Projet python DIGICHEESE (PMP : Plan nagement de Projet + Doc technique) - Groupe2

- Imen KHAMMASSI
- Stanislas DELANNOY
- Xavier DEGUERCY
- Thi Thu Hien NGUYEN

- Le projet : un petit logiciel “serveur” (une API) qui permet de gérer des données pour DIGICHEESE.
  - API = une application qui répond à des URL (exemple : /admin/communes) et renvoie des données (Json)
  - On va utiliser Python + FastAPI (framework d’API)
  - On va stocker les données dans une base MYSQL
  - On doit fournir :
    - L’API
    - La documentation automatique via swagger
    - Des tests
    - Un repo Git avec les branches demandées
    - Des documents de présentation technique et fonctionnel
  - (API backend + Swagger + tests + MYSQL + requirements.txt + documentation + branches)
- Le livrable
  - Un dépot Git complet, avec :
    - Les branches master, dev, test, prod + branche par développeur
    - Le code python (sources)
    - Des scripts MYSQL (ou de quoi créer la base)
    - Un fichier requirements.txt
  - Une base MYSQL
    - Une base qui contient les tables nécessaires pour les fonctionnalités “Admin”
  - Une API FastAPI fonctionnelle
    - CRUD Admin obligatoire
    - Swagger opérationnel pour simuler un front
  - Des tests automatisés
    - Avec Pytest ou Unittest
  - De la documentation
    - Un dossier documentation technique et utilisation, avec README.md qui présente l’équipe et les roles
    - Un dossier avec les scénarios de test et les scripts
    - Il faut prendre en compte les attentes qualité/ sécurité/ monitoring indiquées dans les consignes
- Le périmètre fonctionnel : Ce que l’API doit permettre de faire

    - Le cahier des charges dit que l'application est centrée sur des roles : Admin, Opérateur Colis, Opérateur Stock
    - Ce qui est obligatoire : Le role Admin
        - L'espace Admin doit permettre de gérer (CRUD = Créer, Lire, Modifier, Supprimer)
            - Des communes
            - Des objets (articles)
            - Des conditionnements
            - Des poids
            - Des poids-vignettes
                - Donc l'API doit au minimum exposer des endpoints pour faire des opérations
    - Ce qui est optionnel : Le role Opérateur Colis
        - On peut ajouter le CRUD des clients (gestion des clients)
        - Le cahier des charges confirme : coté Opérateur Colis, il y a "Gestion des clients"
- CRUD (c'est quoi concrètement)
    - Pour chaque "objet" du système (une commune par exemple), on doit fournin des routes qui permettent :
        - Créer (exemple ajouter une nouvelle commune)
        - Lister (Exemple : Récupérer la liste de toutes les communes)
        - Afficher un élément (Exemple : Récupérer une commune précise par son identifiant)
        - Modifier (Exemple : Changer le nom/ la ville/ le code postal... d'une commune)
        - Supprimer (Exemple : Supprimer une commune)
    - Dans une API, on fait ça avec des routes "standards", souvent :
        - POST pour créer
        - GET pour lire
        - PUT/ PATCH pour modifier
        - DELETE pour supprimer
- Les tables/ données (sur quoi on doit travailler),
    - On a le fichier Exemples_models.py qui montre des modèles SQLModel existants
    - Les tables Admin indispensables (celles du cahier des charges)
        - Le CDC liste les tables suivantes pour les paramètres métiers Admin :
            - t_poids
            - t_poidsV
            - t_objet
            - t_conditionnement
            - t_commune
        - Dans les modèles d'exemples, on retrouve :
            - Communes dans t_communes avec id, dep, cp, ville
            - Objet dans t_objet avec des champs comme codeobj, libobj, puobj, poidsobj...
            - Conditionnement dans t_conditionnement avec idcondit, libcondit, poidscondit, prixcondit...
            - Poids dans t_poids (id, valmin, valtimbre)
            - Poids-vignette dans t_poidsV (id, valmin, valtimbre)
            - (Option) Client dans t_client avec nom, prenom, email, téléphone...
                - En pratique, on va reprendre ces modèles (ou on en inspire) pour que l'API écrive/ lise dans MYSQL
- Swagger : A quoi ça sert et ce qu'on doit faire

- Swagger (dans FastAPI : la page /docs) est une interface web qui :
  - Lise toutes les routes
  - Explique quelles données envoyer
  - Permet de tester les routes sans Postman
- Concrètement :
  - Pour chaque endpoint
    - On voit le nom de l'opération
    - On voit les champs attendus (schémas)
    - On a des exemples (idéalement)
    - Et quand une erreur arrive (exemple : id inconnu), il faut que l'API renvoie une réponse compréhensible

