Sujet de projet : Outil de gestion de budget personnel

Contexte du projet

L'objectif est de concevoir et développer une application de **gestion de budget personnel** permettant à un utilisateur de :

- enregistrer ses revenus et ses dépenses au fil du temps ;
- organiser ces mouvements par catégories (alimentation, logement, loisirs, transports, etc.) ;
- définir des budgets sur ces catégories et suivre leur niveau de consommation.

Les étudiants sont libres de choisir :

- le type d'application :
  - application en **ligne de commande (CLI)**,
  - application avec **interface graphique** simple (desktop, web, mobile) ;
- l'architecture :
  - application monolithique sans API,
  - ou application avec **API** (ex. backend REST + front séparé) ;
- le mode de stockage : fichiers, base locale (SQLite, autre SGBD), etc.

Les étudiants travaillent **en groupes de 3 à 4 étudiants**.

Fonctionnalités du MVP à implémenter

1. Saisie des transactions

L'application doit permettre à l'utilisateur de :

- ajouter une transaction avec au minimum :
  - montant ;
  - libellé (description courte) ;
  - type (revenu ou dépense) ;
  - catégorie (alimentation, loyer, etc.) ;
  - date ;

- lister les transactions enregistrées, avec au moins un filtre simple (par période ou par catégorie).

Exemples de comportements (adaptables selon l'interface) :

- En CLI : `add 25.50 "Courses Leclerc" alimentation 2026-01-06`
- En interface web : formulaire de saisie + tableau des transactions filtrable.

2. Gestion des budgets par catégorie

L'application doit permettre à l'utilisateur de :

- définir un **budget** pour une catégorie sur une période (par exemple, 300 € pour "alimentation" sur le mois de janvier) ;
- consulter, pour chaque catégorie et pour une période donnée (ex. un mois) :
    - le montant total dépensé ;
    - le budget fixé ;
    - le montant restant (ou le dépassement) ;
    - idéalement le pourcentage de budget consommé.

Ces deux blocs (saisie + budgets) constituent le **MVP obligatoire**.

Tests sur le MVP

Même pour le MVP, les étudiants doivent mettre en place des **tests automatisés** :

- tests écrits en **TDD** ou **BDD** (ou un mélange des deux selon les parties du code) ;
- au moins **80 % de couverture de tests** sur la logique métier du MVP (calculs, validations, agrégations).

Exemples de tests MVP :

- calcul correct du total dépensé par catégorie sur un mois ;
- calcul du montant restant sur un budget ;
- rejet ou gestion de transactions invalides (montant nul, catégorie inconnue, date invalide, etc.).

Travail à réaliser par les groupes

Les étudiants travaillent **en groupes de 3 à 4 étudiants** avec répartition claire des responsabilités.

Phase 1 : Conception et implémentation du MVP (code + tests)

Chaque groupe doit :

- choisir sa stack technique (ex. Python + SQLite, Node.js + PostgreSQL, Java + fichiers, etc.) ;
- concevoir le modèle de données (schéma de base, tables, fichiers ou modèles objets) ;
- implémenter les fonctionnalités de **saisie de transactions** et de **gestion des budgets par catégorie** ;
- mettre en place un **ensemble de tests automatisés** sur ce MVP :
    - tests unitaires et/ou d'intégration ;
    - démarche TDD ou BDD au moins sur une partie significative du code ;
    - couverture globale de tests sur le projet ≥ 80 % (mesurée avec l'outil de test choisi).

Un **README** doit présenter :

- comment installer/lancer l'application ;
- comment utiliser les fonctionnalités du MVP (exemples de commandes ou captures d'écran) ;
- comment exécuter les tests.

Phase 2 : Développement de fonctionnalités supplémentaires (code + tests)

À partir du MVP, chaque groupe doit ajouter des fonctionnalités supplémentaires.

Organisation conseillée :

- chaque étudiant prend en charge **1 à 2 fonctionnalités** supplémentaires ;
- le groupe vise au total **au moins 4 fonctionnalités supplémentaires**.

Pour **chaque fonctionnalité supplémentaire** :

- utiliser **au moins une approche TDD** (écrire les tests avant le code, cycle red → green → refactor) ;
- rédiger **au moins un scénario BDD** décrivant le comportement attendu de cette fonctionnalité du point de vue utilisateur ;
- implémenter la fonctionnalité en respectant le scénario BDD ;
- ajouter ou adapter les tests pour conserver une couverture globale ≥ 80 %.

Exemple de scénario BDD :

Feature: Alerte de dépassement de budget
Scenario: Dépassement du budget alimentation

User Story : Alerte de dépassement de budget

Titre : En tant qu'utilisateur, je souhaite être alerté quand une dépense fait dépasser mon budget de catégorie, afin de mieux contrôler mes finances.

Description :

L'utilisateur doit recevoir une indication claire (alerte, message, couleur, etc.) lorsqu'il ajoute une transaction qui fait franchir la limite du budget défini pour une catégorie.

