



Atelier 3 : Base de Données et Déploiement en Production

Durée estimée : 3h00 **Prérequis :** Ateliers 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** (filtrage, recherche, statistiques)
 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
```

Phase 1 : Migration vers PostgreSQL (60 min)

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
-

Étape 1.1 : Installer les Dépendances

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

Ce que font ces packages :

- `sqlalchemy` : ORM (Object-Relational Mapping) pour Python
- `psycopg2-binary` : Driver PostgreSQL

Vérifiez l'installation :

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

Étape 1.2 : Créer `backend/src/database.py`

Ce fichier configure la connexion à la base de données.

```

import os
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, Session, declarative_base
from typing import Generator
import logging

logger = logging.getLogger("taskflow")

# Lire l'URL de la base de données depuis l'environnement
# Par défaut : SQLite pour le développement local
DATABASE_URL = os.getenv("DATABASE_URL", "sqlite:///./taskflow.db")

# Fix pour Render : postgres:// → postgresql://
if DATABASE_URL.startswith("postgres://"):
    DATABASE_URL = DATABASE_URL.replace("postgres://", "postgresql://", 1)

# Configuration du moteur SQLAlchemy
engine_kwargs = {}
if DATABASE_URL.startswith("sqlite"):
    # SQLite : désactiver le check_same_thread
    engine_kwargs["connect_args"] = {"check_same_thread": False}
else:
    # PostgreSQL : configuration de la pool de connexions
    engine_kwargs.update({
        "pool_size": 5,          # 5 connexions dans la pool
        "max_overflow": 10,     # 10 connexions supplémentaires max
        "pool_pre_ping": True,  # Vérifier que la connexion est vivante
    })

# Créer le moteur SQLAlchemy
engine = create_engine(DATABASE_URL, **engine_kwargs)

# Créer la factory de sessions
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Base pour les modèles ORM
Base = declarative_base()

def get_db() -> Generator[Session, None, None]:
    """
    Dependency function pour obtenir une session de base de données.

```

```

Utilisée avec FastAPI Depends().
"""
db = SessionLocal()
try:
    yield db
finally:
    db.close()

def init_db() -> None:
    """Initialise la base de données en créant toutes les tables."""
    logger.info("🔧 Initializing database tables...")
    Base.metadata.create_all(bind=engine)
    logger.info("✅ Database tables created successfully!")

```

Ce que fait ce fichier :

- Supporte SQLite (local) et PostgreSQL (production)
- Configure une pool de connexions pour PostgreSQL
- Fournit `get_db()` pour FastAPI Depends
- Fournit `init_db()` pour créer les tables

Étape 1.3 : Créer `backend/src/models.py`

Ce fichier définit le schéma de la table `tasks`.

```

from sqlalchemy import Column, String, DateTime, Enum as SQLAlchemyEnum
from sqlalchemy.sql import func
from .database import Base
from enum import Enum

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"

    # Colonnes
    id = Column(String, primary_key=True, index=True)
    title = Column(String(200), nullable=False)
    description = Column(String(1000), nullable=True)
    status = Column(
        SQLAlchemyEnum(TaskStatus, values_callable=lambda x: [e.value for e in x],
        nullable=False,
        default=TaskStatus.TODO.value
    )
    priority = Column(
        SQLAlchemyEnum(TaskPriority, values_callable=lambda x: [e.value for e in x],
        nullable=False,
        default=TaskPriority.MEDIUM.value
    )
    assignee = Column(String(100), nullable=True)
    due_date = Column(DateTime, nullable=True)
    created_at = Column(DateTime, nullable=False, server_default=func.now())
    updated_at = Column(DateTime, nullable=False, server_default=func.now())

```

Ce que fait ce fichier :

- Définit la structure de la table `tasks`
- Chaque `Column` = une colonne SQL
- Les `Enum` définissent les valeurs valides
- `created_at` et `updated_at` automatiques

Étape 1.4 : Migrer `backend/src/app.py`

Modifications à apporter :

1. Importer les nouveaux modules

Ajoutez en haut du fichier :

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

2. Supprimer l'ancien code

✗ SUPPRIMEZ ces lignes :

```
# SUPPRIMEZ les définitions d'Enum (maintenant dans models.py)
class TaskStatus(str, Enum):
    ...

class TaskPriority(str, Enum):
    ...

