



TP 3 : Base de Données et Déploiement en Production

Durée estimée : 3h00 **Prérequis :** TP 1 & 2 terminés + compte GitHub



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) :

```
Frontend (localhost:5173) ← → Backend (localhost:8000)
                        ↓
                    Liste Python (RAM)
                    ❌ Données perdues au redémarrage
```

Après (Production avec PostgreSQL) :

```
Frontend (Render)                Backend (Render)                Database (PostgreSQL)
taskflow-frontend.onrender.com → taskflow-backend.onrender.com → PostgreSQL
      HTTPS                                HTTPS + CORS                                256
                                            Données sauvegardées
```

Pourquoi PostgreSQL ?

Problème actuel : Les données sont stockées dans une liste Python en mémoire

- ❌ Données perdues à chaque redémarrage
- ❌ Impossible de scaler (plusieurs instances)
- ❌ Pas de requêtes complexes

Avec PostgreSQL :

- ✅ Données persistantes
 - ✅ Requêtes SQL puissantes
 - ✅ Base de données professionnelle
 - ✅ Gratuit sur Render
-

Exercice 1 : Installer les Dépendances PostgreSQL

Objectif

Ajouter SQLAlchemy et le driver PostgreSQL au backend.

Instructions

1. Ajoutez les packages nécessaires :

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

2. Vérifiez l'installation :

```
uv run python -c "import sqlalchemy; print(f'SQLAlchemy {sqlalchemy.__version__}')
```

✅ Résultat attendu

Vous devriez voir la version de SQLAlchemy s'afficher (ex: `SQLAlchemy 2.0.25`)

💡 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

1. Créez le fichier `backend/src/database.py`
2. Ajoutez la configuration suivante :
 - Importer les modules nécessaires : `sqlalchemy` , `sessionmaker` , `declarative_base`
 - Lire `DATABASE_URL` depuis les variables d'environnement (défaut: `sqlite:///./taskflow.db`)
 - Créer un moteur SQLAlchemy avec `create_engine()`
 - Pour SQLite : ajouter `connect_args={"check_same_thread": False}`
 - Pour PostgreSQL : configurer la pool de connexions avec `pool_size=5` , `max_overflow=10` , `pool_pre_ping=True`
 - Créer une factory de sessions avec `sessionmaker()`
 - Créer une `Base` avec `declarative_base()` pour les modèles ORM
3. Ajoutez deux fonctions :
 - `get_db()` : Générateur qui fournit une session de base de données (pour FastAPI Depends)
 - `init_db()` : Initialise la base de données en créant toutes les tables

💡 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

1. Créez le fichier `backend/src/models.py`
2. Déplacez les enums depuis `app.py` :
 - `TaskStatus` : todo, in_progress, done
 - `TaskPriority` : low, medium, high
3. Créez la classe `TaskModel` qui hérite de `Base` :
 - Définir `__tablename__ = "tasks"`
 - Ajouter les colonnes avec `Column()` :
 - `id` : String, primary key, index
 - `title` : String(200), non nullable
 - `description` : String(1000), nullable
 - `status` : Enum (TaskStatus), défaut TODO
 - `priority` : Enum (TaskPriority), défaut MEDIUM
 - `assignee` : String(100), nullable
 - `due_date` : DateTime, nullable
 - `created_at` : DateTime, auto (server_default=func.now())
 - `updated_at` : DateTime, auto (server_default=func.now(), onupdate=func.now())

💡 Avantages de l'ORM

- Pas besoin d'écrire du SQL directement
- Type-safety avec Python
- Migrations de schéma facilitées
- Timestamps automatiques

🔪 Exercice 4 : Migrer l'Application vers PostgreSQL

Objectif

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

Instructions

Partie 1 : Imports et nettoyage

1. Ajoutez les imports nécessaires :

- `from sqlalchemy.orm import Session`
- `from sqlalchemy import text`
- `from .database import get_db, init_db`
- `from .models import TaskModel, TaskStatus, TaskPriority`

2. Supprimez l'ancien code :

- ❌ Supprimez les définitions de `TaskStatus` et `TaskPriority` (maintenant dans `models.py`)
- ❌ Supprimez `tasks_storage: List[Task] = []`
- ❌ Supprimez les fonctions `clear_tasks()` et `get_tasks_storage()`

Partie 2 : Modifier le lifespan

3. Dans la fonction `lifespan()`, appelez `init_db()` au démarrage

Partie 3 : Modifier les endpoints (utilisez `db: Session = Depends(get_db)`)