- Tests : ce qui est attendu
  - On doit mettre en place des tests avec Pytest ou Unittest
  - Pour chaque ressource admin (communes, objets, conditionnements, poids, poids-vignettes)
    - 1 test nominal (ça marche quand on fait une requete correcte)
    - 1 test erreur (ça échout correctement quand on fait n'importe quoi)
      - Exemple erreur : envoyer un champ manquant ou demander /communes/999999 (id inexistant)
  - L'idée, c'est de prouver que :
    - L'API marche
    - Qu'elle gère bien les cas d'erreur
- Qualité/ Sécurité/ Monitoring (Ce qu'on attend de nous, au moins en doc)
  - Les consignes d'évaluation demandent notamment
    - Code comforme aux normes de sécurité (exemple OWASP)
    - Doc des protections des données (RGPD, gestion des accès...)
    - Scénarios de tests détaillés + résultats interprétés
    - KPIs et alertes de monitoring "définis et opérationnels"
  - Meme si on ne fait pas un monitoring industriel, on peut faire :
    - Un endpoint /health qui répond "OK"
    - Un endpoint /version
    - Une petite section doc "KPIs" (exemple : temps de réponse, taux d'erreurs, disponibilité...)
- Objectif final (ce que "finir le travail" veut dire)
  - Pour que le TP7 soit terminé, il faut qu'on puisse faire :
    - On lance la base MYSQL + L'API (idéalement via Docker, ou sinon en local)
    - Swagger (/docs) affiche tous les endpoints attendus
    - Les endpoints CRUD Admin obligatoires fonctionnent avec validation, erreurs propres et persistance MYSQL
    - Il y a des tests automatisés (au min 1 test nominal + 1 test erreur par ressource CRUD)
    - La doc est complète (fonctionnelle + technique + utilisation + scénarios de tests)
    - Le repo est "propre" : requirements, env.example, conventions, et branches demandées

- Ce qu'on a déjà dans le repos (base)
  - Le dépot contient déjà les éléments "socle" attendus :
    - Un dossier /src pour le code
    - Un dossier tests/ pour les tests
    - Un dossier DOC/ pour la documentation
    - .env et .env.example
    - Dockerfile et docker-compose.yml
    - requirements.txt
    - Une structure annoncée : routes/ , services/ , repositories , models/ , conf/ , utils/
      - Donc le travail de finition consiste surtout à implémenter correctement les CRUD, brancher la base, documenter Swagger, et écrire les tests

- Rappels essentiels des commandes Git
  - Vérifier que Git est installé
    - git --version
      - Afficher la version de Git
  - Cloner un depot existant DIGICHEESE
    - git clone lien
    - Créer un dossier TP_CHEESE avec tout le projet
      - Entrer dans le projet (cd TP_CHEESE)
  - Voir l'état du depot
    - git status
      - Indique sur quelle branche on est
      - Quels fichiers on été modifiés
      - Quels fichiers sont prets à etre commités
    - Voir les branches existantes
      - git branch
        - Affiche les branches locales (* indique la branche actuelle)
      - git branch -a
        - Affiche les branches distantes
    - Changer de branche
      - git checkout nom-de-la-branche
        - Exemple : git checkout dev/ git checkout test/ git checkout prod
        - Permet de travailler sur la bonne branche
    - Créer une nouvelle branche
      - git checkout -b ma-branche
        - Exemple : git checkout -b feature-communes
    - Mettre à jour la branche avec le dépot distant
      - Avant de coder, toujours récupérer les dernières modifications
        - git pull
          - Récupère les changements de la branche distante actuelle
    - Ajouter des fichiers à l'index (avant commit)
      - Quand on modifie des fichiers

        - git add nom-du-fichier
            - Exemple : git src/main.py
    - Ajouter tous les fichiers modifiés
        - git add
- Créer un Commit (sauvegarde Git)
    - git commit -m “message clair du commit “
        - Exemple de bons messages (git commit -m “Ajout CRUD communes”/ git commit -m “Connexion MYSQL fonctionnelle”/ git commit -m “Ajout tests communes”)
    - Un commit => Une étape logique du travail
- Envoyer son travail sur Github
    - git push
        - Si c’est la première fois sur cette branche
    - git push -u origin ma_branche
        - Envoyer le travail pour que le groupe puisse le voir
- Mettre à jour la branche avec dev
    - git checkout dev
    - git pull
    - git checkout ma-branche
    - git merge dev
        - ça évite les conflits à la fin
- Résoudre un conflit
    - Si Git dit qu’il ya un conflit
        - On ouvre le fichier concerné
        - Git affiche
            - <<<<<<< HEAD
            - ton code
            - =======
            - autre code
            - >>>>>>> dev
        - On choisit quoi garder
        - On supprime les marqueurs
        - Puis :
            - git add fichier
            - git commit -m “résolution conflit”
- Historique des commits
    - git log
    - Version simplifiée
        - git log --oneline (très utile pour comprendre ce qui a été fait)
- Supprimer une branche locale (quand fini)
    - git branch -d nom-de-la-branche
        - Exemple (git branch -d feature-communes”
- Commandes Git “réflexes”
    - A utiliser tt le temps
        - git status
        - git pull

        - git add
        - git commit -m “message”
        - git push
    - Workflow conseillé
        - On se place sur dev
        - On crée la branche perso
        - On code
        - On commit souvent
        - On push
        - On merge vers dev
    - Ce qu’il faut pas faire
        - Travailler directement sur master
        - Faire un seul gros commit à la fin
        - Oublier git pull avant de commencer
    - Recap
        - git clone URL
        - cd projet
        - git checkout dev
        - git pull
        - git checkout -b ma-branche
        - #Je code
        - git add
        - git commit -m “message clair”
        - git push -u origin ma-branche”

