docker fournit un environnement de livraison. Une erreur de prédiction ne se diagnostique donc pas au même endroit qu'une erreur d'import ou qu'un problème de port réseau.

Le contrôle important est la continuité de l'artefact. Le `run_id` doit permettre de retrouver le modèle chargé par l'API ; la réponse HTTP doit respecter le schéma attendu ; l'image doit démarrer avec les mêmes fichiers. En conservant ces trois repères, vous évitez de tester une API qui servirait par mégarde

un ancien modèle.

1. Préparer l'environnement

Les trois briques auront des rôles distincts : MLflow conserve l'expérience, le script d'entraînement produit l'artefact et FastAPI charge cet artefact pour répondre aux requêtes. Préparez leur environnement commun avant de les lancer. Les dossiers `src/` et `artifacts/` séparent le code des fichiers générés ; les chemins des scripts seront construits à partir du projet.

Vérifiez que l'interpréteur actif est bien celui de l'environnement virtuel, puis conservez les versions installées. Un échec d'import relève de cette préparation, pas encore du modèle ou du serveur HTTP. Les exclusions Git empêchent de versionner l'environnement et les fichiers locaux de suivi ; elles ne remplacent pas la sauvegarde des artefacts que vous souhaitez conserver.

```
Terminal · Bash

mkdir -p mlflow-fastapi-docker/{artifacts,src}
cd mlflow-fastapi-docker
python3 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install mlflow scikit-learn fastapi 'uvicorn[standard]' joblib
python -m pip freeze > requirements.txt
```

Ajoutez un `.gitignore` :

Sortie / exemple

```
.venv/
mlflow.db
mlartifacts/
__pycache__/
*.pyc
```

Dans un premier temps, les artefacts restent locaux. Pour un déploiement partagé, séparez ensuite le backend store, le stockage d'artefacts, les secrets et les droits d'accès.

Sortie obtenue

Sortie / exemple

```
requirements.txt créé
artifacts/ créé
environnement Python activé
```

Commande de vérification

Vérifiez `python --version`, `python -c "import mlflow, fastapi"` et la présence de `requirements.txt`. Ne poursuivez pas si l'installation mélange le Python système et l'environnement virtuel.

2. Démarrer MLflow Tracking

Laissez MLflow tourner dans un premier terminal et utilisez-en un second pour l'entraînement. L'adresse de suivi définie dans le script doit désigner ce serveur. La base SQLite stocke les informations sur les expériences et leurs exécutions ; le dossier d'artefacts reçoit les fichiers associés. Conserver seulement l'un des deux ne suffit pas à retrouver tout le contenu d'une expérience.

Avant d'entraîner, ouvrez l'interface et vérifiez que le serveur répond. Après l'entraînement, vous y chercherez une nouvelle exécution avec ses paramètres et ses métriques. Si l'interface semble vide, comparez l'URI utilisée par le script avec celle du serveur : écrire dans un autre stockage local peut réussir sans alimenter l'interface que vous consultez.

Dans un premier terminal :

```
Terminal · Bash

source .venv/bin/activate
mlflow server \
  --host 127.0.0.1 \
  --port 5000 \
  --backend-store-uri sqlite:///mlflow.db \
  --default-artifact-root ./mlartifacts
```

Vérifiez que l'interface répond sur <http://127.0.0.1:5000>. Laissez ce terminal ouvert. Le serveur doit être démarré avant l'entraînement afin que le script enregistre réellement son run à distance via HTTP.

Sortie obtenue

```
Terminal · Bash

curl --fail http://127.0.0.1:5000/health
```

Sortie / exemple

Le serveur répond avec HTTP 200 ou l'interface MLflow s'ouvre dans le navigateur.

Commande de vérification

Capturez l'écran d'accueil MLflow ou conservez la réponse HTTP avant `python src/train.py`, sinon le run peut être enregistré dans un autre backend.

3. Entraîner et enregistrer le modèle

Le découpage du jeu Iris réserve une partie des exemples à l'évaluation et utilise la stratification pour conserver la répartition des classes. Le modèle est ajusté sur l'ensemble d'entraînement uniquement ; les prédictions servant aux métriques proviennent du lot de test. La graine fixe stabilise ce découpage pour faciliter la comparaison entre vos essais.

Suivez les deux sorties de l'exécution : MLflow reçoit les paramètres, métriques et artefacts de suivi, tandis que `artifacts/model.joblib` sera chargé par l'API. Le fichier d'informations relie cette copie au run et aux scores observés. Un score élevé sur ce petit jeu vérifie le déroulement de la chaîne ; il ne mesure pas la robustesse du service sur des données différentes.