4. GET `/tasks` :

- Récupérer toutes les tâches avec `db.query(TaskModel).all()`

5. **POST /tasks :**

- Créer un `TaskModel` avec les données reçues
- Ajouter à la session avec `db.add()`
- Sauvegarder avec `db.commit()`
- Rafraîchir avec `db.refresh()`

6. **GET /tasks/{task_id} :**

- Chercher avec `db.query(TaskModel).filter(TaskModel.id == task_id).first()`
- Lever `HTTPException(404)` si non trouvé

7. **PUT /tasks/{task_id} :**

- Chercher la tâche
- Mettre à jour les champs avec `setattr()`
- Commit et refresh

8. **DELETE /tasks/{task_id} :**

- Chercher la tâche
- Supprimer avec `db.delete()`
- Commit

9. **Améliorer /health :**

- Tester la connexion DB avec `db.execute(text("SELECT 1"))`
- Compter les tâches avec `db.query(TaskModel).count()`
- Retourner le statut de la DB et le nombre de tâches

Checkpoint

Testez localement :

```
cd backend
uv run uvicorn 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.

Instructions

1. Dans `backend/tests/conftest.py`, modifiez la fixture :

- Créer une base de données de test temporaire avec `tempfile.mktemp()`
- Créer un moteur de test avec `create_engine(TEST_DATABASE_URL)`
- Créer une factory de sessions de test avec `sessionmaker()`
- Fixture `setup_test_database` (`scope="session"`) : créer toutes les tables
- Fixture `clear_test_data` (`autouse=True`) : nettoyer entre chaque test
- Fixture `client` : override `get_db` pour utiliser la DB de test

2. Lancez les tests :

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

Résultat attendu

Tous les tests doivent passer (19+ tests)

Exercice 6 : Créer un Compte Render

Objectif

Préparer le déploiement sur Render.

Instructions





1. Créer un compte Render :

- Allez sur <https://render.com>
- Cliquez "**Get Started**"
- Inscrivez-vous avec votre compte **GitHub**
- Autorisez Render à accéder à vos repositories

2. Explorez le Dashboard :

- Familiarisez-vous avec l'interface
- Notez le bouton "**New +**" pour créer des services

Render vs Heroku

-  Gratuit pour PostgreSQL + 2 services
-  Déploiement automatique depuis GitHub
-  Infrastructure as Code avec `render.yaml`
-  HTTPS automatique

Exercice 7 : Comprendre render.yaml

Objectif

Comprendre l'Infrastructure as Code pour Render.

Instructions

1. Ouvrez `render.yaml` à la racine du projet
2. Analysez la structure :

Section `databases` :

- Définit une base PostgreSQL gratuite
- Région : Frankfurt (proche de vous)
- Nom : `taskflow-db`

Section `services` (Backend) :

- Type : `web` (service HTTP)
- Runtime : `python`
- Build command : installe UV et les dépendances
- Start command : lance uvicorn
- Variables d'environnement :
 - `DATABASE_URL` : injectée automatiquement depuis la DB
 - `CORS_ORIGINS` : à configurer manuellement
- Health check : `/health`

Section `services` (Frontend) :

- Type : `web`
- Runtime : `static` (site statique)
- Build command : `npm ci && npm run build`
- Publish path : `frontend/dist`
- Variable : `VITE_API_URL` à configurer

💡 Avantages de `render.yaml`

- ☒ Toute l'infrastructure est versionnée dans Git
- ☒ Déploiement reproductible
- ☒ Création automatique de tous les services
- ☒ Injection automatique de `DATABASE_URL`

📝 Exercice 8 : Déployer avec Blueprint

Objectif

Déployer toute l'application sur Render en un clic.

Instructions

1. Assurez-vous que vos changements sont poussés sur GitHub :

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

2. Sur Render Dashboard :

- Cliquez **"New +"** → **"Blueprint"**
- Sélectionnez votre repository
- Render détecte automatiquement `render.yaml`
- Cliquez **"Apply"**

3. Attendez le déploiement (5-7 minutes) :

- 3 services vont être créés :
 - `taskflow-db` (PostgreSQL)
 - `taskflow-backend` (FastAPI)
 - `taskflow-frontend` (React)

4. Notez les URLs générées :

