ECF Finale Bachelor developpeur
Annexe sur le dépôt Github

Projet Ventalis,
Documentation technique.

Sommaire

I. Réflexions initiales technologiques sur le sujet.

(Pour plus de détails, et notamment sur la justification des choix, merci de vous reporter à la copie à rendre.)
Après analyse du sujet, j'ai pu concevoir son architecture en grand blocs :

- L'application Web « centrale » avec ses trois couches : présentation, business layer et persistance (soit front, back et SGBDR/base de donnée).
- Il nous faut développer une API donnant accès au back-end/BDD de l'application « centrale » pour les clients extérieurs que sont les applications Desktop et Mobile.
- Il nous faut une technologie pour l'application Desktop.
- Et une technologie pour l'application Mobile.

Stack technique.

Appli Web MVC : Django
Partie Front : Bootstrap, CSS, JS sur les gabarits Django.
Base de données : PostgreSQL.
API : Django REST Framework.
Déploiement : Heroku.
Appli Desktop :
PyQt6.
Communique avec l'API Django REST.
Appli Mobile :
Front : React Native Expo.
Communique avec l'API Django REST.

II. Configuration de l'environnement de travail.

- Sur MacOS 12.5.1 (Monterey).
- Conception :
  - Documentation (hors code) :
    - Microsoft Word, Microsoft PowerPoint, Apple Aperçu.
    - Figma,
    - Diagrammes UML : draw.io de diagrams.net.
  - Gestion de projet : Jira avec tableau Kanban, Roadmap.
- Développement :
  - Éditeur : VSCode avec extensions.
  - Gestionnaire de version : Git et Github en dépôt distant.
  - Bac à sable : dossier / projet destiné aux tests de bouts de code.
- Tests fonctionnels manuels :
  - Navigateurs : Chrome, Firefox, Safari, Edge (sous Win10 virtuel).
  - Smartphone Android et Émulateur iOS pour l'appli Mobile.
  - OS : MacOS 12.5.1 et Windows 10 pour les exécutables appli Desktop.
- Tests unitaires et fonctionnels programmés : voir ci-après, en page 4.
- Technos : voir le Stack Technique ci-dessus.

III. Architecture.

Étape obligatoire avant de se lancer dans le codage, j'ai essayé de me réserver à la conception architecturale suffisamment de temps et ne pas me précipiter vers le codage. Et au final je n'ai pas regretté, et j'ai même éprouvé une certaine satisfaction à la réalisation de cette étape, il y avait quelque chose de rassurant dans le fait de s'être établi un modèle de structure bien défini.

Modèle conceptuel de données.

Vous pouvez retrouver le **diagramme MCD** dans le dossier `documentation/UML` du dépôt.

Ce diagramme MCD est l'exemple type du document sur lequel il faut faire plusieurs « passes » de mise au point. Il est essentiel et central au processus de conception et de développement.
Afin de pas en alourdir la lecture, j'ai opté pour un diagramme épuré, en gardant les relations de cardinalités entre entités.
J'ai choisi d'y inclure une entité « User Account » qui joue le rôle de lien central entre l'utilisateur, son panier, ses éventuelles commandes, le n° de l'employé ou vendeur attitré.
L'idée est de créer un système ouvert à l'évolutivité.

Diagramme de classes.

Vous pouvez retrouver le **diagramme de classes** dans le dossier `documentation/UML` du dépôt.

La transcription applicative du diagramme MCD.
Ce diagramme a été pour moi un fil d'Ariane tout au long de l'implémentation de l'appli Web MVC. Plusieurs fois en m'y référant, j'ai pu améliorer la structure de mon code.

Diagrammes de cas d'utilisation.

Dans un souci de lisibilité, j'ai élaboré trois diagrammes d'utilisation, un par application.
Vous pouvez retrouver les **trois diagrammes de cas d'utilisations** dans le dossier `documentation/UML/use_case/` du dépôt.

Diagrammes de séquence.

Et trois diagrammes de séquence, élaborés non par application mais par type (large) de fonctionnalité représentée.
Vous pouvez retrouver les **trois diagrammes de séquence** dans le dossier `documentation/UML/sequence/` du dépôt.

IV. Plan de tests.

(Ce § est en complément du §3.3 de la copie à rendre)
Les tests sont présents sur la partie appli Web MVC Django.

Méthodologie : sans faire du Test Driven Development pur et dur (je n'ai pas encore la pratique nécessaire), les tests unitaires et d'intégration sont développés et joués au fur et à mesure de la production de code.

On peut les jouer comme suit :

```
(env) $ python manage.py test
```

J'inspecte également le code coverage avec l'outil Coverage :

```
(env) $ coverage run manage.py test
(env) $ coverage report -m
(env) $ coverage html
```

En suivant les pratiques habituelles, je m'impose une **Definition Of Done** stipulant entre autres choses que tout le code des fonctionnalités passe les tests en local (tests unitaires et d'intégration) avant d'être placé comme ticket dans la colonne « Done » du Kanban.

L'idéal dans le cadre du CI/CD est que lorsque les fonctionnalités « Done » sont push sur le repo Github, elles repassent la série de tests grâce à un script de Github Actions. Au moment où j'écris ces lignes, les Github Actions ne sont pas encore mises en place.
(À noter qu'il existe le même système sur la plateforme de déploiement choisie, Heroku.)

V. Fichier d’exemple de transaction SQL.

Veuillez trouver le fichier dans ce repo, à l’emplacement du repo :
`documentation/SQL_transaction_example/`

La transaction est une séquence de requêtes SQL (lecture / écriture) qui finit par l'instruction COMMIT (validation) ou l'instruction ROLLBACK (annulation).

- COMMIT valide les mises à jour.
- ROLLBACK annule les requêtes ou plus précisément effectue un "retour à l'état antérieur" des éléments concernés.

La séquence de requêtes est donc soit jouée dans sa totalité, soit abandonnée dans sa totalité.

En outre, la séquence de requêtes peut être regroupée dans une procédure stockée appelable par le client du SGBD pour plus d'efficacité.
On évite le transfert de toute la série de requêtes depuis le client au SGBD.

En guise d'exemple, imaginons qu'on veuille appliquer une baisse de 40% sur le prix des produits ajoutés sur le site avant le 01/01/2022, afin d'en vider le stock, sinon de l'en réduire.
Et on voudrait que cette baisse des prix soit bien appliquée à TOUS les produits concernés, sinon à AUCUN d'entre eux. (C'est le principe de la transaction).

Dans l’exemple, j'ai utilisé le langage PL/pgSQL.
Les variables de "date limite" et de ratio sont codées en dur ; on pourrait bien sûr les passer par des paramètres si l’on construit une procédure stockée.

Cette transaction a été testée dans la BDD locale.