Proposition de plan de travail – STRUCTURE DU PROJET (En GROS)

- Jour 1 – Comprendre le socle + Préparation du terrain (sans coder)
    - Objectif
        - Comprendre le projet et préparer le terrain de travail
            - Se placer sur la branche de dev et vérifier que l’API démarre (Swagger OK)
            - Prendre connaissance des conventions du projet (structure, routes, erreurs, DB)
            - Repérer un CRUD exsitant comme modèle de référence (CRUD de Robin avec SQLAlchemy + attributes de Examples-Models.py)
            - Analyser le métier (Poids, Poids-Vignette, adresse, client, commune, conditionnement, objet, role, utilisateur) à partir des docs
            - Lister les endpoints et les fichiers qu’on doit créer
        - Sortie J1 :
            - Environnement OK
            - Périmètre clair
            - Structure comprise

- Jour 2 – Modèles + persistance
  - Objectif
    - Poser une BD fiable
      - Créer/ valider les modèles t_poids, t_poidsv, t_adresse, t_client, t_commune, t_conditionnement, t_objet, t_role, t_utilisateur.
      - Définir les relations nécessaires
      - Mettre à jour la persistance MYSQL
      - Vérifier que les opérations DB fonctionnent
    - Sortie J2 :
      - Modèles stables
      - DB opérationnelle
- Jour 3 – Services + routes (API CRUD complète)
  - Objectif
    - Exposer un CRUD complet et cohérent
      - Implémenter repositories et services
      - Créer les routes API pour Poids,Poids-Vignette, adresse, client, commune, conditionnement, objet, role, utilisateur.
      - Appliquer validation et gestion des erreurs
      - Vérifier le bon fonctionnement via Swagger
    - Sortie J3 :
      - CRUD (Poids, Poids-Vignette, adresse, client, commune, conditionnement, objet, role, utilisateur) fonctionnels dans Swagger
- Jour 4 – Tests (DOD) + Stabilisation
  - Objectif
    - Sécuriser le travail selon la DOD
      - Ecrire les tests (nominal + erreur)
      - Harmoniser réponses et codes HTTP
      - Corriger bugs et incohérence
      - Vérifier que le code n'impacte pas le reste du projet
    - Résultat fin J4 :
      - Tests passants
      - API stable
- Jour 5 – Documentation & livraison & Présentation
  - Objectif
    - Rendre un bloc propre et intégrable
      - Rédiger/ mettre à jour la documentation liée au périmétre
      - Relecture finale du code
      - Push final sur la branche et préparation du merge vers dev
    - Sortie J5 :
      - Bloc (Poids, Poids-Vignette, adresse, client, commune, conditionnement, objet, role, utilisateur) complet, documenté, pret à merger

Détails du projet

- Jour 1 – Comprendre le socle + Préparation du terrain (sans coder)
  - Objectif du J1
    - Le dépot est cloné sur mon pc
    - Chaque développeur touche à sa branche créée (par exemple, j'ai la branche dev/imen, il ne faut pas toucher main)
    - Environnement Python pret (venv activé + dépendances installées)
    - MYSQL est installé en local et la base du projet est créée
    - Le fichier .env est pret et pointe vers la base MYSQL
    - L'API démarre sans erreur et Swagger s'ouvre dans le navigateur (/docs)
    - Savoir exactement où coder Poids/ Vignette (structure)
    - Liste des endpoints et des fichiers à créer (pour J2 et J3)
  - Vérifier les outils indispensables (Python + Git)
    - Sur cmd
      - python --version
      - git --version
    - Résultat
      - Une version python s'affiche
      - Une version git s'affiche
  - Cloner le depot Github et entrer dans le dossier
    - Se placer dans le dossier de travail
      - cd fichier
      - git clone URL_GIT
      - cd TP_DIGICHEESE
    - Vérifier qu'on est au bon endroit
      - dir (on doit voir les dossiers/ fichiers du projet : exemple src, tests, DOC, requirements.txt, .env.example,...)
  - Se mettre sur la branche dev/imen (obligatoire)
    - Voir la branche actuelle
      - git branch
    - Créer ou activer la branche
      - Si dev/imen n'existe pas, par exemple
        - git checkout -b dev/imen
      - Si dev/imen existe par exemple
        - git checkout dev/imen
        - git pull
      - Vérification finale
        - git branch
  - Créer un environnement Python (venv) et installer les dépendances
    - Toujours à la racine du projet
      - Créer le venv :
        - python -m venv .venv
      - Activer le venv
        - .\.venv\scripts\Activate.ps1
          - si powershell ne veut pas l'activation, saisir : Set-ExecutionPolicy -Scope CurrentUser RemoteSigned
          - Puis réactive