# SUPPRIMEZ le stockage en mémoire
tasks_storage: List[Task] = []
```

3. Modifier la fonction lifespan

Remplacez :

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    """Lifespan context manager for startup/shutdown events."""
    logger.info("🚀 Starting TaskFlow backend...")
    yield
    logger.info("👋 Shutting down TaskFlow backend...")
```

Par :

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    """Lifespan context manager for startup/shutdown events."""
    logger.info("🚀 Starting TaskFlow backend...")
```

```
# Initialiser la base de données (créer les tables)
init_db()

yield

logger.info("👋 Shutting down TaskFlow backend...")
```

4. Modifier tous les endpoints

GET /tasks :

```
@app.get("/tasks", response_model=list[Task])
async def get_tasks(db: Session = Depends(get_db)):
    """Get all tasks."""
    logger.info("Fetching all tasks")
    db_tasks = db.query(TaskModel).all()
    return db_tasks
```

POST /tasks :

```
@app.post("/tasks", response_model=Task, status_code=201)
async def create_task(task_data: TaskCreate, db: Session = Depends(get_db)):
    logger.info(f"Creating task: {task_data.title}")

    db_task = TaskModel(
        id=str(uuid4()),
        **task_data.model_dump()
    )

    db.add(db_task)
    db.commit()
    db.refresh(db_task)

    logger.info(f"Task created successfully: {db_task.id}")
    return db_task
```

GET /tasks/{task_id} :

```

@app.get("/tasks/{task_id}", response_model=Task)
async def get_task(task_id: str, db: Session = Depends(get_db)):
    logger.info(f"Fetching task: {task_id}")

    db_task = db.query(TaskModel).filter(TaskModel.id == task_id).first()
    if not db_task:
        raise HTTPException(status_code=404, detail="Task not found")

    return db_task

```

PUT /tasks/{task_id} :

```

@app.put("/tasks/{task_id}", response_model=Task)
async def update_task(task_id: str, task_update: TaskUpdate, db: Session = Depends(get_db)):
    logger.info(f"Updating task: {task_id}")

    db_task = db.query(TaskModel).filter(TaskModel.id == task_id).first()
    if not db_task:
        raise HTTPException(status_code=404, detail="Task not found")

    update_data = task_update.model_dump(exclude_unset=True)
    for field, value in update_data.items():
        setattr(db_task, field, value)

    db.commit()
    db.refresh(db_task)

    logger.info(f"Task updated: {task_id}")
    return db_task

```

DELETE /tasks/{task_id} :

```

@app.delete("/tasks/{task_id}", status_code=204)
async def delete_task(task_id: str, db: Session = Depends(get_db)):
    logger.info(f"Deleting task: {task_id}")

    db_task = db.query(TaskModel).filter(TaskModel.id == task_id).first()
    if not db_task:
        raise HTTPException(status_code=404, detail="Task not found")

```



```
db.delete(db_task)
db.commit()
logger.info(f"Task deleted: {task_id}")
```

5. Améliorer le Health Check

```
@app.get("/health")
async def health_check(db: Session = Depends(get_db)):
    """Health check endpoint with database connectivity test."""
    try:
        # Tester la connexion à la base de données
        db.execute(text("SELECT 1"))
        db_status = "connected"

        # Compter les tâches
        tasks_count = db.query(TaskModel).count()
    except Exception as e:
        logger.error(f"Database health check failed: {e}")
        db_status = "disconnected"
        tasks_count = 0

    return {
        "status": "healthy",
        "database": db_status,
        "tasks_count": tasks_count,
        "timestamp": datetime.utcnow().isoformat(),
        "environment": os.getenv("ENVIRONMENT", "development"),
        "version": "1.0.0"
    }
```

Étape 1.5 : Adapter les Tests

Modifiez `backend/tests/conftest.py` :

```
import pytest
import tempfile
import os as os_module
from fastapi.testclient import TestClient
```

```

from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

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

# Créer une base de données de test 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},
)

TestSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=test_engine)

@pytest.fixture(scope="session", autouse=True)
def setup_test_database():
    """Créer les tables de test une seule fois pour toute la session."""
    Base.metadata.create_all(bind=test_engine)
    yield
    # Nettoyer après tous les tests
    Base.metadata.drop_all(bind=test_engine)
    if os.path.exists(TEST_DB_FILE):
        os.remove(TEST_DB_FILE)

@pytest.fixture(autouse=True)
def clear_test_data():
    """Nettoyer les données entre chaque test."""
    session = TestSessionLocal()
    try:
        session.query(TaskModel).delete()
        session.commit()
    finally:
        session.close()
    yield

@pytest.fixture
def client():
    """Fournir un client de test avec une base de données de test."""

```

```
def override_get_db():
    session = TestSessionLocal()
    try:
        yield session
    finally:
        session.close()

app.dependency_overrides[get_db] = override_get_db

try:
    with TestClient(app) as test_client:
        yield test_client
finally:
    app.dependency_overrides.clear()
```

Étape 1.6 : Tester Localement

