**✅ ÉTAPE 1 : Initialiser le dépôt Git**

**But** : dire à Git que ce dossier est un projet sous gestion de version.

**📌 Commande**

Dans le terminal (PyCharm ou autre), place-toi dans le dossier churn\_V1 et tape :

***git init***

**✅ Ce que ça fait :**

* **Cela crée un dossier caché .git à la racine de ton projet. Ce dossier contient toutes les informations de versionnement pour Git.**
* **Git te donne un message comme ça : Initialized empty Git repository in ...**

**Résultat attendu :**

* **Après avoir lancé git init, tu devrais voir dans le terminal une confirmation indiquant que le dépôt Git a bien été initialisé, comme par exemple :**

**Initialized empty Git repository in C:/Users/Username/Documents/Churn/churn\_V1/.git/**

**Vérification**

1. **Vérifie si le dossier .git a été créé** :
   * Ouvre l'Explorateur de fichiers sur ton PC.
   * Va dans ton dossier churn\_V1.
   * Assure-toi que les fichiers cachés sont visibles en activant "Afficher les fichiers cachés" dans l'explorateur (Sous **Affichage**, coche **Éléments masqués**).
   * Tu devrais voir un dossier .git dans ton répertoire churn\_V1.

**✅ Étape 2 : Créer le fichier .gitignore**

Le fichier **.gitignore** permet à Git de savoir quels fichiers il ne doit pas suivre. Par exemple, tu veux que Git ignore les fichiers temporaires, les fichiers système ou les fichiers de l'environnement virtuel (comme ceux créés par Python).

1. **Crée le fichier .gitignore** à la racine de ton projet :
   * Va dans le dossier churn\_V1 et crée un fichier nommé .gitignore.
   * Pour cela, tu peux créer un fichier texte classique et simplement le renommer en .gitignore (assure-toi que l'extension .txt soit bien supprimée, sinon ça ne fonctionnera pas).
2. **Ajoute des règles de base dans ce fichier** :  
   Voici quelques règles standard à ajouter dans ton fichier .gitignore pour ignorer les fichiers et dossiers courants qui ne doivent pas être suivis par Git :

**Contenu du fichier .gitignore** :

# Ignorer les fichiers de l'environnement virtuel

.env/

venv/

env/

# Ignorer les fichiers Python compilés

\*.pyc

\*.pyo

# Ignorer les fichiers d'IDE

.idea/

\*.iml

# Ignorer les fichiers système

Thumbs.db

.DS\_Store

* + **.env/** : ignorer les dossiers d'environnements virtuels.
  + \*\**.pyc, .pyo* : ignorer les fichiers Python compilés.
  + **.idea/** : ignorer les dossiers et fichiers créés par PyCharm (IDE).
  + **Thumbs.db, .DS\_Store** : ignorer les fichiers système générés sous Windows ou macOS.

1. **Sauvegarde le fichier .gitignore**.

**✅ Ce que ça fait :**

* Le fichier .gitignore indique à Git quels fichiers ou dossiers ne doivent pas être suivis ni envoyés vers le dépôt Git.
* En utilisant ce fichier, tu évites de pousser des fichiers inutiles ou sensibles vers ton dépôt Git, comme ton environnement virtuel ou des fichiers de configuration IDE.

**Vérification**

1. Si tu ouvres ton terminal dans PyCharm, tu peux vérifier si le fichier .gitignore a été bien ajouté à ton projet en utilisant cette commande :

dir

Sous Windows, cette commande listera les fichiers et dossiers de ton répertoire actuel. Tu devrais voir ton fichier .gitignore dans la liste.

1. **Vérifie aussi que les fichiers ignorés ne sont pas suivis par Git** :  
   Si tu as déjà ajouté des fichiers à Git (avant de créer .gitignore), tu peux vérifier les fichiers suivis avec :

git status

Cela te montrera les fichiers suivis ou modifiés par Git. Si des fichiers comme .env ou d'autres éléments ignorés apparaissent, cela signifie qu'ils sont déjà suivis.