        - .\.venv\scripts\Activate.ps1
- Installer les dépendances du projet
    - python -m pip install --upgrade pip
    - pip install -r requirements.txt
- Vérifier rapidement que les libs essentielles sont bien installées
    - python -c “import fastapi; import sqlmodel; print(‘ok’)” (on doit voir OK)
- Installer MYSQL en local
    - Installer MYSQL Community Server
        - Port 3306
        - Mot de passe root à déterminer
        - MYSQL Workbensh
- Vérifier que MYSQL tourne
    - Ouvrir MYSQL Workbensh et tester la connexion “Local instance”
- Créer la base de donnée du projet
    - Dans MYSQL Workbensh (nouvel onglet SQL)
        - CREATE DATABASE digicheese CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
    - Il faut utiliser le nom de la BD dans .en (exmple si le projet utilise digicheese_DB)
- Préparer le fichier .env (connexion DB)
    - Créer .env à partir du projet
        - copy .env.example .env
- Remplir le fichier .env correctement
    - DB_HOST=localhost
    - DB_PORT=3306
    - DB_USER=root (ou un user dédié)
    - DB_PASSWORD=mdp
    - DB_NAME=digicheese
        - NB : Le .env doit rester local, dans gitignore (on ne commite jamais les mdp)
- Lancer l’API et ouvrir Swagger
    - On doit pouvoir démarrer le backend
        - Démarrer l’API (toujours dans PowerShell, à la racine du projet .venv activé)
            - uvicorn src.main:app --reload
        - Ouvrir l’API :
            - http://127.0.0.1:8000/docs
- Repérer la structure du projet (où coder Poids/ Poids-Vignette
    - Sans modifier le code, il faut explorer :
        - src/models/ : modèle SQLModel ou SQLAlchemy
        - src/repositories/ : accès DB (CRUD SQL)
        - src/services/ : logique métier (validation, règles)
        - src/routes/ : Endpoints FastAPI (Routers)
        - src/conf/db.py : Connexion DB (engine + session)
        - tests/ : tests PyTest
        - DOC/ : documentation
- Actions importantes J1
    - CRUD repéré (CRUD de Robin avec SQLAlchemy :
      🌐 Exemple-FastAPI-Docker/api/models/client.py at main · T-lamo/Exemple-FastAPI-...

+ Attributs de Examples-Models.py (remplacer SQLModel par SQLAlchemy)
    - Comment les fichiers sont nommés
    - Comment les Routes appellent les Services
    - Comment les Services appellent les Repositories
    - Comment sont gérées les erreurs (404, 400)
    - Comment sont déclarés les Schémas Swagger
- Préparer le périmètre (t_poids/ t_poidsv, adresse, client, commune, conditionnement, objet, role, utilisateur) sans coder
    - A coder
        - Poids (CRUD Poids)
        - Poidsv (CRUD Poids-Vignette)
        - Adresse (CRUD adresse)
        - Client (CRUD Client)
        - Commune (CRUD Commune)
        - Conditionnement (CRUD Conditionnement)
        - Objet (CRUD Objet)
        - Role (CRUD Role)
        - Utilisateur (CRUD Utilisateur)
    - Les endpoints (à noter , conforme aux conventions TP7, sous /api/)
        - Poids (t_poids) :
            - GET /api/poids
            - POST /api/poids
            - GET /api/poids/{id}
            - PUT ou PATCH /api/poids/{id}
            - DELETE /api/poids/{id}
        - Poids-Vignettes (t_poidsv) :
            - GET /api/poids-vignettes
            - POST /api/poids-vignettes
            - GET /api/poids-vignettes/{id}
            - PUT ou pATCH /api/poids-vignette/{id}
            - DELETE /api/poids-vignettes/{id}
        - NB : les autres endpoints se trouvent dans le fichier src/routers/
    - Les fichiers à créer
        - src/models/poids.py
        - src/models/poids_v.py
        - src/repositories/poids_repository.py
        - src/repositories/poidsv_repository.py
        - src/services/poids_service.py
        - src/services/poidsv_service.py
        - src/routes/poids_route.py
        - src/routes/poidsv_route.py
        - tests/test_poids.py
        - tests/test_poidsv.py
        - NB : Les autres fichiers (adresse, client, commune, conditionnement, objet, role, utilisateur) sont déjà créées)
- Fin du J1 –

        - Vérification :
            - Git
                - git.status
                - git branch (devoir etre sur dev/imen et les modifications effectuées)
            - API
                - Swagger s'ouvre (http://127.0.0.1:8000/docs)
                - L'API ne crash pas au lancement
            - DB
                - MySQL tourne
                - la base digisheese existe
        - Commit/ push (si utile)
            - git add .
            - git commit -m "J1 : setup environnement et prise en main)

- Jour 2 – Modèles + persistances
    - Objectif du J2
        - Le but du Jour 2 est de **poser une base de données fiable et exploitable**, parfaitement alignée avec le cahier des charges DIGICHEESE, afin de préparer l'implémentation des CRUD API du Jour 3.
        - À la fin du J2 :
            - Les modèles métiers sont définis et stables
            - La base MySQL est opérationnelle
            - Les tables Admin obligatoires existent
            - La connexion entre l'API et MySQL fonctionne
            - Les opérations de persistance (lecture / écriture) sont validées au niveau base
        - Aucun endpoint API n'est encore exposé : le J2 est exclusivement centré sur la modélisation et la persistance.
    - Périmètre fonctionnel J2
        - Le cahier des charges impose obligatoirement, côté **Admin**, la gestion des entités suivantes :
            - Poids
            - Poids-Vignettes
            - Commune
            - Objet (Article)
            - Conditionnement
        - En complément, pour assurer la cohérence globale du système, les entités suivantes sont également prises en compte :
            - Adresse
            - Client (Optionnel - Opérateur Colis)
            - Role
            - Utilisateur
        - Toutes ces entités sont modélisées afin de garantir une base évolutive et conforme aux besoins futurs.
    - Etape 1 – Analyse des modèles existants
        - Avant toute création, analyse du fichier **Exemples_models.py** afin de :

