



# Atelier 3 : Déploiement en Production

---

**Durée estimée :** 3 heures **Prérequis :** Ateliers 1 & 2 terminés (application full-stack avec CI/CD)



## Objectifs de l'Atelier

---

**Objectif principal :** Déployer votre application full-stack en production sur le cloud

À la fin de cet atelier, vous aurez **déployé** :

1. Un **backend FastAPI en production** sur Render
  2. Un **frontend React en production** sur Render
  3. Une **configuration CORS** pour connecter frontend et backend en production
  4. Des **variables d'environnement** pour gérer les différents environnements
  5. Un **monitoring actif** avec health checks
- 



## Architecture Cible

---

**Avant (Local) :**

```
Frontend (localhost:3000) → Vite Proxy → Backend (localhost:8000)
```

**Après (Production) :**

Frontend (Render)	Backend (Render)
taskflow-frontend-XXX.onrender.com	taskflow-backend-XXX.onrender.com
HTTPS	HTTPS + CORS

---

## Phase 1 : Préparation pour la Production (30 min)

---

### 1.1 - Créer un Compte Render

#### EXERCICE : S'inscrire sur Render

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

**Niveau gratuit** : 750 heures/mois gratuites (suffisant pour ce workshop)

### 1.2 - Préparer le Backend pour la Production

#### EXERCICE : Configurer CORS pour la production

Ouvrez `backend/src/app.py` et vérifiez la configuration CORS :

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

# Configuration CORS
cors_origins = os.getenv("CORS_ORIGINS", "http://localhost:3000").split(' ')

app.add_middleware(
    CORSMiddleware,
    allow_origins=cors_origins, # Origines autorisées
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

#### Pourquoi c'est important ?

- En **développement** : CORS permet `localhost:3000`
- En **production** : CORS doit permettre votre URL Render frontend

**Variables d'environnement** :

- `CORS_ORIGINS` : Liste des origines autorisées (séparées par des virgules)

## 1.3 - Préparer le Frontend pour la Production

### 🎯 EXERCICE : Configurer l'URL du backend

Le frontend doit savoir où trouver le backend en production.

Ouvrez `frontend/src/api/api.ts` :

```
const API_BASE = import.meta.env.VITE_API_URL || '/api';
```

Comment ça marche ?

- **Développement** : `VITE_API_URL` n'est pas défini → utilise `/api` (proxy Vite)
- **Production** : `VITE_API_URL` = URL du backend Render → appels directs

Créez `frontend/.env.example` :

```
# URL du backend en production
# Exemple : VITE_API_URL=https://taskflow-backend-XXXX.onrender.com
VITE_API_URL=
```

## 1.4 - Vérifier le Health Check

### 🎯 EXERCICE : Tester le endpoint de santé

Le backend doit avoir un endpoint `/health` pour le monitoring :

```
@app.get("/health")
async def health_check():
    """Health check endpoint pour Render."""
    return {
        "status": "healthy",
        "timestamp": datetime.utcnow().isoformat(),
        "environment": os.getenv("ENVIRONMENT", "development"),
        "version": "1.0.0"
    }
```

Testez localement :

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

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

Réponse attendue :

```
{
  "status": "healthy",
  "timestamp": "2025-01-21T10:00:00",
  "environment": "development",
  "version": "1.0.0"
}
```

## 1.5 - Configurer le Routing Client-Side

### EXERCICE : Créer le fichier de redirects

React Router a besoin de ce fichier pour fonctionner correctement sur Render.

Créez `frontend/public/_redirects` :

```
/*      /index.html    200
```

Que fait ce fichier ?

- Redirige toutes les routes vers `index.html`
- Permet au routing React de gérer les URLs (au lieu de Render)



## Phase 2 : Déployer le Backend (45 min)

### 2.1 - Créer le Service Backend sur Render

#### EXERCICE : Configurer le backend

1. Connectez-vous à <https://dashboard.render.com>

2. Cliquez sur **"New +"** → **"Web Service"**
3. Connectez votre repository GitHub

### Configuration :

```
Name: taskflow-backend
Branch: main
Region: Frankfurt (ou votre région préférée)
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
```

### Important :

- `$PORT` : Variable fournie par Render (ne pas changer)
- `--host 0.0.0.0` : Écoute sur toutes les interfaces (requis pour Render)

## 2.2 - Configurer les Variables d'Environnement

### EXERCICE : Ajouter les variables d'environnement

Dans la page de configuration Render, section **"Environment"** :

```
ENVIRONMENT=production
CORS_ORIGINS=*
PYTHON_VERSION=3.11
```

