

🚀 TP 3 : Base de Données et Déploiement en Production

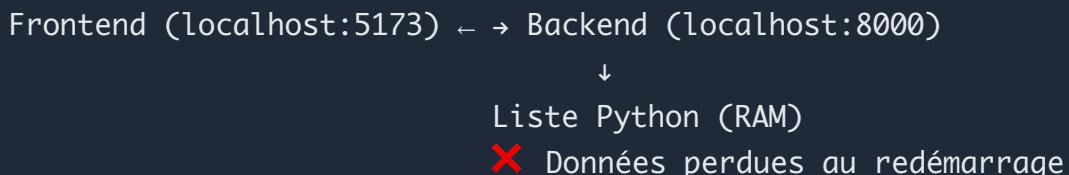
🎯 Objectifs de l'Atelier

À la fin de cet atelier, vous aurez :

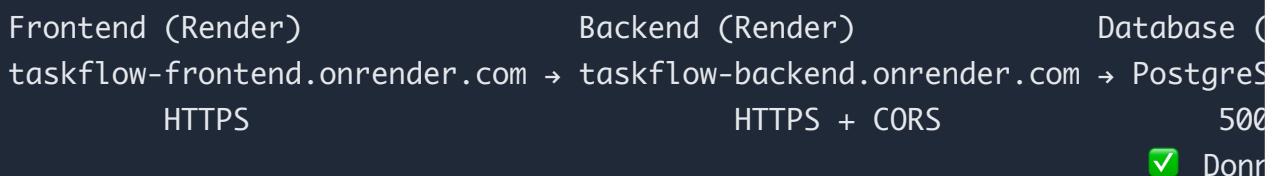
1. Migré vers **PostgreSQL** avec SQLAlchemy ORM
 2. Déployé automatiquement avec **render.yaml** (Infrastructure as Code)
 3. Ajouté de **nouvelles fonctionnalités** (comptage de tâches)
 4. Vérifié le **déploiement automatique** (Continuous Deployment)
-

📦 Architecture Cible

Avant (Local - Stockage en mémoire) :



Après (Production avec PostgreSQL) :



✍ Exercice 1 : Installer les Dépendances PostgreSQL

Objectif

Ajouter SQLAlchemy et le driver PostgreSQL au backend.

Instructions

Ajoutez les packages nécessaires :

```
cd backend  
uv add sqlalchemy psycopg2-binary
```

💡 Ce que font ces packages

- **sqlalchemy** : ORM (Object-Relational Mapping) pour Python - permet de manipuler la base de données avec des objets Python
 - **psycopg2-binary** : Driver PostgreSQL - permet à Python de se connecter à PostgreSQL
-

✍ Exercice 2 : Configurer la Base de Données

Objectif

Créer le fichier de configuration pour la connexion à la base de données.

Instructions

Créez le fichier `backend/src/database.py` :

```
import os  
from sqlalchemy import create_engine  
from sqlalchemy.orm import sessionmaker, declarative_base  
  
# Lire l'URL de la base de données depuis les variables d'environnement  
DATABASE_URL = os.getenv("DATABASE_URL", "sqlite:///taskflow.db")
```

```

# Configuration du moteur SQLAlchemy
if DATABASE_URL.startswith("sqlite"):
    # SQLite (développement local)
    engine = create_engine(
        DATABASE_URL,
        connect_args={"check_same_thread": False}
    )
else:
    # PostgreSQL (production)
    engine = create_engine(
        DATABASE_URL,
        pool_size=5,
        max_overflow=10,
        pool_pre_ping=True
    )

# Factory de sessions
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Base pour les modèles ORM
Base = declarative_base()

def get_db():
    """Générateur qui fournit une session de base de données."""
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

def init_db():
    """Initialise la base de données en créant toutes les tables."""
    from . import models # Import des modèles pour créer les tables
    Base.metadata.create_all(bind=engine)

```

Points importants

- **DATABASE_URL** : URL de connexion (SQLite en local, PostgreSQL en production)

- **Pool de connexions** : Réutilise les connexions pour améliorer les performances
 - `pool_pre_ping` : Vérifie que la connexion est vivante avant de l'utiliser
-

✍ Exercice 3 : Créer le Modèle de Données

Objectif

Définir le schéma de la table `tasks` avec SQLAlchemy ORM.