        - Comprendre les noms exacts des tables attendues (t_poids, t_poidsv, t_communes, t_objet, t_conditionnement, etc.)
        - Identifier les champs obligatoires
        - Repérer les types (Integer, String, Decimal, Boolean)
        - Vérifier les clés primaires, clés étrangères et index
    - Cette analyse permet d'assurer une **compatibilité stricte avec le CDC** et d'éviter toute divergence de schéma.
- Etape 2 – Création des modèles ORM
    - Dans le dossier src/models/, création ou validation des modèles suivants :
        - Modèles Admin obligatoires
            - poids.py → table t_poids
            - poids_v.py → table t_poidsv
            - commune.py → table t_communes
            - objet.py → table t_objet
            - conditionnement.py → table t_conditionnement
        - Modèles complémentaires
            - adresse.py
            - client.py
            - role.py
            - utilisateur.py
            - table d'association utilisayeur/ role
        - Chaque modèle respecte
            - Un nom de table explicite
            - Une clé primaire unique
            - Des types cohérents avec le métier
            - Des champs nullable uniquement lorsque justifié
            - Des index sur les champs fréquemment recherchés
        - Les champs financiers et de poids utilisent des **types décimaux** afin d'éviter toute erreur d'arrondi.
- Etape 3 – Définition des relations entre tables
    - Les relations nécessaires sont définies pour assurer l'intégrité des données :
        - Une **commune** peut être liée à plusieurs clients
        - Un **client** est rattaché à une commune
        - Un **utilisateur** peut avoir un ou plusieurs rôles
        - Les objets et conditionnements sont indépendants mais exploitables ensemble dans les flux métiers
    - Les clés étrangères sont déclarées explicitement dans les modèles afin que MySQL garantisse la cohérence référentielle.
- Etape 4 – Mise en place de la persistance MYSQL
    - Dans src/conf/db.py :
        - Configuration de l'engine MySQL
        - Chargement des variables d'environnement depuis .env
        - Création de la session de base de données
        - Vérification de la connexion au démarrage de l'application
    - Les paramètres utilisés :
        - Host : localhost

        - Port : 3306
        - Base : digicheese
        - Utilisateur dédié à l'application (pas root en production)
        - Encodage : utf8mb4
- Etape 5 – Création de la base de données
    - Voir src/utils/database.py
        - On a :
            - Tables créées sans erreur
            - Clés primaires et étrangères fonctionnelles
            - Index correctement appliqués
        - La structure finale de la base correspond strictement aux modèles définis dans l'API.
- Etape 6 – Vérification de la persistance
    - Avant de passer à l'API, validation manuelle de la base :
    - Pour chaque table Admin :
        - Insertion d'un enregistrement test
        - Lecture de l'enregistrement
        - Mise à jour d'un champ
        - Suppression de l'enregistrement
    - Ces tests confirment que :
        - La base accepte les données
        - Les contraintes sont respectées
        - Les données sont bien persistées
- Etape 7 – Sécurité et bonnes pratiques
    - Dès le J2, les règles suivantes sont appliquées :
        - Aucun mot de passe ou secret n'est versionné
        - Le fichier .env reste local et ignoré par Git
        - Les accès MySQL sont limités aux droits nécessaires
        - Les modèles sont conçus pour être compatibles RGPD (pas de données inutiles)
- Vérifications J2
    - Avant validation du J2
        - Base de données
            - MySQL démarre sans erreur
            - La base digicheese existe
            - Toutes les tables nécessaires sont présentes
        - Code
            - Les modèles sont propres et cohérents
            - Aucun import cassé
            - L'API démarre sans crash
        - Git
            - git status
            - git branch
                - Branche active dev/imen
                - Les fichiers modèles et DB sont bien versionnés
    - **Commit de fin de J2**
        - git add .