### Explications :

- `ENVIRONMENT=production` : Mode production
- `CORS_ORIGINS=*` : Permet toutes les origines (à restreindre en vrai production)
- `PYTHON_VERSION=3.11` : Version Python à utiliser

**Note** : En production réelle, remplacez `*` par l'URL exacte du frontend :

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

## 2.3 - Configurer les Build Filters (Monorepo)

### EXERCICE : Optimiser les déploiements

Pour éviter de rebuildier quand seul le frontend change :

**Included Paths :**

```
backend/**  
.github/workflows/**
```

**Ignored Paths :**

```
frontend/**  
docs/**  
*.md
```

## 2.4 - Configurer le Health Check

### EXERCICE : Activer le monitoring

Dans **"Settings"** → **"Health & Alerts"** :

```
Health Check Path: /health
```

Render vérifiera automatiquement que votre backend répond.

## 2.5 - Déclencher le Premier Déploiement

### EXERCICE : Déployer le backend

1. Cliquez sur **"Create Web Service"**
2. Render va :
  - Cloner votre repository
  - Installer les dépendances ( `uv sync` )

- Démarrer le serveur
- Vérifier le health check

**Observez les logs en temps réel** dans la console Render.

**Temps de déploiement** : 2-5 minutes

## 2.6 - Vérifier le Déploiement

### EXERCICE : Tester le backend en production

Une fois le déploiement terminé, vous aurez une URL :

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

Testez dans votre terminal :

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

# Liste des tâches (vide au début)
curl https://taskflow-backend-XXXX.onrender.com/tasks

# Documentation API
# Ouvrez dans le navigateur :
# https://taskflow-backend-XXXX.onrender.com/docs
```

 **Checkpoint** : Le backend doit répondre à tous ces endpoints.



## Phase 3 : Déployer le Frontend (45 min)

---

### 3.1 - Créer le Site Statique sur Render

#### EXERCICE : Configurer le frontend

1. Sur Render Dashboard, cliquez **"New +"** → **"Static Site"**
2. Sélectionnez le même repository GitHub

## Configuration :

```
Name: taskflow-frontend  
Branch: main  
Root Directory: frontend  
  
Build Command: npm install && npm run build  
Publish Directory: dist  
  
Instance Type: Free
```

## 3.2 - Configurer les Variables d'Environnement

### EXERCICE : Pointer vers le backend

Dans "Environment Variables" :

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

⚠ **IMPORTANT** : Remplacez `XXXX` par l'ID de votre backend Render !

### Comment trouver l'URL du backend ?

- Allez sur votre service backend Render
- Copiez l'URL en haut de la page

## 3.3 - Configurer les Build Filters

### EXERCICE : Optimiser les rebuilds

Included Paths :

```
frontend/**  
.github/workflows/**
```

Ignored Paths :

```
backend/**  
docs/**
```



\*.md

### 3.4 - Déclencher le Déploiement Frontend

#### EXERCICE : Déployer le frontend

1. Cliquez sur **"Create Static Site"**
2. Render va :
  - Installer les dépendances ( `npm install` )
  - Builder le projet ( `npm run build` )
  - Publier les fichiers statiques du dossier `dist/`

**Temps de déploiement** : 3-7 minutes

### 3.5 - Vérifier le Déploiement

#### EXERCICE : Tester l'application complète

Votre frontend sera disponible à :

`https://taskflow-frontend-XXXX.onrender.com`

#### Tests à faire :

1. **Ouvrez l'URL dans votre navigateur**
2. **Ouvrez DevTools (F12) → Onglet Network**
3. **Créez une tâche :**
  - Cliquez sur "Nouvelle Tâche"
  - Remplissez le formulaire
  - Soumettez

#### Dans Network tab :

- Vous devez voir : `POST https://taskflow-backend-XXXX.onrender.com/tasks`
- Statut : `201 Created`

#### 4. **Rafraîchissez la page :**

- La tâche doit toujours être là

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

✅ **Checkpoint** : Votre application full-stack fonctionne en production !

---

## Phase 4 : Configuration Avancée (30 min)

---

### 4.1 - Activer le Déploiement Automatique

#### EXERCICE : Auto-deploy sur GitHub push

Par défaut, Render redéploie automatiquement quand vous pushez sur `main`.

Vérifiez dans **Settings** → **Build & Deploy** :

Auto-Deploy: Yes

Test :

1. Faites un petit changement (ex: titre de l'app)
2. Committez et pushez :