Critère	Détail
AC 1	Budget alimentation janvier = 300 €
AC 2	Dépenses existantes alimentation = 290 €
AC 3	Ajout nouvelle dépense = 20 €
AC 4	Total = 290 + 20 = 310 €
AC 5	Dépassement = 310 - 300 = +10 €
AC 6	Affichage alerte utilisateur visible
AC 7	Message clair : montant et % dépassement

Cas d'usage (UX)

Avant :

Budget alimentation janvier : 300 €

Dépensé : 290 €

Restant : 10 € [96.7% consommé]

Après ajout 20 € :

Budget alimentation janvier : 300 €

Dépensé : 310 €

Restant : -10 € (DÉPASSÉ) [103.3% consommé]

⚠ ALERTE : Budget alimentation dépassé de 10 € !

Liste de fonctionnalités supplémentaires possibles

Chaque groupe choisit parmi les pistes suivantes (ou propose d'autres idées validées avec l'enseignant).

A. Gestion avancée des transactions

- Modification d'une transaction existante (montant, libellé, catégorie, date).
- Suppression de transaction avec mise à jour des totaux et budgets.
- Filtres avancés sur les transactions :
  - par type (revenus / dépenses) ;
  - par plage de dates ;
  - par catégorie.

B. Budgets et logique métier

- **Report** : report d'un excédent ou d'un dépassement d'un mois sur le mois suivant.
- Notifications ou indicateurs de **proximité/dépassement** de budget (par exemple, alerte à partir de 80 % consommés).
- Système d'**enveloppes** : un revenu mensuel réparti automatiquement entre plusieurs catégories de budget selon des pourcentages définis.

C. Export et persistance

- Export des transactions et/ou des budgets au format CSV ou JSON.
- Sauvegarde et chargement des données (fichier, base de données, etc.).
- Fonction de **réinitialisation des données** (raz de la base/test).

D. Interface et expérience utilisateur

- Interface graphique simple (web, desktop, TUI) pour compléter ou remplacer un CLI.
- Tableaux lisibles et éventuelles visualisations simples (barres, camemberts, etc.).

L'ajout d'une interface graphique ou d'une API REST est optionnel mais valorisé s'il est bien conçu et reste cohérent avec la structure du projet et les tests.

Critères d'évaluation

L'évaluation tiendra compte des points suivants :

Critère	Points /20
MVP fonctionnel	4 pts
Tests sur le MVP (≥ 80% coverage)	4 pts
Fonctionnalités supplémentaires TDD/BDD	6 pts
Qualité du code / Gestion du repository	3 pts
Documentation (README + scénarios BDD)	3 pts
Bonus (GUI/API/CI)	A définir
TOTAL	**20/20**

Détail des critères

- **MVP fonctionnel** :
    - saisie de transactions complète et robuste ;
    - gestion des budgets (création, calcul des totaux, montants restants).
- **Tests sur le MVP** :
    - présence de tests automatisés ;
    - pertinence des cas de test (calculs, validations, scénarios usuels et limites) ;
    - couverture globale de tests ≥ 80 %.
- **Fonctionnalités supplémentaires** :
    - nombre de features implémentées par groupe (minimum 4) ;
    - difficulté et intérêt des fonctionnalités ;
    - bonne utilisation de TDD et BDD sur ces features (tests écrits en amont, scénarios BDD documentés).
- **Qualité du code** :
    - organisation du projet (modules, couches) ;
    - lisibilité, nommage, commentaires pertinents ;
    - séparation claire entre logique métier et interface (CLI/GUI/API).
    - Repository git. Bien nommé, commit clairs et régulier
- **Qualité de la documentation** :
    - README complet (installation, usage, tests) ;
    - scénarios BDD lisibles ;

    - explication des choix techniques et d'architecture.

Livraison

- **Repo GitHub** : Code source complet et bien organisé.
- **Branches** : `feature-nom` pour chaque fonctionnalité supplémentaire.
- **Historique** : commits clairs et fréquents montrant la progression.
- **README** : installation, utilisation, exécution des tests, documentation des scénarios BDD.
- **Vidéo démo** : 3-5 minutes montrant le MVP et les features supplémentaires.

Durée

4 semaines - Projet complet pour portfolio/CV !

Rendu le 08/02/2026

Notes importantes

- Les étudiants partent **de zéro** : pas de code fourni.
- **Tests dès le MVP** : TDD ou BDD (ou les deux).
- **Architecture libre** : CLI, GUI, API, ou combinaison → au choix du groupe.
- **TDD obligatoire** pour chaque feature supplémentaire (tests écrits avant le code).
- **BDD** : Au moins 1 scénario par feature supplémentaire.
- **Coverage ≥ 60 ~ 80 %** : Vérifiable avec l'outil de test du langage choisi.
- **Groupe 3-4** : répartition claire des features par étudiant (1-2 features/étudiant).
- **Nom du repository** : nom-nom-nom-nom-mybudget-testing
- **Envoi du repository** : rida@lamerkanterie.fr