        - git commit -m “J2 – modèles métiers et persistance MYSQL”
        - git push origin imen
    - Résultat final J2
        - Modèles métiers finalisés
        - Base MySQL fonctionnelle
        - Persistance validée
        - Fondations solides pour le CRUD API du Jour 3

- Jour 3 – Services + routes (API CRUD complète)
    - Objectif du J3
        - Le but du Jour 3 est d’**exposer une API REST complète et fonctionnelle**, basée sur les modèles et la persistance validés au Jour 2.
        - A la fin du J3 :
            - Tous les CRUD Admin obligatoires sont accessibles via l’API
            - Les routes FastAPI sont opérationnelles
            - Les services métiers et repositories sont en place
            - La validation des données et la gestion des erreurs sont appliquées
            - Swagger permet de simuler un front et de tester l’ensemble des endpoints
    - Périmètre fonctionnel J3
        - Les fonctionnalités implémentées au Jour 3 couvrent les entités suivantes :
        - CRUD Admin obligatoires
            - Poids
            - Poids-Vignettes
            - Commune
            - Objet
            - Conditionnement
        - CRUD complémentaires
        - Adresse
        - Client
        - Role
        - Utilisateur
            - Toutes les routes respectent les conventions REST et sont exposées sous le préfixe /api
    - Etape 1 – Mise en place des repositories
        - Dans le dossier src/repositories/, création ou finalisation des repositories :
            - poids_repository.py
            - poidsv_repository.py
            - commune_repository.py
            - objet_repository.py
            - conditionnement_repository.py
            - adresse_repository.py
            - client_repository.py
            - role_repository.py
            - utilisateur_repository.py
        - Chaque repository :

        - Gère exclusivement l’accès à la base de données
        - Implémente les opérations CRUD :
            - create
            - get_all
            - get_by_id
            - update
            - delete
        - Utilise la session MySQL définie dans conf/db.py
        - Ne contient aucune logique métier
- Etape 2 – Implémentation des services métiers
    - Dans le dossier src/services/, création des services correspondants :
        - poids_service.py
        - poidsv_service.py
        - commune_service.py
        - objet_service.py
        - conditionnement_service.py
        - adresse_service.py
        - client_service.py
        - role_service.py
        - utilisateur_service.py
    - Chaque service :
        - Appelle le repository associé
        - Implémente la logique métier
        - Gère les règles de validation
        - Centralise la gestion des erreurs
    - Exemples de règles appliquées :
        - Refus de création si un champ obligatoire est manquant
        - Retour d’une erreur 404 si l’identifiant n’existe pas
        - Interdiction de mise à jour partielle incohérente
        - Messages d’erreur explicites et compréhensibles
- Etape 3 – Gestion des erreurs et codes HTTP
    - Une stratégie uniforme est mise en place :
        - 200 OK : lecture réussie
        - 201 Created : création réussie
        - 204 No Content : suppression réussie
        - 400 Bad Request : données invalides
        - 404 Not Found : ressource inexistante
        - 422 Unprocessable Entity : erreur de validation automatique FastAPI
    - Les erreurs sont remontées depuis les services vers les routes sous forme d’exceptions HTTP contrôlées.
- Etape 4 – Création des routes FastAPI
    - Dans le dossier src/routes/, création ou validation des routers :
        - poids_route.py
        - poidsv_route.py
        - commune_route.py
        - objet_route.py

        - conditionnement_route.py
        - adresse_route.py
        - client_route.py
        - role_route.py
        - utilisateur_route.py
    - Chaque router :
        - Déclare un APIRouter
        - Définit un préfixe explicite (/api//ressource)
        - Associe les méthodes HTTP aux opérations CRUD
        - Injecte les services nécessaires
- Etape 5 – Détail des endpoints exposés
    - Exemple : Poids
        - GET /api/poids
        - POST /api/poids
        - GET /api/poids/{id}
        - PUT /api/poids/{id}
        - DELETE /api/poids/{id}
    - Exemple : Poids-Vignettes
        - GET /api/poids-vignettes
        - POST /api/poids-vignettes
        - GET /api/poids-vignettes/{id}
        - PUT /api/poids-vignettes/{id}
        - DELETE /api/poids-vignettes/{id}
- Etape 6 – Schémas Swagger et validation
    - Pour chaque endpoint :
        - Définition de schémas d'entrée (Create / Update)
        - Définition de schémas de sortie (Response)
        - Ajout de descriptions claires
        - Validation automatique des champs via Pydantic
    - Swagger (/docs) permet :
        - De visualiser toutes les routes
        - De connaître les champs attendus
        - De tester chaque opération sans Postman
- Etape 7 – Tests manuels via Swagger
    - Avant de passer aux tests automatisés :
        - Création d'objets via POST
        - Vérification via GET
        - Modification via PUT/ PATCH
        - Suppression via DELETE
        - Test des erreurs (ID inexistant, données invalides)
    - Chaque ressource est validée fonctionnellement via Swagger.
- Vérifications finales J3
    - Avant validation du J3
        - API
            - L'API démarre sans erreur
            - Swagger s'ouvre correctement

        - Tous les endpoints CRUD sont visibles
    - Fonctionnel
        - Les opérations CRUD fonctionnent
        - Les erreurs sont propres et cohérentes
        - Les données sont bien persistées en base
    - Git
        - git status
        - git branch
            - Branche active : dev/imen
            - Les routes, services et repositories sont versionnés
    - Commit de fin de J3
        - git add .
        - git commit -m “J3 : Implémentation CRUD API et routes FastAPI”
        - git push origin imen
- Résultat dinal de J3
    - API CRUD complète
    - Routes FastAPI fonctionnelles
    - Swagger opérationnel
    - Base exploitée via l’API
    - Projet prêt pour la phase de tests du Jour 4

