Projet Inter Modules

SID31 (DataOps & Git) — SID32 (Django & PostgreSQL) — SID34 (ML & Prédiction)

Année universitaire 2025–2026

1 Contexte et objectif général

Vous devez réaliser **individuellement** une application web **Django** permettant d'afficher :

— les **devises** `USD, EUR, CNY, BTC` converties en **MRU** ;
— les **matières premières** `GOLD, IRON, COPPER` en **MRU**.

L'application doit présenter le **dernier prix disponible** et un **historique d'au moins 2 ans**, avec des **graphiques** d'évolution et des vues de **comparaison** (devises vs matières premières).

Le projet est **à rendre sous 5 semaines** à compter de la date de publication de l'énoncé.

Modules concernés et logique d'évaluation

— **SID31** : qualité DataOps (Docker/Compose, scripts robustes, logs), hygiène Git, documentation et reproductibilité. *(Aligné avec les attendus du module SID31 : pipelines robustes, Docker/Compose, collaboration Git).*
— **SID32** : application Django + PostgreSQL (modèles, vues, templates, admin, auth, qualité d'implémentation).
— **SID34** : prédiction de prix sur un horizon court (semaine/mois) via un **modèle ML pré-entraîné** et une intégration propre côté application.

2 Périmètre fonctionnel (Front-office)

2.1 Pages obligatoires

Votre application doit proposer, au minimum, les pages suivantes :

1. **Accueil** : tableau récapitulatif avec le **dernier prix** de chaque actif (devises + matières premières) et la **variation** (ex. J-1, ou J-7 si J-1 indisponible).
2. **Détail d'un actif** : série temporelle sur 2 ans minimum + graphique (courbe), filtres de dates.
3. **Comparaison par catégorie** :
   — comparaison des **devises** entre elles (USD/EUR/CNY/BTC),
   — comparaison des **matières premières** entre elles (GOLD/IRON/COPPER),
   — comparaison **devises vs matières premières** (au moins via 2 graphiques séparés ou un choix de groupe).
4. **Prédiction** : vue permettant de choisir un actif et un horizon (**7 jours** et **30 jours**), puis d'afficher :
   — les valeurs prédites,
   — un graphique superposant **historique récent** + **prédictions**.

2.2 Contraintes UX/UI

— Interface claire, responsive (Bootstrap ou équivalent).
— Graphiques via une bibliothèque web (ex. Chart.js, Plotly.js, ECharts, etc.).
— Tous les montants affichés doivent indiquer l'unité (MRU) et la date de référence du prix.

3 Back-office (Administration et sécurité)

3.1 Authentification

— Une interface d'administration est **obligatoire** et doit être protégée par l'authentification Django (`django.contrib.auth`).
— Aucun accès d'administration ne doit être possible sans connexion.

3.2 Fonctionnalités admin minimales

Depuis l'admin Django (ou une interface staff dédiée), un enseignant doit pouvoir :
— consulter et filtrer les prix par actif et par date ;
— lancer manuellement (au moins en développement) :
  — un **scraping / import** des données,
  — une **synchronisation MongoDB**.

4 Architecture technique imposée

Le projet doit tourner via **Docker Compose** (un simple `docker compose up -d` doit permettre de démarrer l'environnement).

4.1 Services minimaux dans docker-compose

— **web** : application Django ;
— **db** : PostgreSQL ;
— **scraper** : service d'acquisition quotidienne (scraping / API publique), **robuste aux erreurs** ;
— **mongo** : base MongoDB ;
— **sync** : service qui lit PostgreSQL et synchronise périodiquement MongoDB.

4.2 Job d'acquisition quotidienne (scraper)

Le service **scraper** doit :
— récupérer quotidiennement les données et mettre à jour PostgreSQL ;
— gérer les erreurs (timeouts, données manquantes, format inattendu) :
  — retries (au moins 3 tentatives),
  — logs structurés,
  — pas d'arrêt silencieux,
  — codes de retour explicites.
— **ne pas dupliquer** les données (gestion d'idempotence : contrainte unique par {actif, date}, upsert, etc.).

Bonus (orchestration) Un **bonus** sera accordé si vous réalisez une **seconde implémentation** du job d'acquisition via un outil d'orchestration (ex. **Prefect**) : planification, retries, observabilité.