**✅ Étape 3 : Ajouter les fichiers au staging (préparer le commit)**

Maintenant que ton fichier .gitignore est prêt et que tu as ton projet dans le dossier churn\_V1, tu vas **indiquer à Git quels fichiers tu veux suivre**. Pour cela, on les ajoute au "**staging area" (zone de préparation).**

**📌 Commande à exécuter :**

Ouvre le **terminal dans PyCharm** (Alt + F12 si besoin), puis tape :

***git add .***

✅ **Explication** :

* git add . : ajoute **tous les fichiers** (sauf ceux ignorés par .gitignore) au "staging area", c’est-à-dire la zone de préparation avant le commit.

💡 Tu peux aussi ajouter un fichier précis avec git add nom\_du\_fichier.py si tu veux être plus sélectif.

**🔍 Vérification :**

Après avoir ajouté les fichiers, tu peux vérifier ce qui est prêt à être "commit" en tapant :

***git status***

Tu verras les fichiers en vert = prêts à être commités ✅

✅ **Objectif** :  
Préparer une "photo" de ton projet à ce moment précis. Une fois les fichiers ajoutés, tu pourras faire un commit.

**✅ Étape 4 : Faire un commit (créer un snapshot de ton projet)**

Maintenant que tes fichiers sont dans la zone de staging, tu vas faire un **commit**, c’est-à-dire une **photo figée de ton projet à un instant donné**, avec un message qui explique ce que tu as fait.

**📌 Commande à exécuter :**

***git commit -m "Initialisation du projet Churn V1"***

✅ **Explication** :

* git commit : crée un snapshot.
* -m : permet de fournir un **message de commit**.
* "Initialisation du projet Churn V1" : c’est un message clair pour décrire l’état ou le changement.

📝 Chaque commit doit avoir un message **clair et concis**. Tu peux penser à ça comme un journal de bord de ton projet.

**🔍 Vérification :**

Tu peux voir ton commit avec :

***git log --oneline***

Tu verras une ligne avec un identifiant court (comme a1b2c3d) suivi de ton message.

**✅ Étape 5 : Créer un dépôt GitHub distant et le connecter à ton dépôt local**

Maintenant, tu vas créer un **dépôt distant sur GitHub** pour y **synchroniser ton dépôt local**.

**🛠️ Étape 5.1 : Créer un dépôt sur GitHub**

1. Va sur <https://github.com>.
2. Clique sur le bouton **"New"** (ou **"New repository"**).
3. Renseigne :
   * **Repository name** : churn\_v1
   * **Description** (facultatif) : "Projet de prédiction de churn avec pipeline CI/CD"
   * **Public** ou **Private** selon ton choix
   * Ne coche **rien d’autre** (pas de README, .gitignore, etc.)
4. Clique sur **Create repository**

Tu devrais maintenant voir une page avec des instructions pour connecter ton dépôt local.

**🛠️ Étape 5.2 : Connecter ton dépôt local à GitHub**

Dans le terminal de PyCharm, exécute la commande suivante **(en remplaçant l’URL par la tienne)** :

***git remote add origin https://github.com/ton-nom-utilisateur/churn\_v1.git***

Remplace ton-nom-utilisateur par ton identifiant GitHub.

Puis :

***git push -u origin master***

**🔍 Explication :**

* git remote add origin ... : lie ton dépôt local à GitHub.
* git push -u origin master : envoie le contenu de ta branche master sur GitHub et suit la branche à distance.

**✅ Étape 6 : Ajouter le pipeline CI/CD (.github/workflows/ci.yml)**

**📌 Rappel : Qu’est-ce que CI/CD ?**

* **CI (Intégration Continue) : tester automatiquement ton projet à chaque modification (ex : commit, pull request)**
* **CD (Déploiement Continu) : (plus tard) déployer automatiquement si les tests passent**

**🛠️ Étape 6.1 : Créer la structure du pipeline GitHub Actions**

1. **À la racine de ton projet :**
   * **Crée un dossier .github**
   * **Dans .github, crée un dossier workflows**
   * **Dans workflows, crée un fichier ci.yml**