- Jour 4 – Tests automatisés + stabilisation (DOD)
    - Objectif du J4
        - Le but du Jour 4 est de **sécuriser l’ensemble du travail réalisé**, conformément à la **Definition of Done (DOD)** du projet.
        - À la fin du J4 :
            - Des tests automatisés sont en place
            - Chaque ressource CRUD est couverte par des tests
            - Les cas nominaux et les cas d’erreur sont validés
            - Les réponses HTTP sont harmonisées
            - L’API est stable, robuste et sans régression
    - Périmètre de tests J4
        - Les tests portent sur toutes les ressources exposées au Jour 3.
            - Ressources Admin obligatoires
                - Poids
                - Poids-Vignettes
                - Commune
                - Objet
                - Conditionnement
            - Ressources complémentaires
                - Adresse
                - Client
                - Role
                - Utilisateur
            - Chaque ressource est testée de manière indépendante.

- Etape 1 – Mise en place de l'environnement de tests
  - Dans le dossier tests/ :
    - Configuration de Pytest
    - Utilisation du client de test FastAPI
    - Connexion à une base de test dédiée ou à une base isolée
    - Nettoyage des données entre les tests afin de garantir l'indépendance
  - Les tests sont conçus pour être exécutables via :
    - pytest
- Etape 2 – Structure des fichiers de tests
  - Pour chaque ressource, un fichier de test dédié est créé :
    - test_poids.py
    - test_poidsv.py
    - test_commune.py
    - test_objet.py
    - test_conditionnement.py
    - test_adresse.py
    - test_client.py
    - test_role.py
    - test_utilisateur.py
  - Chaque fichier est organisé de manière lisible et cohérente.
- Etape 3 – Tests nominaux (cas de succès)
  - Pour chaque ressource CRUD, un test nominal est implémenté :
    - Création d'une ressource via POST
    - Vérification du code HTTP (201)
    - Lecture de la ressource via GET
    - Vérification des données retournées
    - Mise à jour via PUT/ PATCH
    - Suppression via DELETE
  - Ces tests prouvent que :
    - L'API fonctionne correctement
    - Les données sont bien persistées
    - Les endpoints respectent les conventions REST
- Etape 4 – Tests d'erreur (cas d'échec)
  - Pour chaque ressource, au moins un test d'erreur est implémenté :
    - Exemples de scénarios testés :
      - Création avec champ obligatoire manquant
      - Requête avec identifiant inexistant
      - Mise à jour d'une ressource inexistante
      - Suppression d'un élément déjà supprimé
    - Les tests vérifient :
      - Le code HTTP retourné (400, 404 ou 422)
      - La cohérence des messages d'erreur
      - L'absence de crash de l'API
- Etape 5 – Harmonisation des réponses API
  - Suite à l'exécution des tests :
    - Uniformisation des messages d'erreur

        - Harmonisation des codes HTTP
        - Correction des incohérences entre routes
        - Vérification des schémas de réponse Swagger
    - Cette étape permet d'avoir une API homogène et prévisible.
- Etape 6 – Stabilisation du code
    - Corrections réalisées pendant le J4 :
        - Bugs détectés par les tests
        - Problèmes de validation des données
        - Cas limites non couverts initialement
        - Nettoyage du code inutile ou redondant
    - Aucune nouvelle fonctionnalité n'est ajoutée : le J4 est dédié uniquement à la **qualité et à la stabilité**.
- Etape 7 – Vérification globale du projet
    - Avant validation finale :
        - Tests (pytest)
            - Tous les tests passent
            - Aucun warning bloquant
        - API
            - L'API démarre sans erreur
            - Swagger fonctionne toujours
            - Les endpoints restent accessibles
        - Base de données
            - Aucune corruption de données
            - Les tests n'impactent pas la base de développement
- Vérifications finales J4
    - Git
        - git status
        - git branch
            - Branche active : dev/imen
            - Tous les fichiers de tests sont versionnés
    - Commit de fin de J4
        - git add .
        - git commit -m "J4 : tests automatisés et stabilisation de l'API"
        - git push origin imen
- Résultat final de J4
    - Tests automatisés en place
    - Cas nominaux et cas d'erreur validés
    - API robuste et stable
    - Projet prêt pour la phase finale de documentation et livraison (Jour 5)

- Jour 5 – Documentation, livraison et préparation du rendu final
    - Objectif du J5
        - Le but du Jour 5 est de **finaliser le projet DIGICHEESE** afin de livrer un dépôt Git **propre, cohérent et exploitable**, conforme à l'ensemble des exigences du cahier des charges.