Instructions

Créez le fichier `backend/src/models.py` :

```
from enum import Enum
from sqlalchemy import Column, String, DateTime, Enum as SQLEnum
from sqlalchemy.sql import func

from .database import Base


class TaskStatus(str, Enum):
    """Statuts possibles d'une tâche."""
    TODO = "todo"
    IN_PROGRESS = "in_progress"
    DONE = "done"


class TaskPriority(str, Enum):
    """Priorités possibles d'une tâche."""
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"


class TaskModel(Base):
    """Modèle SQLAlchemy pour la table tasks."""
    __tablename__ = "tasks"
```

```
id = Column(String, primary_key=True, index=True)
title = Column(String(200), nullable=False)
description = Column(String(1000), nullable=True)
status = Column(SQLEnum(TaskStatus), default=TaskStatus.TODO)
priority = Column(SQLEnum(TaskPriority), default=TaskPriority.MEDIUM)
assignee = Column(String(100), nullable=True)
due_date = Column(DateTime, nullable=True)
created_at = Column(DateTime, server_default=func.now())
updated_at = Column(DateTime, server_default=func.now(), onupdate=fun
```

💡 Avantages de l'ORM

- Pas besoin d'écrire du SQL directement
- Type-safety avec Python
- Migrations de schéma facilitées
- Timestamps automatiques (`created_at` , `updated_at`)

✍ Exercice 4 : Migrer l'Application vers PostgreSQL

Objectif

Adapter `app.py` pour utiliser SQLAlchemy au lieu du stockage en mémoire.

Contexte

Actuellement, `app.py` stocke les tâches dans un dictionnaire Python (`tasks_db`). Au redémarrage du serveur, toutes les données sont perdues. Nous allons migrer vers SQLAlchemy pour persister les données dans PostgreSQL.

Instructions

Étape 1 : Mettre à jour les imports

Ouvrez `backend/src/app.py` et **ajoutez** les imports nécessaires (gardez les imports existants !) :

```
from contextlib import asynccontextmanager
import uuid
from fastapi import Depends
from sqlalchemy.orm import Session
from sqlalchemy import text

from .database import get_db, init_db
from .models import TaskModel, TaskStatus, TaskPriority
```

Pourquoi ces imports ?

- `Depends` : Injection de dépendances FastAPI pour la session DB
- `Session` : Type de la session SQLAlchemy
- `text` : Pour exécuter du SQL brut (health check)
- `uuid` : Pour générer des identifiants uniques

Étape 2 : Nettoyer le code obsolète

Cherchez et supprimez ces éléments dans `app.py` :

```
# ❌ SUPPRIMER : Ces classes (lignes ~31-42)
class TaskStatus(str, Enum):
    TODO = "todo"
    ...
    ...

class TaskPriority(str, Enum):
    LOW = "low"
    ...
    ...

# ❌ SUPPRIMER : Le stockage en mémoire (lignes ~78-79)
tasks_db: Dict[int, Task] = {}
next_id = 1

# ❌ SUPPRIMER : Ces fonctions (lignes ~82-94)
def get_next_id() -> int:
    ...
    ...

def clear_tasks():
    ...
```

```
# △ À MODIFIER : Le health check (sera réécrit à l'étape 6)
@app.get("/health")
async def health_check():
    return {"status": "healthy", "tasks_count": len(tasks_db)}
```

 Ces éléments sont maintenant dans `models.py` ou remplacés par SQLAlchemy.