**Arborescence :**

churn\_V1/

│

├── .github/

│ └── workflows/

│ └── ci.yml

**🛠️ Étape 6.2 : Contenu du fichier ci.yml**:

***name: Python CI***

***on:***

***push:***

***branches: [ master, main ]***

***pull\_request:***

***branches: [ master, main ]***

***jobs:***

***build:***

***runs-on: ubuntu-latest***

***steps:***

***- name: Checkout code***

***uses: actions/checkout@v3***

***- name: Set up Python***

***uses: actions/setup-python@v4***

***with:***

***python-version: '3.12'***

***- name: Install dependencies***

***run: |***

***python -m pip install --upgrade pip***

***pip install -r requirements.txt***

***- name: Run tests***

***run: |***

***pytest***

🧠 **Ce que fait ce pipeline** :

1. Il se déclenche sur chaque push ou pull request vers master/main
2. Il installe Python 3.12 dans un environnement Ubuntu
3. Il installe tes dépendances via requirements.txt
4. Il lance tes tests avec pytest

✅ Une fois que tu as bien créé ce fichier :

1. Fais ***git add .***
2. Puis ***git commit -m "Ajout du pipeline CI GitHub Actions"***
3. Et pousse-le sur GitHub : ***git push origin master***

**✅ Étape 7 : Vérifier le pipeline CI sur GitHub**

Maintenant que tu as poussé ton fichier ci.yml :

**📍 Va sur GitHub :**

1. Ouvre ton dépôt sur GitHub
2. Clique sur l’onglet **“Actions”** (il est à côté de “Code” et “Issues”)
3. Tu devrais voir un **workflow en cours** (ou déjà terminé)
4. Clique dessus pour **voir les détails** :
   * **Check-out du code**
   * **Installation de Python**
   * **Installation des dépendances**
   * **Exécution des tests**

**🧠 Introduction rapide au CD**

**CD = Continuous Deployment**, ça veut dire :

**Une fois que le code passe les tests automatiquement, il est déployé automatiquement** vers ton environnement (prod/staging).

Pas besoin de le faire manuellement → GitHub s’en occupe une fois que tu push.

**⚙️ Outils qu’on peut utiliser**

Puisque tu es sur GitHub et que ton app est en **FastAPI + Python**, on a plusieurs options :

**🔹 Option 1 : Deploy sur Render.com**

* Simple à mettre en place
* Gratos pour un petit projet
* Très bien pour FastAPI
* Peut se déclencher automatiquement depuis GitHub

**🔹 Option 2 : Deploy avec Docker + GitHub Actions**

* Tu fais une image Docker de ton app
* GitHub Actions build/push ton image sur DockerHub
* Puis déploie vers un serveur (ex : Railway, Render, EC2…)

**🔹 Option 3 : Heroku**

* Un classique mais un peu en perte de vitesse
* Gratuit limité, plus de carte bancaire obligatoire

**💡 Ma reco**

Vu que tu veux aller **pas à pas**, je te recommande :

✅ **Render.com** — Très facile, déploiement automatique depuis GitHub, pas besoin de Docker tout de suite.

**📦 Ce que tu vas obtenir :**

* Une API FastAPI en ligne (genre : https://tonapp.onrender.com)
* À chaque push sur main/master, l’app est **déployée automatiquement**
* Tu peux tester l’API en ligne directement

**🧩 Étape 1 : Créer un Dockerfile pour ton app FastAPI**

Voici un Dockerfile prêt à l'emploi que tu vas placer **à la racine** de ton projet (churn\_V1/ par exemple) :

***# Utilise une image Python officielle comme base***

***FROM python:3.12-slim***

***# Crée un répertoire pour l'application***

***WORKDIR /app***

***# Copie les fichiers nécessaires***

***COPY . /app***

***# Installe les dépendances***

***RUN pip install --upgrade pip***

***RUN pip install -r requirements.txt***

***# Expose le port sur lequel FastAPI tourne***

***EXPOSE 8000***

***# Commande pour démarrer l'application***

***CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]***

**📦 Étape 2 : Créer un compte DockerHub**

1. Va sur https://hub.docker.com
2. Crée un compte (gratuit)
3. Crée un **repository public** (nom : churn-api par exemple)

Tu vas ensuite **pousser l’image depuis GitHub Actions** dans ce repo.