- À la fin du J5 :
  - La documentation est complète et structurée
  - Le projet est prêt à être livré et évalué
  - Le dépôt Git est propre et cohérent
  - L'ensemble du travail est relisible, compréhensible et maintenable
- Périmètre du J5
  - Le Jour 5 ne comporte **aucun développement fonctionnel supplémentaire**.
    Il est exclusivement dédié à :
    - La documentation
    - La relecture du code
    - La préparation du livrable final
    - La validation globale du projet
- Étape 1 – Organisation de la documentation
  - Dans le dossier DOC/, structuration de la documentation en plusieurs parties :
    - Documentation fonctionnelle
    - Documentation technique
    - Guide d'utilisation de l'API
    - Scénarios de tests
    - Éléments qualité, sécurité et monitoring
  - Un fichier README.md principal est ajouté à la racine du dépôt.
- Étape 2 – Rédaction du README.md
  - Le fichier README.md présente :
    - Le contexte du projet DIGICHEESE
    - Les objectifs de l'API
    - La stack technique utilisée
    - La structure globale du projet
    - Les rôles applicatifs (Admin, Opérateur Colis, Opérateur Stock)
    - La procédure d'installation et de lancement
    - L'accès à Swagger (/docs)
    - Les commandes principales (API, tests, Docker)
  - Le README permet à un tiers de comprendre et lancer le projet sans assistance.
- Étape 3 – Documentation fonctionnelle
  - Rédaction d'une documentation décrivant :
    - Le périmètre fonctionnel couvert
    - Les entités gérées par l'API
    - Les opérations CRUD disponibles
    - Les rôles applicatifs et leurs responsabilités
    - Les limites connues et choix fonctionnels
  - Cette documentation permet de comprendre **ce que fait l'application**, indépendamment de la technique.
- Étape 4 – Documentation technique
  - Rédaction d'une documentation détaillant :
    - L'architecture du projet
    - La structure des dossiers (models, repositories, services, routes)
    - Le fonctionnement de la persistance MySQL
    - Les principes des services métiers

        - La gestion des erreurs
        - Les conventions de nommage
        - Les choix techniques réalisés
    - Cette partie est destinée à un développeur ou un mainteneur.
- Étape 5 – Documentation d'utilisation de l'API
    - Création d'un guide expliquant :
        - Comment accéder à Swagger
        - Comment tester les endpoints via Swagger
        - Les principales routes /api/v1
        - Des exemples de requêtes et réponses
        - Les codes d'erreur possibles
    - Swagger est présenté comme l'outil principal de simulation du front.
- Étape 6 – Scénarios de tests et résultats
    - Dans un dossier dédié :
        - Description des scénarios de tests
        - Cas nominaux et cas d'erreur
        - Résultats attendus
        - Résultats observés
        - Conclusion sur la stabilité de l'API
    - Cette section permet de démontrer la conformité à la DOD.
- Étape 7 – Qualité, sécurité et monitoring
    - Ajout d'une section spécifique décrivant :
        - Qualité
            - Respect des conventions
            - Couverture de tests
            - Lisibilité et maintenabilité du code
        - Sécurité
            - Gestion des accès
            - Protection des données sensibles
            - Principes RGPD
            - Bonnes pratiques OWASP (validation des entrées, gestion des erreurs)
        - Monitoring (à faire)
            - Endpoint /health retournant OK
            - Endpoint /version
            - KPIs définis :
                - Temps de réponse
                - Taux d'erreur
                - Disponibilité de l'API
- Étape 8 – Relecture finale du code
    - Avant livraison :
        - Suppression du code mort
        - Harmonisation des noms
        - Vérification des imports
        - Vérification du formatage
        - Vérification des messages d'erreur
    - Le projet est relu comme s'il devait être repris par un autre développeur.

- Étape 9 – Vérification globale finale
  - Avant livraison définitive :
    - API
      - L'API démarre sans erreur
      - Swagger est accessible
      - Tous les endpoints fonctionnent
    - Tests (pytest)
      - Tous les tests passent
    - Base de données
      - Les scripts SQL permettent de recréer la base
      - Les données de test sont cohérentes
- Étape 10 – Livraison Git finale
  - Nettoyage et préparation du dépôt :
    - Vérification des branches (master, dev, test, prod, branches développeurs)
    - Push final du travail
    - Projet prêt à être mergé vers dev
  - Commit final
    - git add .
    - git commit -m "J5 : documentation complète et préparation du livrable final"
    - git push
- Résultat final J5
  - Projet DIGICHEESE complet
  - API fonctionnelle et testée
  - Documentation claire et exploitable
  - Dépôt Git propre et conforme
  - Livrable prêt pour évaluation

- Conclusion générale du projet DIGICHEESE
  - Ce projet a abouti à la réalisation d'une **API backend FastAPI complète et fonctionnelle**, conforme au cahier des charges DIGICHEESE.
  - L'architecture mise en place, la persistance MySQL, les CRUD Admin, les tests automatisés et la documentation garantissent une solution **robuste, maintenable et évolutive**.
  - Le projet respecte les exigences de **qualité, sécurité et bonnes pratiques**, et constitue une base solide pour des évolutions futures et un déploiement en conditions réelles.

Axe	Résultat
Architecture	API REST FastAPI structurée (models / repositories / services / routes)
Base de données	MySQL opérationnelle, schéma cohérent, persistance validée
Fonctionnalités	CRUD Admin complets + entités complémentaires
Qualité	Validation des données, erreurs harmonisées, code lisible
Tests	Tests automatisés (cas nominaux + erreurs)
Documentation	Fonctionnelle, technique, utilisation, tests, sécurité
Outils	Swagger, Pytest, Git, Docker
Swagger, Pytest, Git, Docker	Dépôt Git propre, branches conformes, projet prêt à être déployé

Rédigé par : Imen KHAMMASSI