Étape 3 : Remplacer le système de démarrage

Cherchez et supprimez les anciens handlers :

```
# ❌ SUPPRIMER ces deux fonctions (lignes ~129-139)
@app.on_event("startup")
def startup():
    ...

@app.on_event("shutdown")
def shutdown():
    ...
```

Remplacez-les par le nouveau système `lifespan` (à placer AVANT la création de `app`) :

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    """Lifecycle manager - initialise la DB au démarrage."""
    logger.info("🚀 TaskFlow backend starting up...")
    init_db() # Crée les tables
    logger.info("✅ Database initialized")
    yield
    logger.info("🔴 TaskFlow backend shutting down...")

app = FastAPI(
    title="TaskFlow API",
    ...
    lifespan=lifespan, # ← Ajouter cette ligne
)
```

Étape 3b : Ajouter le middleware CORS

Ajoutez ces imports en haut du fichier (avec les autres imports) :

```
from fastapi.middleware.cors import CORSMiddleware  
import os
```

Puis ajoutez ce code **juste après** `app = FastAPI(...)` :

```
# Configuration CORS pour le frontend  
cors_origins_str = os.getenv("CORS_ORIGINS", "http://localhost:5173,http:  
cors_origins = [origin.strip() for origin in cors_origins_str.split(",")]  
  
app.add_middleware(  
    CORSMiddleware,  
    allow_origins=cors_origins,  
    allow_credentials=True,  
    allow_methods=["*"],  
    allow_headers=["*"],  
)
```

⚠️ Important : Le middleware CORS est essentiel pour que le frontend puisse communiquer avec le backend en production. La variable `CORS_ORIGINS` sera configurée sur Render à l'exercice 7.

Étape 4 : Modifier la classe Task existante

Cherchez la classe `Task` dans `app.py` et remplacez-la :

```
# ❌ AVANT (ne fonctionne plus avec SQLAlchemy)  
class Task(TaskCreate):  
    id: int # ← int ne marche pas avec UUID  
    created_at: datetime  
    updated_at: datetime  
  
# ✅ APRÈS (compatible SQLAlchemy)  
class Task(BaseModel):  
    """Model for task response."""  
    id: str # ← Changé en str pour UUID  
    title: str
```

```

description: Optional[str] = None
status: TaskStatus
priority: TaskPriority
assignee: Optional[str] = None
due_date: Optional[datetime] = None
created_at: datetime
updated_at: datetime

class Config:
    from_attributes = True # Permet la conversion depuis SQLAlchemy

```

Pourquoi ces changements ?

- `id: str` au lieu de `int` → les UUID sont des strings
- `from_attributes = True` → permet à Pydantic de lire les objets SQLAlchemy

Étape 5 : Modifier les endpoints

Pour chaque endpoint, ajoutez `db: Session = Depends(get_db)` comme paramètre.

Exemple simple avec health check :

```

# 🔞 AVANT (ne fonctionne plus)
@app.get("/health")
async def health_check():
    return {"status": "healthy", "tasks_count": len(tasks_db)}

# ✅ APRÈS (avec SQLAlchemy)
@app.get("/health")
async def health_check(db: Session = Depends(get_db)):
    """Health check with database status."""
    try:
        db.execute(text("SELECT 1"))
        tasks_count = db.query(TaskModel).count()
        return {
            "status": "healthy",
            "database": "connected",
            "tasks_count": tasks_count
        }
    
```

```

except Exception as e:
    return {"status": "unhealthy", "database": str(e)}

```

⚠️ Attention : `db: Session = Depends(get_db)` doit être dans les paramètres de la fonction, pas dans le corps !

Exemple complet avec GET /tasks :

```

@app.get("/tasks", response_model=List[Task])
async def get_tasks(
    status: Optional[TaskStatus] = None,
    priority: Optional[TaskPriority] = None,
    assignee: Optional[str] = None,
    db: Session = Depends(get_db) # ← Toujours en dernier dans les param
):
    """Get all tasks with optional filtering."""
    query = db.query(TaskModel)

    if status:
        query = query.filter(TaskModel.status == status)
    if priority:
        query = query.filter(TaskModel.priority == priority)
    if assignee:
        query = query.filter(TaskModel.assignee == assignee)