```
Backend: https://taskflow-backend-XXXX.onrender.com  
Frontend: https://taskflow-frontend-YYYY.onrender.com
```



Pendant l'attente

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



Exercice 9 : Configurer CORS et API URL

Objectif

Connecter le frontend au backend en production.

Instructions

1. Configurer le Backend :

- Dashboard → **taskflow-backend** → **Environment**
- Ajoutez : `CORS_ORIGINS = https://taskflow-frontend-YYYY.onrender.com`
- (Remplacez YYYY par votre ID frontend)
- Cliquez "**Save Changes**"
- Attendez le redéploiement automatique (2-3 min)

2. Configurer le Frontend :

- Dashboard → **taskflow-frontend** → **Environment**
- Ajoutez : `VITE_API_URL = https://taskflow-backend-XXXX.onrender.com`
- (Remplacez XXXX par votre ID backend)
- Cliquez "**Save Changes**"
- Attendez le redéploiement automatique (2-3 min)

✅ Résultat attendu

Les deux services redémarrent automatiquement avec les nouvelles configurations.



Exercice 10 : Vérifier le Déploiement

Objectif

Tester que tout fonctionne en production.

Instructions

1. Testez l'API Backend :

```
# Health check
curl https://taskflow-backend-XXXX.onrender.com/health
```

Vous devriez voir :

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

2. Créez une tâche :

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

3. Listez les tâches :

```
curl https://taskflow-backend-XXXX.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! 🎉



Exercice 11 : Vérifier la Persistance

Objectif

Prouver que PostgreSQL persiste les données.

Instructions

1. Créez 3-4 tâches depuis le frontend
2. Forcez un redéploiement :
 - Dashboard → **taskflow-backend**
 - Cliquez "**Manual Deploy**" → "**Deploy latest commit**"
 - Attendez le redéploiement (2-3 minutes)
3. Rafraîchissez votre frontend

✅ Résultat attendu

Les tâches sont toujours là! PostgreSQL conserve les données entre les redémarrages.



Exercice 12 : Explorer la Base de Données

Objectif

Voir directement les données dans PostgreSQL.

Instructions

1. Ouvrez le shell PostgreSQL :
 - Dashboard → **taskflow-db** → **Shell**
2. Exécutez ces commandes SQL :

```
-- Voir toutes les tables
\dt

-- Voir la structure de la table tasks
\d tasks
```

```
-- Voir toutes les tâches
SELECT id, title, status, priority, created_at FROM tasks;

-- Compter les tâches par statut
SELECT status, COUNT(*) FROM tasks GROUP BY status;
```

✅ Résultat attendu

Vous voyez vos données stockées dans PostgreSQL!

📝 Exercice 13 : Ajouter une Nouvelle Fonctionnalité

Objectif

Démontrer le déploiement automatique en ajoutant un endpoint simple.

Instructions

1. Dans `backend/src/app.py`, ajoutez un endpoint de comptage :

```
@app.get("/tasks/count")
async def count_tasks(db: Session = Depends(get_db)):
    """Count total number of tasks."""
    logger.info("Counting tasks")
    total = db.query(TaskModel).count()
    return {"total": total}
```

2. Testez localement :

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

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

3. Ajoutez un test dans `backend/tests/test_count.py` :

```
def test_count_tasks(client):
    """Test counting tasks."""
    # Au début, 0 tâches
    response = client.get("/tasks/count")
    assert response.status_code == 200
    assert response.json()["total"] == 0

    # Créer 3 tâches
    for i in range(3):
        client.post("/tasks", json={
            "title": f"Task {i+1}",
            "status": "todo",
            "priority": "medium"
        })

    # Maintenant, 3 tâches
    response = client.get("/tasks/count")
    assert response.status_code == 200
    assert response.json()["total"] == 3
```

4. Vérifiez que les tests passent :

```
uv run pytest -v
```

✅ Résultat attendu

Tous les tests passent (20+ tests maintenant)

Exercice 14 : Déployer la Nouvelle Fonctionnalité

Objectif

Observer le cycle complet CI/CD automatique.

Instructions



1. Committez et poussez :

```
git add .
git commit -m "feat: add task count endpoint"

- Add GET /tasks/count endpoint
- Add test for count endpoint
- Returns total number of tasks in database"

git push origin main
```

2. Observez GitHub Actions (1-2 min) :

- GitHub → **Actions**
- Workflow démarre automatiquement
- Backend tests 
- Frontend tests 

3. Observez Render Auto-Deploy (3-5 min) :

- Render Dashboard → **taskflow-backend**
- Status : "Deploying..."
- Observez les logs de build en temps réel

4. Testez en production :

```
curl https://taskflow-backend-XXXX.onrender.com/tasks/count
```

5. Vérifiez dans Swagger UI :

- Ouvrez : `https://taskflow-backend-XXXX.onrender.com/docs`
- Le nouveau endpoint `GET /tasks/count` apparaît
- Testez-le avec "Try it out"