Créez `src/train.py` :

```
Python

from pathlib import Path
import json

import joblib
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, f1_score
from sklearn.model_selection import train_test_split

ROOT = Path(__file__).resolve().parents[1]
mlflow.set_tracking_uri("http://127.0.0.1:5000")
mlflow.set_experiment("iris-fastapi")

dataset = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    dataset.data,
    dataset.target,
    test_size=0.2,
    random_state=42,
    stratify=dataset.target,
)

params = {"C": 1.0, "max_iter": 300, "solver": "lbfgs", "random_state": 42}
model = LogisticRegression(**params)

with mlflow.start_run(run_name="iris-baseline") as run:
```

```
model.fit(X_train, y_train)
predictions = model.predict(X_test)
metrics = {
    "accuracy": accuracy_score(y_test, predictions),
    "f1_weighted": f1_score(y_test, predictions, average="weighted"),
}
mlflow.log_params(params)
mlflow.log_metrics(metrics)
mlflow.sklearn.log_model(model, "model")

(ROOT / "artifacts").mkdir(exist_ok=True)
joblib.dump(model, ROOT / "artifacts" / "model.joblib")
(ROOT / "artifacts" / "model-info.json").write_text(
    json.dumps({"run_id": run.info.run_id, "metrics": metrics}, indent=2) + "\n",
    encoding="utf-8",
)
print(json.dumps({"run_id": run.info.run_id, "metrics": metrics}, indent=2))
```

Exécutez :

Terminal · Bash

```
python src/train.py
cat artifacts/model-info.json
```

La commande affiche un `run_id`, une accuracy proche de `0.9` ou supérieure et crée `artifacts/model.joblib`. La valeur exacte dépend de la version des bibliothèques et doit être reprise depuis la sortie réelle.

Dans l'interface MLflow, vérifiez que le run contient les paramètres, les métriques et l'artefact `model`. C'est la première figure à capturer pour le tutoriel : elle doit montrer le run réellement produit par votre commande.

Sortie obtenue

Sortie / exemple

```
run_id: <32 caractères hexadécimaux>
metrics: accuracy=<valeur>, f1_weighted=<valeur>
artifacts/model.joblib: présent
```

Commande de vérification

Capturez la page du run MLflow affichant le nom `iris-baseline`, les paramètres, les métriques et l'artefact. Ajoutez `artifacts/model-info.json` au rapport : la capture seule ne suffit pas à relier le modèle au run.

4. Créer l'API FastAPI

Le modèle est chargé lors de l'import du module, avant de traiter les requêtes. L'API n'interroge donc pas MLflow à chaque prédiction : elle utilise le fichier local préparé précédemment. Si ce fichier manque, revenez à l'entraînement et vérifiez son chemin plutôt que de modifier le corps de la requête HTTP.

Les quatre mesures doivent être transmises dans l'ordre attendu par le modèle. La validation Pydantic impose des valeurs positives, mais ne garantit pas qu'elles correspondent à une fleur ou à la bonne unité. Le résultat `class_id` est un identifiant de classe, pas une probabilité. Testez une requête valide et une valeur refusée ; `/health` seul ne vérifie pas la qualité des prédictions.

Créez `src/api.py` :

Python

```
from pathlib import Path

import joblib
from fastapi import FastAPI
from pydantic import BaseModel, Field

ROOT = Path(__file__).resolve().parents[1]
model = joblib.load(ROOT / "artifacts" / "model.joblib")
app = FastAPI(title="Iris model API", version="1.0.0")

class IrisInput(BaseModel):
    sepal_length: float = Field(gt=0)
    sepal_width: float = Field(gt=0)
    petal_length: float = Field(gt=0)
    petal_width: float = Field(gt=0)

@app.get("/health")
def health() → dict[str, str]:
    return {"status": "ok", "model": "iris-baseline"}

@app.post("/predict")
def predict(payload: IrisInput) → dict[str, int]:
    values = [
        payload.sepal_length,
        payload.sepal_width,
        payload.petal_length,
        payload.petal_width,
    ]
    prediction = int(model.predict(values)[0])
    return {"class_id": prediction}
```

Lancez l'API :

```
Terminal · Bash  
  
uvicorn src.api:app --host 127.0.0.1 --port 8000 --reload
```

Testez le service :

```
Terminal · Bash  
  
curl http://127.0.0.1:8000/health  
curl -X POST http://127.0.0.1:8000/predict \  
-H 'content-type: application/json' \  
-d '{"sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4, "petal_width": 0.2}'
```

Ouvrez <http://127.0.0.1:8000/docs> et capturez la documentation OpenAPI réelle. Testez aussi une valeur négative pour vérifier que FastAPI refuse l'entrée avec une erreur `422`.