```
git add .  
git commit -m "test: verify auto-deploy"  
git push origin main
```

3. Observez dans Render Dashboard :

- ✅ GitHub Actions exécute les tests
- ✅ Render détecte le push
- ✅ Nouveau déploiement automatique

### 4.2 - Configurer CORS Restreint (Production Réelle)

#### EXERCICE : Sécuriser le backend

Pour une vraie production, ne laissez pas `CORS_ORIGINS=*`.

Dans **Render Backend** → **Environment** :

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

Pour plusieurs domaines :

```
CORS_ORIGINS=https://taskflow-frontend-XXXX.onrender.com,https://www.votr
```

**Redéployez manuellement** : Cliquez sur "Manual Deploy" → "Deploy latest commit"

## 4.3 - Surveiller les Logs

### EXERCICE : Déboguer en production

**Backend logs :**

1. Allez sur votre service backend
2. Cliquez sur l'onglet "**Logs**"
3. Vous verrez toutes les requêtes en temps réel

**Frontend logs :**

1. Les logs de build sont dans l'onglet "Logs"
2. Les erreurs runtime sont dans DevTools du navigateur (F12 → Console)

**Commandes utiles :**

```
# Voir les logs backend en live
# (dans le dashboard Render, onglet Logs)

# Chercher une erreur
# Utilisez Ctrl+F dans les logs
```

## 4.4 - Optimiser les Performances

### EXERCICE : Configuration production

**Backend ( backend/src/app.py ) :**

```
# En production, ajoutez :
import logging
```

```
logging.basicConfig(  
    level=logging.INFO if os.getenv("DEBUG") != "true" else logging.DEBUG  
)
```

### Frontend (déjà optimisé par Vite) :

- Minification automatique
- Tree-shaking
- Code splitting
- Compression gzip

---

## Phase 5 : Migration vers PostgreSQL (Optionnel - 90 min)

---

⚠ **IMPORTANT** : Cette phase est **optionnelle** et permet d'ajouter une vraie base de données PostgreSQL. Les phases 1-4 utilisent le stockage en mémoire (les données sont perdues au redémarrage). Si vous voulez que vos données persistent, suivez cette phase !

### 5.0 - Pourquoi Ajouter une Base de Données ?

#### Avec le stockage en mémoire (Phases 1-4) :

- ❌ Les tâches disparaissent quand vous redémarrez le backend
- ❌ Chaque redéploiement efface toutes les données
- ❌ Impossible de scaler (plusieurs instances)

#### Avec PostgreSQL :

- ✅ Les données persistent entre les redémarrages
- ✅ Déploiements sans perte de données
- ✅ Base de données professionnelle
- ✅ Requêtes complexes et relations

### 5.1 - Vue d'Ensemble de la Migration

Vous allez transformer votre backend pour utiliser PostgreSQL au lieu de la liste Python en mémoire.

### Ce que vous allez faire :

1. Créer une base de données PostgreSQL sur Render (gratuit)
2. Créer 2 nouveaux fichiers : `database.py` et `models.py`
3. **Supprimer** le code de stockage en mémoire de `app.py`
4. **Remplacer** par des appels à la base de données
5. Adapter les tests
6. Déployer avec la base de données

**Durée estimée :** 60-90 minutes

## 5.2 - Créer la Base de Données sur Render

### EXERCICE : Provisionner PostgreSQL

1. Allez sur <https://dashboard.render.com>
2. Cliquez "New +" → "PostgreSQL"
3. Configuration :

```
Name: taskflow-db  
Region: Frankfurt (même région que votre backend)  
PostgreSQL Version: 16  
Instance Type: Free
```

4. Cliquez "Create Database"
5. **Attendez 2-3 minutes** que la base soit provisionnée
6. Une fois prête, cliquez sur votre database → "Info"
7. Copiez "Internal Database URL" (commence par `postgresql://` )

### Exemple d'URL :

```
postgresql://taskflow_db_user:mot_de_passe_tres_long@dpg-xxxxx-a/taskflow
```

⚠ **Gardez cette URL** - vous en aurez besoin plus tard !

## 5.3 - Ajouter les Dépendances Python

### 🎯 EXERCICE : Installer SQLAlchemy et psycopg2

Utilisez `uv add` pour ajouter les dépendances nécessaires :

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

**Ce que fait cette commande :**

- Ajoute `sqlalchemy` et `psycopg2-binary` au fichier `pyproject.toml`
- Installe automatiquement les packages
- Met à jour le fichier de lock ( `uv.lock` )

**Vérifiez l'installation :**

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

Vous devriez voir : `SQLAlchemy 2.0.x`

## 5.4 - Créer le Fichier `database.py`

### 🎯 EXERCICE : Configuration de la base de données

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