```
cd backend

# Lancer les tests
uv run pytest -v

# Lancer le serveur
uv run uvicorn src.app:app --reload

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

# Créer une tâche
curl -X POST http://localhost:8000/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Test PostgreSQL",
    "status": "todo",
    "priority": "high"
  }'
```

✓ **Checkpoint** : Vous devriez voir un fichier `taskflow.db` créé dans `backend/`

Phase 2 : Déploiement avec render.yaml (45 min)

Étape 2.1 : Créer un Compte Render

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

Étape 2.2 : Comprendre render.yaml

Le fichier `render.yaml` à la racine définit toute l'infrastructure :

```
databases:
  # PostgreSQL Database
  - name: taskflow-db
    databaseName: taskflow
    region: frankfurt
    plan: free
    user: taskflow

services:
  # Backend Service - FastAPI
  - type: web
    name: taskflow-backend
    runtime: python
    region: frankfurt
    plan: free
    branch: main
    buildCommand: "cd backend && pip install uv && uv sync"
    startCommand: "cd backend && uv run uvicorn src.app:app --host 0.0.0.0"
    envVars:
      - key: PYTHON_VERSION
        value: "3.11"
```

```
- key: ENVIRONMENT
  value: "production"
- key: CORS_ORIGINS
  sync: false # À configurer manuellement
- key: DATABASE_URL
  fromDatabase:
    name: taskflow-db
    property: connectionString # ✅ Connexion automatique !
healthCheckPath: /health

# Frontend Service - React + Vite
- type: web
  name: taskflow-frontend
  runtime: static
  region: frankfurt
  plan: free
  branch: main
  buildCommand: "cd frontend && npm ci && npm run build"
  staticPublishPath: frontend/dist
  envVars:
    - key: VITE_API_URL
      sync: false # À configurer manuellement
```

Ce que Render fait automatiquement :

- ✅ Crée la base PostgreSQL
- ✅ Injecte `DATABASE_URL` dans le backend
- ✅ Build et déploie backend + frontend
- ✅ Configure HTTPS partout
- ✅ Active health checks

Étape 2.3 : Déployer avec Blueprint

1. Dashboard Render : <https://dashboard.render.com>
2. Cliquez "New +" → "Blueprint"
3. Sélectionnez votre repository "edl-tp-1"
4. Render détecte `render.yaml`

5. Cliquez **"Apply"**

 **Attendez 5-7 minutes** pour le déploiement complet.

Notez les URLs :

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

Étape 2.4 : Configurer CORS et URLs

Backend → Environment :

```
CORS_ORIGINS = https://taskflow-frontend-YYYY.onrender.com
```

Frontend → Environment :

```
VITE_API_URL = https://taskflow-backend-XXXX.onrender.com
```

Attendez les redéploiements automatiques (2-3 min chacun).

Étape 2.5 : Vérifier le Déploiement

```
# Health check (doit montrer "database": "connected")
curl https://taskflow-backend-XXXX.onrender.com/health

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

```
# Lister les tâches
curl https://taskflow-backend-XXXX.onrender.com/tasks
```

Testez aussi depuis le frontend : <https://taskflow-frontend-YYYY.onrender.com>

✅ **Checkpoint** : L'application fonctionne en production avec PostgreSQL !

Phase 3 : Vérification et Tests (30 min)

Étape 3.1 : Tester l'API Directement

Une fois déployé, testez tous les endpoints de l'API :

Health Check :

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

Vous devriez voir :

```
{
  "status": "healthy",
  "database": "connected",
  "tasks_count": 0,
  "timestamp": "2025-01-21T10:00:00",
  "environment": "production",
  "version": "1.0.0"
}
```

Créer une tâche :

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

```
"priority": "high"
}'
```

Lister les tâches :

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

Récupérer une tâche par ID :

```
curl https://taskflow-backend-XXXX.onrender.com/tasks/{TASK_ID}
```

Étape 3.2 : Tester le Frontend en Production

1. Ouvrez `https://taskflow-frontend-YYYY.onrender.com`
2. **Créez plusieurs tâches** avec différents statuts et priorités
3. **Modifiez une tâche** (changez son statut)
4. **Supprimez une tâche**

✅ Tout doit fonctionner !

Étape 3.3 : Vérifier la Persistance des Données

Test de persistance PostgreSQL :

1. Créez 3-4 tâches depuis le frontend
2. Allez sur Render Dashboard → **taskflow-backend**
3. Cliquez **Manual Deploy** → **Deploy latest commit**
4. Attendez le redéploiement (2-3 minutes)
5. Rafraîchissez votre frontend

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

Étape 3.4 : Explorer la Base de Données PostgreSQL

Allez voir directement dans la base de données :

1. Dashboard → **taskflow-db** → **Shell**
2. Dans le shell PostgreSQL :

```
-- Voir toutes les tables
\dt

-- Voir les colonnes 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;
```

✅ **Vous voyez vos données !** Elles sont bien stockées dans PostgreSQL.