    return query.all()

```

⚠️ Ne confondez pas `Task` et `TaskModel` !

Modèle	Type	Utilisation
<code>Task</code>	Pydantic	<code>response_model=Task</code> (réponses API)
<code>TaskModel</code>	SQLAlchemy	<code>db.query(TaskModel)</code> , <code>TaskModel(...)</code> (opérations DB)

Règle simple : Pour tout ce qui touche à la base de données → `TaskModel`

Adaptez chaque endpoint selon ce tableau :

Endpoint	Logique SQLAlchemy
GET /tasks	<code>db.query(TaskModel).all()</code>
GET /tasks/{id}	<code>db.query(TaskModel).filter(TaskModel.id == task_id).first()</code>
POST /tasks	<code>TaskModel(id=str(uuid.uuid4()), ...)</code> → <code>db.add()</code> → <code>db.commit()</code> → <code>db.refresh()</code>
PUT /tasks/{id}	<code>setattr(task, field, value)</code> → <code>db.commit()</code> → <code>db.refresh(task)</code>
DELETE /tasks/{id}	<code>db.delete(task)</code> → <code>db.commit()</code>

💡 Pattern SQLAlchemy :

1. `db.add(obj)` - Ajoute à la session
2. `db.commit()` - Sauvegarde en base
3. `db.refresh(obj)` - Recharge les valeurs générées (timestamps, etc.)

Indice POST /tasks : Créez un `TaskModel` (pas `Task`) avec `id=str(uuid.uuid4())` au lieu de `get_next_id()`

💡 Points clés à retenir

Concept	Explication
<code>Depends(get_db)</code>	FastAPI injecte automatiquement une session DB
<code>db.commit()</code>	Obligatoire pour sauvegarder les changements
<code>db.refresh()</code>	Recharge l'objet avec les valeurs de la DB (timestamps)
<code>from_attributes = True</code>	Permet à Pydantic de lire les attributs SQLAlchemy

Checkpoint

Testez localement :

```
cd backend
uv run unicorn src.app:app --reload

# Dans un autre terminal
curl http://localhost:8000/health
curl http://localhost:8000/tasks
```

Vous devriez voir un fichier `taskflow.db` créé dans `backend/`

Exercice 5 : Adapter les Tests

Objectif

Modifier les tests pour utiliser une base de données SQLite temporaire.

Contexte

Les tests utilisent actuellement `clear_tasks()` qui n'existe plus. Nous devons créer une base de données de test isolée et nettoyer les données entre chaque test.

Instructions

Étape 1 : Ajouter les imports

Ouvrez `backend/tests/conftest.py` et ajoutez :

```
import tempfile
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.pool import StaticPool

from src.database import Base, get_db
from src.models import TaskModel
```

Étape 2 : Configurer la base de test

Créez une base SQLite temporaire :

```
TEST_DB_FILE = tempfile.mktemp(suffix=".db")
TEST_DATABASE_URL = f"sqlite:///{{TEST_DB_FILE}}"

test_engine = create_engine(
    TEST_DATABASE_URL,
    connect_args={"check_same_thread": False},
    poolclass=StaticPool,
)

TestSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=t
```

Étape 3 : Créer les fixtures

Ajoutez ces 3 fixtures dans `conftest.py` :

```
@pytest.fixture(scope="session")
def setup_test_database():
    """Crée les tables une seule fois pour tous les tests."""
    Base.metadata.create_all(bind=test_engine)
    yield
    Base.metadata.drop_all(bind=test_engine)

@pytest.fixture(autouse=True)
def clear_test_data(setup_test_database):
    """Nettoie les données entre chaque test."""
    db = TestSessionLocal()
    db.query(TaskModel).delete()
    db.commit()
    db.close()

@pytest.fixture
def client(setup_test_database):
    """Client de test avec base de données isolée."""
    def override_get_db():
        db = TestSessionLocal()
```

```
try:  
    yield db  
finally:  
    db.close()  
  
app.dependency_overrides[get_db] = override_get_db  
with TestClient(app) as c:  
    yield c  
app.dependency_overrides.clear()
```

 `dependency_overrides` permet de remplacer `get_db` par une version qui utilise la base de test au lieu de la vraie base.

Étape 4 : Supprimer l'ancien code

-  `from src.app import clear_tasks`
-  Les appels à `clear_tasks()` dans les fixtures

Checkpoint