```
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 les variables d'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 :

- Configure la connexion à PostgreSQL (ou SQLite en local)
- Crée le moteur SQLAlchemy avec une pool de connexions
- Fournit `get_db()` pour les dépendances FastAPI
- Fournit `init_db()` pour créer les tables

## 5.5 - Créer le Fichier `models.py`

### EXERCICE : Définir le modèle ORM

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

```
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 le schéma de la table `tasks` en base de données
- Chaque `Column` correspond à une colonne SQL
- Les enums `TaskStatus` et `TaskPriority` seront déplacés ici

## 5.6 - Modifier `app.py` : Supprimer le Stockage en Mémoire

### EXERCICE : Migration du code

⚠ **ATTENTION** : Vous allez **supprimer** et **remplacer** du code dans `app.py` . Suivez attentivement !

#### Étape 1 : Ajouter les imports en haut du fichier

Après les imports existants, ajoutez :

```

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

```

#### Étape 2 : SUPPRIMER les anciennes définitions

**✗ SUPPRIMEZ** ces lignes (dans `app.py` ) :

```
# SUPPRIMEZ cette classe (TaskStatus)
class TaskStatus(str, Enum):
    TODO = "todo"
    IN_PROGRESS = "in_progress"
    DONE = "done"

# SUPPRIMEZ cette classe (TaskPriority)
class TaskPriority(str, Enum):
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"

# SUPPRIMEZ cette ligne (tasks_storage)
tasks_storage: List[Task] = []
```

**Pourquoi ?** Ces éléments sont maintenant dans `models.py` !

### Étape 3 : Modifier la fonction lifespan

**✗ REMPLACEZ cette section :**

```
@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 ce nouveau code :**

```
@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)
    if not app.dependency_overrides: # Seulement en production
        init_db()
    else:
```

```
logger.info("Test mode - skipping database initialization")

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

#### Étape 4 : Modifier TOUS les endpoints

Chaque endpoint doit maintenant utiliser la base de données au lieu de `tasks_storage`.

##### ◆ Endpoint GET /tasks

❌ REMPLACEZ :

```
@app.get("/tasks", response_model=list[Task])
async def get_tasks():
    """Get all tasks."""
    logger.info("Fetching all tasks")
    return tasks_storage
```

✅ PAR :

```
@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
```

##### ◆ Endpoint POST /tasks

❌ REMPLACEZ :

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

    task = Task(
        id=str(uuid4()),
        **task_data.model_dump()
```

```

)
tasks_storage.append(task)

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

```

✓ PAR :

```

@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

```

◆ Endpoint GET /tasks/{task\_id}

✗ REMPLACEZ :

```

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

    task = next((t for t in tasks_storage if t.id == task_id), None)
    if not task:
        raise HTTPException(status_code=404, detail="Task not found")

    return task

```

✓ PAR :

```
@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
```

#### ◆ Endpoint PUT /tasks/{task\_id}

#### ✗ REMPLACEZ :

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

    task = next((t for t in tasks_storage if t.id == task_id), None)
    if not 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(task, field, value)

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

#### ✓ PAR :

```
@app.put("/tasks/{task_id}", response_model=Task)
async def update_task(task_id: str, task_update: TaskUpdate, db: Session):
    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

```

◆ Endpoint DELETE /tasks/{task\_id}

✗ REMPLACEZ :

```

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

    task = next((t for t in tasks_storage if t.id == task_id), None)
    if not task:
        raise HTTPException(status_code=404, detail="Task not found")

    tasks_storage.remove(task)
    logger.info(f"Task deleted: {task_id}")

```

✓ PAR :

```

@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}")

```

Étape 5 : Modifier le Health Check

 **REPLACEZ :**

```
@app.get("/health")
async def health_check():
    """Health check endpoint."""
    return {
        "status": "healthy",
        "timestamp": datetime.utcnow().isoformat(),
        "environment": os.getenv("ENVIRONMENT", "development"),
        "version": "1.0.0"
    }
```

 **PAR :**

```
from sqlalchemy import text

@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"
    except Exception as e:
        logger.error(f"Database health check failed: {e}")
        db_status = "disconnected"

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

## 5.7 - Adapter les Tests

 **EXERCICE : Configurer les tests avec la base de données**

Ouvrez `backend/tests/conftest.py` et remplacez tout le contenu par :