Sortie obtenue

```
Sortie / exemple  
  
GET /health → 200 {"status":"ok","model":"iris-baseline"}  
POST /predict → 200 {"class_id": 0}  
entrée négative → 422 Unprocessable Entity
```

Commande de vérification

Capturez `/docs` avec les deux routes visibles et conservez les trois réponses HTTP. La capture doit montrer le service réellement lancé, pas seulement le code source.

5. Construire l'image Docker

Le conteneur embarque l'API et l'artefact déjà entraîné ; il ne relance ni l'entraînement ni le serveur MLflow. Copier les dépendances avant le code permet de réutiliser cette couche lorsque seuls les fichiers applicatifs changent. Le modèle inclus dans le contexte de construction doit être celui dont vous avez examiné les métriques et les informations de run.

Arrêtez l'API locale si elle utilise déjà le port 8000, puis démarrez le conteneur et rejouez la même requête. L'écoute sur `0.0.0.0` se fait à l'intérieur du conteneur ; la publication du port permet l'accès depuis l'hôte. Comparez les prédictions des deux exécutions et vérifiez les versions des dépendances, notamment si le Python de l'image diffère de celui de l'entraînement.

Créez `Dockerfile` :

```
FROM python:3.12-slim

WORKDIR /app
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY src ./src
COPY artifacts ./artifacts

EXPOSE 8000
CMD ["uvicorn", "src.api:app", "--host", "0.0.0.0", "--port", "8000"]
```

Construisez et démarrez :

```
Terminal · Bash

docker build -t iris-fastapi:1.0 .
docker run --rm -p 8000:8000 iris-fastapi:1.0
```

Le service doit maintenant répondre depuis <http://127.0.0.1:8000>. Conservez la sortie de `docker image inspect iris-fastapi:1.0` et le tag utilisé dans le compte rendu.

Sortie obtenue

```
Terminal · Bash

curl --fail http://127.0.0.1:8000/health
docker image inspect iris-fastapi:1.0 --format '{{.Id}}'
```

```
Sortie / exemple

{"status":"ok","model":"iris-baseline"}
sha256:<empreinte-de-votre-image>
```

Commande de vérification

Capturez le terminal où le conteneur répond et conservez l'empreinte de l'image. Cette sortie établit que l'artefact entraîné a été inclus dans l'image et servi par le processus Docker.

Exemples de sorties

Conservez au minimum les sorties suivantes :

```
Sortie / exemple

$ curl http://127.0.0.1:8000/health
{"status":"ok","model":"iris-baseline"}

$ curl -X POST .../predict -d '{"...}'
{"class_id":0}

$ docker image inspect iris-fastapi:1.0 --format '{{.Id}}'
sha256:<empreinte-de-votre-image>
```

Dans MLflow, vérifiez que le run contient les paramètres, les métriques et l'artefact référencé par `artifacts/model-info.json`. La réponse API et l'image sont des contrôles complémentaires : aucune des deux ne remplace la traçabilité de l'entraînement.

Dépannage et diagnostics

Symptôme	Cause probable	Vérification
<code>FileNotFoundError</code> sur <code>model.joblib</code>	l'entraînement n'a pas été exécuté ou le dossier n'est pas copié	relancer <code>python src/train.py</code> , puis vérifier <code>artifacts/</code>
<code>connection refused</code> sur MLflow	serveur non lancé ou mauvais port	tester <code>curl http://127.0.0.1:5000/health</code> et vérifier <code>MLFLOW_TRACKING_URI</code>
HTTP <code>422</code>	schéma d'entrée invalide	envoyer quatre nombres strictement positifs et conserver la réponse d'erreur
l'image répond mais le modèle manque	artefact exclu du contexte Docker	vérifier <code>COPY artifacts ./artifacts</code> et reconstruire sans cache

Vérification finale — chaîne de traçabilité

Avant de déclarer le service reproductible, vérifiez :

- le run MLflow est retrouvable avec son `run_id` ;
- le modèle embarqué correspond bien à ce run ;

- l'API refuse les entrées invalides ;
- l'image Docker peut être reconstruite à partir du `Dockerfile` et de `requirements.txt` ;
- la version du modèle et les métriques sont visibles dans `artifacts/model-info.json` ;
- la commande et la réponse de `/health` sont conservées dans le rapport.

Limites et prolongements

Ce tutoriel utilise un artefact local et un modèle simple. Pour un déploiement réel, ajoutez une gestion de secrets, un registre d'images, des tests d'intégration, une authentification, une limite de débit, des traces et une procédure de retour à la version précédente. Le fait que l'API réponde ne prouve pas que le modèle est acceptable : la décision doit rester liée aux métriques, aux données et au contexte d'usage.

Références

- [MLflow Tracking — documentation officielle](#)
- [FastAPI — documentation officielle](#)
- [Docker — documentation officielle](#)