Résultat attendu

La nouvelle fonctionnalité est déployée automatiquement! 🚀

Workflow Complet CI/CD

Ce qui s'est passé automatiquement :

```
1. git push origin main
  ↓
2. GitHub Actions démarre
   ├── Backend: uv run pytest ✅
   ├── Frontend: npm test ✅
   └── Les tests passent
  ↓
3. Render détecte le push
  ↓
4. Render clone le nouveau code
  ↓
5. Render rebuild le backend
   ├── pip install uv
   ├── uv sync (install dependencies)
   └── uv run uvicorn (start server)
  ↓
6. Health check: /health ✅
  ↓
7. 🎉 Nouvelle version LIVE !

Temps total: ~5-7 minutes
```

Zero configuration nécessaire ! Tout est automatique grâce à :

- `.github/workflows/backend.yml` (tests)
- `render.yaml` (déploiement)

Récapitulatif

Félicitations ! Vous avez maintenant :

✅ **Exercice 1** : Installé SQLAlchemy et psycpg2 ✅ **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** : Créé un compte Render ✅ **Exercice 7** : Compris render.yaml (IaC) ✅ **Exercice 8** : Déployé avec Blueprint en un clic ✅ **Exercice 9** : Configuré CORS et API URL ✅ **Exercice 10** : Vérifié le déploiement en production ✅ **Exercice 11** : Prouvé la persistance des données ✅ **Exercice 12** : Exploré PostgreSQL avec SQL ✅ **Exercice 13** : Ajouté un endpoint de comptage ✅ **Exercice 14** : Déployé automatiquement avec CD

Temps total estimé : 3 heures



Ce que Vous Avez Appris

✅ **SQLAlchemy ORM** - Modèles Python ↔ Tables SQL ✅ **PostgreSQL** - Base de données relationnelle professionnelle ✅ **Infrastructure as Code** - render.yaml pour définir l'infra ✅ **Continuous Deployment** - Push → Tests → Deploy automatique ✅ **API REST** - Nouveaux endpoints avec tests ✅ **Production monitoring** - Logs, health checks, database status ✅ **Data persistence** - Les données survivent aux redémarrages



Pour Aller Plus Loin

Fonctionnalités Simples (30 min chacune)

1. **Endpoint de recherche** : `GET /tasks/search?q=query`
2. **Endpoint de filtrage** : `GET /tasks/filter/{status}`
3. **Endpoint de statistiques** : `GET /tasks/stats` (compte par statut/priorité)
4. **Badge de comptage** : Afficher le count dans le frontend

Fonctionnalités Avancées (1-2h chacune)

1. **Pagination** : Ajouter `skip` et `limit` aux endpoints
2. **Authentification** : JWT tokens avec FastAPI Security
3. **Filtrage UI** : Boutons pour filtrer par statut dans le frontend

4. **Dashboard de stats** : Graphiques avec Chart.js

DevOps Avancé

1. **Monitoring** : Intégrer Sentry pour error tracking
 2. **Staging Environment** : Environnement de pré-production
 3. **Database Migrations** : Alembic pour migrations SQL
 4. **Custom Domain** : Utiliser votre propre nom de domaine
-

Checklist de Fin d'Atelier

Migration PostgreSQL :

- ☐ SQLAlchemy et psycopg2 installés
- ☐ `database.py` créé avec configuration
- ☐ `models.py` créé avec TaskModel
- ☐ `app.py` migré pour utiliser la DB
- ☐ Tests adaptés avec base de test
- ☐ Tests locaux passent

Déploiement :

- ☐ Compte Render créé
- ☐ `render.yaml` compris
- ☐ Blueprint déployé avec succès
- ☐ Backend accessible via HTTPS
- ☐ Frontend accessible via HTTPS
- ☐ CORS configuré
- ☐ PostgreSQL connectée

Continuous Deployment :

- ☐ Push déclenche GitHub Actions
- ☐ Tests passent automatiquement
- ☐ Render auto-deploy fonctionne

- ☐ Nouvelles fonctionnalités visibles en prod
- ☐ Données persistent après redéploiement

Si tout est coché : Bravo, vous maîtrisez le cycle complet ! 🎉🚀

Ressources

Documentation Technique :

- [SQLAlchemy Docs](#)
 - [FastAPI Database Guide](#)
 - [Render Blueprint Spec](#)
 - [PostgreSQL Docs](#)
-

Version 5.0 - TP 3 : Base de Données et Déploiement en Production (3h)