```

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_module.path.exists(TEST_DB_FILE):
        os_module.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()

```

#### Ce que fait ce code :

- Crée une base SQLite temporaire pour les tests
- Nettoie les données entre chaque test
- Override `get_db()` pour utiliser la DB de test

## 5.8 - Tester Localement

### EXERCICE : Vérifier que tout fonctionne

```

cd backend

# Lancer les tests
uv run pytest

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

# Dans un autre terminal, tester l'API
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 database",
    "description": "Vérifier PostgreSQL",
    "status": "todo",
    "priority": "high"
  }'
```

✅ **Checkpoint** : Les tests doivent passer et l'API doit fonctionner !

En local, SQLite est utilisé automatiquement ( `taskflow.db` créé dans `backend/` ).

## 5.9 - Connecter PostgreSQL sur Render

### 🎯 EXERCICE : Lier la base de données au backend

1. Allez sur <https://dashboard.render.com>
2. Cliquez sur votre service **taskflow-backend**
3. Allez dans **"Environment"** (menu de gauche)
4. Cliquez **"Add Environment Variable"**
5. Ajoutez :

```
Key: DATABASE_URL
Value: postgresql://taskflow_db_user:mot_de_passe@dpg-xxxxx-a/taskflow_db
```

⚠️ **Utilisez l'URL "Internal Database URL"** que vous avez copiée en 5.2 !

6. Cliquez **"Save Changes"**
7. Render va automatiquement redéployer le backend

## 5.10 - Vérifier le Déploiement avec PostgreSQL

### 🎯 EXERCICE : Tester en production

```
# 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 database test",
    "status": "todo",
    "priority": "high"
  }'

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

### Test final :

1. Créez une tâche depuis le frontend en production
2. Dans Render Dashboard, allez dans le backend → **"Manual Deploy"** → **"Deploy latest commit"**
3. Attendez le redéploiement (2-3 minutes)
4. **Rafraîchissez le frontend** → La tâche doit toujours être là ! 🎉

✅ **Félicitations !** Vos données persistent maintenant entre les redémarrages !

## 5.11 - Checklist de Migration Complète

### Fichiers créés :

- ☐ backend/src/database.py
- ☐ backend/src/models.py

### Fichiers modifiés :

- ☐ backend/pyproject.toml (dépendances ajoutées)
- ☐ backend/src/app.py (migration vers DB)
- ☐ backend/tests/conftest.py (tests adaptés)

### Déploiement :

- ☐ Base de données PostgreSQL créée sur Render
  - ☐ Variable `DATABASE_URL` configurée sur le backend
  - ☐ Backend redéployé avec succès
  - ☐ Health check montre `"database": "connected"`
  - ☐ Les tâches persistent après redéploiement
- 

## Phase 6 : Test et Validation (30 min)

---

### 6.1 - Checklist de Déploiement

#### EXERCICE : Vérifier que tout fonctionne

##### Backend :

- ☐ URL accessible : `https://taskflow-backend-XXXX.onrender.com`
- ☐ Health check : `/health` retourne `{"status": "healthy"}`
- ☐ API Docs : `/docs` fonctionne
- ☐ Endpoints API : `/tasks` répond
- ☐ CORS configuré : Requêtes du frontend acceptées

##### Frontend :

- ☐ URL accessible : `https://taskflow-frontend-XXXX.onrender.com`
- ☐ Page se charge sans erreur
- ☐ Connexion au backend fonctionne
- ☐ Création de tâches fonctionne
- ☐ Suppression de tâches fonctionne
- ☐ Modification de tâches fonctionne

##### CI/CD :

- ☐ GitHub Actions passe tous les tests
- ☐ Auto-deploy activé
- ☐ Push sur main déclenche un redéploiement


## 6.2 - Tester les Scénarios Réels

### EXERCICE : Cas d'utilisation complets

#### Scénario 1 : Créer une tâche

1. Ouvrez votre frontend en production
2. Créez une tâche "Déploiement réussi !"
3. Priorité : High
4. Vérifiez qu'elle apparaît dans la colonne "À Faire"