```
cd backend  
uv run pytest -v
```

Tous les tests doivent passer (19+ tests)

Exercice 6 : Configurer Supabase (Base de Données)

Objectif

Créer une base de données PostgreSQL gratuite sur Supabase.

Instructions

Étape 1 : Créer un compte Supabase

1. Allez sur <https://supabase.com>
2. Cliquez "Start your project"

3. Inscrivez-vous avec votre compte **GitHub**

Étape 2 : Créer un projet

1. Cliquez "**New project**"

2. Configurez le projet :

- **Name** : taskflow
- **Database Password** : Choisissez un mot de passe fort (**⚠ notez-le !**)
- **Region** : West EU (Ireland) (le plus proche)

3. Cliquez "**Create new project**"

4. Attendez la création (1-2 minutes)

Après la création, vous arriverez sur la page d'accueil du projet :



Étape 3 : Récupérer la **DATABASE_URL**

1. Cliquez sur le bouton "**Connect**" en haut (visible dans le header)



2. Dans l'onglet **Connection String** :

- Vérifiez que **Type** = URI
- Changez **Method** : Session pooler (au lieu de "Direct connection")

⚠ Important : Render utilise IPv4, mais la connexion directe Supabase nécessite IPv6. Le **Session Pooler** résout ce problème.

3. Copiez l'URL affichée

4. **Important** : Remplacez [YOUR-PASSWORD] par le mot de passe que vous avez choisi à l'étape 2

L'URL ressemble à :

```
postgresql://postgres.[PROJECT-ID]:[YOUR-PASSWORD]@aws-0-eu-west-1.pooler.supabase.com
```

💡 Notez le host pooler.supabase.com (au lieu de db.xxx.supabase.co).

⚠ Gardez cette URL ! Vous en aurez besoin pour l'exercice 7.

Pourquoi Supabase ?

Avantage	Description
Gratuit	500 MB de stockage
PostgreSQL	Base de données professionnelle
Interface web	Explorer les données facilement
Pas de carte bancaire	Contrairement à d'autres services

Exercice 7 : Déployer sur Render

Objectif

Déployer le backend et le frontend sur Render. Deux méthodes sont proposées.

Prérequis

1. Créez un compte Render : <https://render.com>
2. Inscrivez-vous avec **GitHub** et autorisez l'accès à vos repositories
3. Poussez vos changements sur GitHub :

```
git add .
git commit -m "feat: migrate to PostgreSQL with SQLAlchemy"
git push origin main
```

Option A : Déploiement Manuel (via Dashboard)

Cette méthode vous permet de comprendre chaque étape du déploiement.

Étape 1 : Déployer le Backend

1. Sur Render Dashboard, cliquez "New +" → "Web Service"
2. Connectez votre repository GitHub
3. Configurez le service :

Paramètre	Valeur
Name	taskflow-backend
Region	Frankfurt (EU Central)
Branch	main
Root Directory	backend
Runtime	Python 3
Build Command	pip install uv && uv sync
Start Command	uv run uvicorn src.app:app --host 0.0.0.0 --port \$PORT
Instance Type	Free

 `$PORT` est défini automatiquement par Render - ne pas le remplacer !

4. Dans la section **Environment Variables**, ajoutez :

Name	Value
DATABASE_URL	L'URL Supabase de l'exercice 6

 `CORS_ORIGINS` sera ajouté après le déploiement du frontend (étape 3)

5. Cliquez "Create Web Service"

Étape 2 : Déployer le Frontend

1. Cliquez "New +" → "Static Site"

2. Connectez le même repository

3. Configurez :

Paramètre	Valeur
Name	taskflow-frontend
Branch	main
Root Directory	frontend
Build Command	npm ci && npm run build
Publish Directory	dist

4. Dans la section **Environment Variables**, ajoutez :

Name	Value
VITE_API_URL	https://edl-starter.onrender.com (URL du backend)

5. Cliquez "Create Static Site"

Étape 3 : Configurer CORS (après déploiement)

Une fois les deux services déployés, retournez dans le **Backend** :

1. Dashboard → **taskflow-backend** → **Environment**

2. Ajoutez la variable :

Name	Value
CORS_ORIGINS	https://taskflow-frontend-XXXX.onrender.com

3. Cliquez "**Save Changes**" → Le backend redémarre automatiquement

Option B : Déploiement avec YAML (Infrastructure as Code)

Cette méthode automatisé le déploiement via un fichier de configuration.

Étape 1 : Créer `render.yaml` à la racine du projet