4.3 Synchronisation PostgreSQL → MongoDB (sync)

Le service **sync** doit :
— extraire les données depuis PostgreSQL (ex. sur une fenêtre glissante ou sur la totalité),
— maintenir une collection MongoDB (ex. `prices`) cohérente et dédoublonnée,
— fonctionner périodiquement (ex. toutes les 6h ou 24h),
— produire des logs expliquant le volume synchronisé et les anomalies.

5 Données, modèle relationnel et règles de qualité

5.1 Actifs

Vous devez modéliser au minimum :

— un référentiel **Asset** (actif) : code, libellé, catégorie (*currency / commodity*) ;
— des **Price** (prix) : date, valeur en MRU, source, métadonnées (optionnel : taux, unité, etc.).

5.2 Historique 2 ans minimum

— L'application doit contenir **au moins 24 mois** d'historique pour **tous** les actifs.
— Vous devez décrire dans le `README` :
  — vos sources,
  — leur fréquence,
  — vos hypothèses (jours sans cotation, valeurs manquantes, etc.).

5.3 Règles de validation

— Prix > 0 (sauf cas justifié).
— Date valide, non future (sauf prédiction, gérée séparément).
— Unicité sur {asset, date} en PostgreSQL.

6 Prédiction ML (SID34)

6.1 Exigences

Vous devez fournir une fonctionnalité de prédiction basée sur un **modèle pré-entraîné** (entraîné par vous, mais livré prêt à l'emploi dans le repo) :

— horizon **7 jours** et **30 jours** ;
— prédiction **par actif** ;
— chargement du modèle et inférence de manière reproductible dans l'environnement Docker ;
— documentation : choix du modèle, features, protocole d'évaluation, métriques (MAPE/RMSE ou équivalent).

6.2 Livrables ML

— un artefact de modèle (ex. `.pkl`, `.joblib`, `.onnx`, etc.) versionné ;
— un script d'inférence (appelé par Django) ;
— un court rapport ML dans `REPORT.md` (ou section dédiée du `README`).

7 Exigences Git et qualité (SID31)

7.1 Dépôt GitHub (privé)

— Le code et la configuration doivent être poussés sur un **dépôt GitHub privé**.
— Vous devez partager le dépôt en **lecture seule** avec les enseignants (accès à fournir avant la date limite).

7.2 Hygiène Git

La **structure du projet** et le **nombre/qualité de commits** seront pris en compte :

— commits atomiques, messages clairs (verbe d'action),
— branches (au moins une fonctionnalité développée sur branche + merge),
— tag de release (ex. `v1.0.0`),

— `.gitignore` correct,
— aucune fuite de secrets (mots de passe, tokens) dans le repo.

Conseil : prévoir une CI simple (optionnel) ou au minimum des commandes de vérification dans le README.

8 Structure minimale recommandée du projet

Vous pouvez adapter, mais votre arborescence doit être lisible. Exemple :

```
project/
  docker-compose.yml
  .env.example
  README.md
  REPORT.md
  app/                  # Django project root
  services/
    scraper/            # code scraper + scheduling
    sync_mongo/         # code sync postgres -> mongo
    ml/                 # models + inference
  docs/                 # captures, schémas, notes
```

9 Livrables attendus

9.1 Obligatoires

— Dépôt GitHub privé partage en lecture seule.
— `docker-compose.yml` opérationnel.
— Base PostgreSQL avec **2 ans** d'historique pour chaque actif.
— Application Django : pages (accueil, détail, comparaisons, prédiction) + admin + auth.
— Service **scraper** quotidien avec gestion d'erreurs.
— Service **sync** PostgreSQL → MongoDB périodique.
— Documentation : `README.md` (installation, exécution, variables, sources, décisions) + `REPORT.md` (bilan, limites, améliorations).

9.2 Bonus

— Orchestration Prefect (flows, scheduling, retries, logs).
— Observabilité : healthchecks Compose, logs structurés, dashboards simples.

10 Règles de soumission

— **Deadline :** 5 semaines après diffusion de l'énoncé (date exacte communiquée par l'enseignant).
— Soumission via : lien GitHub + (optionnel) archive `.zip` du code.
— Le README doit contenir :
  — `docker compose up -d` et les commandes utiles (`migrate`, `createsuperuser`, etc.),
  — les variables d'environnement et un `.env.example`,
  — les sources de données et la stratégie de gestion des erreurs,
  — comment tester la prédiction (exemples).