#### Scénario 2 : Modifier une tâche

1. Cliquez sur " " pour éditer
2. Changez le statut en "En Cours"
3. Vérifiez qu'elle se déplace dans la bonne colonne

#### Scénario 3 : Partager avec un collègue

1. Copiez l'URL de votre frontend
2. Envoyez-la à un collègue
3. Il doit voir les mêmes tâches !

#### Scénario 4 : Tester sur mobile

1. Ouvrez l'URL sur votre téléphone
2. L'interface doit être responsive

## 6.3 - Déboguer les Problèmes Courants

### "Connection Error" dans le frontend

**Cause :** `VITE_API_URL` mal configuré

**Solution :**

1. Vérifiez dans Render Frontend → Environment
2. La variable doit être : `VITE_API_URL=https://taskflow-backend-XXXX.onrender.com`
3. Redéployez

## ✖ CORS Error

**Cause :** Backend ne permet pas l'origine du frontend

**Solution :**

```
# Dans Render Backend → Environment  
CORS_ORIGINS=https://taskflow-frontend-XXXX.onrender.com
```

## ✖ Backend "Service Unavailable"

**Cause :** Health check échoue

**Solution :**

1. Vérifiez les logs backend
2. Assurez-vous que `/health` répond
3. Vérifiez que le port est `$PORT` (fourni par Render)

## ✖ Frontend montre du code au lieu de l'app

**Cause :** Publish Directory incorrect

**Solution :**

```
Publish Directory: dist # PAS frontend/dist !
```

---

## ✅ Checklist de Fin d'Atelier

---

**Services Déployés :**

- ☐ Backend en production et accessible
- ☐ Frontend en production et accessible
- ☐ Communication frontend ↔ backend fonctionne
- ☐ Health checks configurés

## Configuration :

- ☐ Variables d'environnement configurées
- ☐ CORS correctement configuré
- ☐ Build filters optimisés (monorepo)
- ☐ Auto-deploy activé

## Tests :

- ☐ Création de tâches fonctionne
- ☐ Modification de tâches fonctionne
- ☐ Suppression de tâches fonctionne
- ☐ Application accessible depuis n'importe où

## Documentation :

- ☐ URLs notées quelque part :
  - Backend : `https://taskflow-backend-XXXX.onrender.com`
  - Frontend : `https://taskflow-frontend-XXXX.onrender.com`

---

## Ce que Vous Avez Appris

Félicitations ! 🎉 Vous avez maintenant :

✅ Déployé une application full-stack en production    ✅ Configuré CORS pour la production    ✅ Utilisé des variables d'environnement    ✅ Mis en place un monitoring avec health checks    ✅ Configuré un déploiement automatique (CI/CD complet)


Votre application est accessible partout dans le monde ! 🌍

---

## Pour Aller Plus Loin

### Améliorations possibles :

1. Base de données persistante PostgreSQL 🌟 **DISPONIBLE MAINTENANT !**

-  [Guide complet : Intégration PostgreSQL](#)
- Remplacer le stockage en mémoire par une vraie base de données
- Déployer PostgreSQL sur Render
- Utiliser SQLAlchemy ORM
- **Durée** : 60-90 minutes
- **Prérequis** : Avoir complété les phases 1-5 de cet atelier

## 2. Domaine personnalisé

- Acheter un nom de domaine
- Le connecter à Render

## 3. Authentification

- Ajouter un login/signup
- Protéger les routes

## 4. Monitoring avancé

- Intégrer Sentry pour les erreurs
- Ajouter des metrics avec Prometheus

## 5. Tests E2E

- Playwright ou Cypress
- Tests automatisés sur l'environnement de production

---

## Ressources

---

### Atelier 3 - Extensions :

- [Guide PostgreSQL Database](#) - Intégration base de données (Partie 5)
- [Migration Checklist](#) - Guide visuel de migration
- [Backend README](#) - Documentation technique complète

### Documentation Externe :

- [Render Documentation](#)



- [FastAPI Deployment Guide](#)
  - [Vite Production Build](#)
  - [Managing Environment Variables](#)
  - [SQLAlchemy Documentation](#) (pour la base de données)
- 



## Notes Finales

---

### Limitations du plan gratuit Render :

- Services s'endorment après 15 min d'inactivité
- Réveil = 30-60 secondes de latence
- Pour éviter ça : Plan payant ou service de "keep-alive"

### Coûts (si vous passez au payant) :

- Starter plan : ~7\$/mois par service
- Adapté pour petits projets personnels

### Alternatives à Render :

- Vercel (frontend)
  - Railway (full-stack)
  - Fly.io (backend)
  - Heroku (full-stack, plus cher)
- 

**Version 1.0** - Atelier 3 : Déploiement en Production 🚀