```
services:
  # Backend FastAPI
  - type: web
    name: taskflow-backend
    runtime: python
    region: frankfurt
    plan: free
    buildCommand: pip install uv && uv sync
    startCommand: uv run unicorn src.app:app --host 0.0.0.0 --port $PORT
    rootDir: backend
    envVars:
      - key: DATABASE_URL
        sync: false # Configuré manuellement
      - key: CORS_ORIGINS
        sync: false
    healthCheckPath: /health

  # Frontend React
  - type: web
    name: taskflow-frontend
    runtime: static
    region: frankfurt
    plan: free
    buildCommand: npm ci && npm run build
    staticPublishPath: ./dist
    rootDir: frontend
    envVars:
```

```
- key: VITE_API_URL  
sync: false
```

Étape 2 : Déployer avec Blueprint

1. Poussez le fichier `render.yaml` sur GitHub
2. Sur Render Dashboard : "**New +**" → "**Blueprint**"
3. Sélectionnez votre repository
4. Render détecte automatiquement `render.yaml`
5. Cliquez "**Apply**"

Étape 3 : Configurer les Variables d'Environnement

Même configuration que l'Option A (voir ci-dessus).

Comparaison des deux méthodes

Aspect	Option A (Manuel)	Option B (YAML)
Apprentissage	 Meilleur pour comprendre	 Plus rapide
Reproductibilité	 Manuel à chaque fois	 Versionné dans Git
Production	 Pas recommandé	 Best practice

Résultat attendu

Après déploiement, notez vos URLs :

```
Backend: https://edl-starter.onrender.com  
Frontend: https://taskflow-frontend-YYYY.onrender.com
```

Pendant l'attente (3-5 minutes)

Observez les logs de build en temps réel pour chaque service.

👉 Exercice 8 : Vérifier le Déploiement

Objectif

Tester que tout fonctionne en production.

Instructions

1. Testez l'API Backend :

```
# Health check
curl https://edl-starter.onrender.com/health
```

Vous devriez voir :

```
{
  "status": "healthy",
  "database": "connected",
  "tasks_count": 0,
  "environment": "production"
}
```

2. Créez une tâche :

```
curl -X POST https://edl-starter.onrender.com/tasks \
-H "Content-Type: application/json" \
-d '{
  "title": "Test production",
  "status": "todo",
  "priority": "high"
}'
```

3. Listez les tâches :

```
curl https://edl-starter.onrender.com/tasks
```

4. Testez le Frontend :

- Ouvrez <https://taskflow-frontend-YYYY.onrender.com>
- Créez plusieurs tâches
- Modifiez une tâche
- Supprimez une tâche

Résultat attendu

Tout fonctionne parfaitement! 🎉

Récapitulatif

Félicitations ! Vous avez maintenant :

- ✓ **Exercice 1** : Installé SQLAlchemy et psycopg2
- ✓ **Exercice 2** : Configuré la connexion à la base de données
- ✓ **Exercice 3** : Créé le modèle ORM TaskModel
- ✓ **Exercice 4** : Migré app.py vers PostgreSQL
- ✓ **Exercice 5** : Adapté les tests avec une DB temporaire
- ✓ **Exercice 6** : Configuré Supabase (base de données PostgreSQL)
- ✓ **Exercice 7** : Déployé sur Render
- ✓ **Exercice 8** : Vérifié le déploiement en production

Temps total estimé : 2h30

Ce que Vous Avez Appris

- ✓ **SQLAlchemy ORM** - Modèles Python ↔ Tables SQL
- ✓ **PostgreSQL** - Base de données relationnelle professionnelle
- ✓ **Déploiement Cloud** - Backend et Frontend sur Render
- ✓ **Base de données managée** - Supabase pour PostgreSQL
- ✓ **Variables d'environnement** - Configuration production vs développement
- ✓ **CORS** - Communication cross-origin frontend/backend