Étape 3.5 : Vérifier les Logs

Backend logs :

1. Dashboard → **taskflow-backend** → **Logs**
2. Vous devriez voir :

```
🚀 Starting TaskFlow backend...
🗄️ Initializing database tables...
✅ Database tables created successfully!
🌐 CORS enabled for origins: ['https://taskflow-frontend-YYYY.onrender.com']
INFO:      Application startup complete.
```

3. Créez une tâche depuis le frontend
4. Observez les logs en temps réel :

```
INFO: Creating task: Première tâche
INFO: Task created successfully: abc-123-def
```

✅ **Checkpoint** : Le déploiement fonctionne parfaitement !

Phase 4 : Ajouter une Fonctionnalité + Auto-Deploy (45 min)

Maintenant, ajoutons une petite fonctionnalité simple pour démontrer le déploiement automatique.

Étape 4.1 : Ajouter un Endpoint Simple de Comptage

Ajoutez un endpoint simple dans `backend/src/app.py` (avant les autres endpoints) :

```
@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}
```

Testez localement :

```
# Dans un terminal, lancez le serveur
cd backend
uv run uvicorn src.app:app --reload

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

Vous devriez voir :

```
{"total": 0}
```

Étape 4.2 : Ajouter un Test Simple

Créez `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
```

Lancez les tests :

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

✅ Tous les tests doivent passer !

Étape 4.3 : Commit et Push vers GitHub

```
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



🤖 Generated with [Claude Code](https://claude.com/claude-code)

Co-Authored-By: Claude <noreply@anthropic.com>"

```
git push origin main
```


Étape 4.4 : Observer le Pipeline CI/CD

1. GitHub Actions (1-2 min) :

- Allez sur GitHub → **Actions**
- Un nouveau workflow démarre automatiquement
- Observez :
 -  Backend tests (pytest)
 -  Frontend tests (vitest)

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

- Allez sur Render Dashboard
- Cliquez sur **taskflow-backend**
- Vous verrez "Deploying..." en haut
- Observez les logs en temps réel :

```
==> Cloning from https://github.com/...
==> Checking out commit abc123...
==> Running build command 'cd backend && pip install uv && uv sync'...
==> Installing dependencies...
==> Build successful!
==> Starting server...
🚀 Starting TaskFlow backend...
🗄️ Initializing database tables...
 Database tables created successfully!
```

Étape 4.5 : Vérifier la Nouvelle Fonctionnalité en Production

Une fois le déploiement terminé (indicateur vert ) :

Tester le nouveau endpoint :

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



Vérifier dans la documentation Swagger :

1. Ouvrez : `https://taskflow-backend-XXXX.onrender.com/docs`
2. Vous devriez voir le nouveau endpoint `GET /tasks/count`
3. Cliquez sur **"Try it out"** → **"Execute"**
4. Vous verrez le nombre total de tâches

 **La nouvelle fonctionnalité est déployée automatiquement !**

Étape 4.6 : Comprendre le Workflow Complet

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)

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. **Afficher le count dans le frontend** : Badge avec nombre total de tâches

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
-

Ressources

Documentation Technique :

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

Atelier Extensions :

- [Backend README](#) - Documentation complète du backend
 - [DEPLOYMENT.md](#) - Guide de déploiement détaillé
-

Checklist de Fin d'Atelier

Migration PostgreSQL :

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

Déploiement :

- ☐ Compte Render créé et connecté à GitHub
- ☐ `render.yaml` compris et expliqué
- ☐ Blueprint déployé avec succès
- ☐ Backend accessible via HTTPS
- ☐ Frontend accessible via HTTPS
- ☐ CORS configuré correctement
- ☐ PostgreSQL connectée (health check montre "connected")

Nouvelle Fonctionnalité :

- ☐ Endpoint `/tasks/count` implémenté et testé
- ☐ Test unitaire pour le comptage
- ☐ Documentation Swagger affiche le nouvel endpoint

Continuous Deployment :

- ☐ Push vers main déclenche GitHub Actions
- ☐ Tests passent automatiquement
- ☐ Render auto-deploy fonctionne
- ☐ Nouvelles fonctionnalités visibles en production
- ☐ Données persistent après redéploiement

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

Version 4.0 - Atelier 3 : Base de Données et Déploiement en Production (3h)