**🔁 Étape suivante : DockerHub (hébergement d’images Docker)**

DockerHub est comme « GitHub pour Docker » : c’est là où on pousse nos images pour ensuite les tirer ailleurs (Render dans notre cas).

**1 : Créer un compte DockerHub**

Va sur https://hub.docker.com/ et crée un compte si tu n’en as pas encore.  
👉 Garde bien **ton nom d’utilisateur DockerHub** en tête (ex : nadir92).

Dis-moi quand c’est fait ✅, et je t’explique comment créer le ci-cd.yml qui :

1. construit ton image Docker à chaque push sur main,
2. la pousse vers DockerHub.

**Étape suivante : Créer un repository DockerHub**

1. Va sur https://hub.docker.com/repositories (tu dois être connecté).
2. Clique sur **“Create Repository”**.
3. Remplis le formulaire :

| **Champ** | **Valeur** |
| --- | --- |
| **Name** | churn-app *(par exemple)* |
| **Visibility** | Public *(plus simple pour commencer)* |
| Description | Déploiement de mon app Churn avec Docker *(facultatif)* |

1. Clique sur **Create**.

**🔹 Lier ton Repository GitHub avec DockerHub :**

Pour que chaque fois que tu pushes ton code sur GitHub, l'image Docker soit automatiquement construite et envoyée à DockerHub, on va créer un **workflow GitHub Actions**. Ce workflow va :

1. **Construire l’image Docker**.
2. **La pousser sur DockerHub** dans ton repository.
3. **La rendre disponible pour être déployée sur Render** (ou autre service).

**Étape 1 : Créer un fichier GitHub Action**

Dans ton projet, tu devras créer un fichier **.github/workflows/docker.yml** (ou tout autre nom de ton choix dans le dossier .github/workflows/).

**Exemple de contenu pour docker.yml :**

***# Nom du workflow***

***name: Build and Push Docker Image to DockerHub***

***# Ce workflow se déclenche sur un `push` dans les branches master ou main.***

***on:***

***push:***

***branches:***

***- master***

***- main***

***# Définition des jobs qui vont s'exécuter.***

***jobs:***

***build:***

***# Spécifie l'environnement d'exécution, ici on utilise Ubuntu.***

***runs-on: ubuntu-latest***

***steps:***

***# 1. Étape de récupération du code source depuis GitHub***

***- name: Checkout code***

***uses: actions/checkout@v3***

***# `actions/checkout` permet de récupérer ton code dans le workflow***

***# sans cela, ton code ne sera pas accessible dans l'environnement d'exécution***

***# 2. Connexion à DockerHub avec tes identifiants***

***- name: Log in to DockerHub***

***uses: docker/login-action@v2***

***with:***

***username: ${{ secrets.DOCKER\_USERNAME }}***

***password: ${{ secrets.DOCKER\_PASSWORD }}***

***# Cette étape utilise une action GitHub officielle qui permet de se connecter***

***# à DockerHub en utilisant les secrets stockés dans GitHub.***

***# Les secrets sont des variables d'environnement sécurisées dans ton repository GitHub.***

***# Nous avons besoin de ton `DOCKER\_USERNAME` et de ton `DOCKER\_PASSWORD` (ou token DockerHub) ici.***

***# 3. Étape de construction de l'image Docker à partir du Dockerfile***

***- name: Build Docker image***

***run: |***

***docker build -t nadiroun/churn-app:latest .***

***# `docker build` va créer une image Docker à partir du Dockerfile situé à la racine de ton projet.***

***# Le `-t nadiroun/churn-app:latest` donne un nom à l'image (ici "nadiroun/churn-app") et une étiquette "latest".***

***# Le `.` à la fin signifie que Docker va utiliser le contexte de construction dans le répertoire actuel.***

***# 4. Étape de push de l'image vers DockerHub***

***- name: Push Docker image to DockerHub***

***run: |***

***docker push nadiroun/churn-app:latest***

***# Cette étape pousse l'image Docker vers ton repository DockerHub (ici "nadiroun/churn-app").***

***# Cela rend l'image accessible depuis DockerHub, prête à être utilisée dans un déploiement.***