Yassine BELAAROUS, Clément VIGNON CHAUDEY,
Ilane RIOTTE

SAE Projet

Dossier de spécifications

Année 2025-2026

IUT'ORLÉANS
INFORMATIQUE
Université d'ORLÉANS

1. Présentation du Projet

1.1 Contexte

Lors de catastrophes naturelles, les assurés doivent fournir une liste précise des dommages et des justificatifs (factures, photos) souvent perdus dans le sinistre.

1.2 Objectif

MobiList est une application web permettant d'informatiser l'inventaire des biens mobiliers, de stocker les justificatifs de manière dématérialisée et d'automatiser le calcul de la valeur de remplacement, en appliquant une vétusté lors d'une déclaration de sinistre.

2. Spécifications Fonctionnelles

2.1 Acteurs

- **Utilisateur (Assuré) :** Gère ses logements, ses pièces et ses biens. Déclare des sinistres.
- **Assureur :** Supervise ses clients, consulte les inventaires et traite les sinistres déclarés.

2.2 Fonctionnalités principales (Cas d'utilisation)

- **Gestion du compte :** Création de compte, connexion/déconnexion et gestion du profil (nom, téléphone, mot de passe).
- **Gestion de l'inventaire :**
    - Ajouter, modifier ou supprimer un Logement (résidence principale, secondaire, etc.).
    - Ajouter, modifier ou supprimer une Pièce au sein d'un logement.
    - Ajouter un bien avec nom, prix d'achat, catégorie, date d'achat et un justificatif obligatoire (image ou PDF).
- **Sinistres :**
    - Déclarer un sinistre en sélectionnant les biens impactés.
    - Calcul automatique de la vétusté d'un bien selon sa date d'achat par rapport à la date du sinistre.
- **Édition de rapports :** Génération de rapports d'inventaire globaux ou par logement au format PDF.

3. Spécifications Techniques

3.1 Architecture et Environnement

L'application adopte une architecture conteneurisée pour garantir la portabilité et la cohérence entre les environnements de développement et de production.

- **Conteneurisation :** Utilisation de Docker.
- **Framework Web :** Flask (Python).
- **Base de Données :** PostgreSQL, déployée via un conteneur.
- **ORM :** SQLAlchemy pour assurer l'interface entre le code Python et le serveur PostgreSQL.

3.2 Configuration Docker

Le projet est structuré autour de deux services principaux définis dans le fichier docker-compose.yml :

1. **Service db (PostgreSQL) :** Utilise l'image officielle postgres:15.
    - Volume persistant pour les données : postgres_data:/var/lib/postgresql/data.
    - Configuration via variables d'environnement (POSTGRES_USER, POSTGRES_PASSWORD, POSTGRES_DB).
2. **Service web (Flask) :** Construit à partir d'un Dockerfile personnalisé.
    - Dépend du service db.
    - Connecté à PostgreSQL via une URL de base de données structurée ainsi : postgresql://user:password@db:5432/mobilist.

3.3 Modèle de Données (MCD)

La base de données PostgreSQL implémente les tables suivantes, générées via le script DDL :

- **ASSURE / ASSUREUR :** Gestion des identités et des profils.
- **LOGEMENT :** Stockage des informations immobilières liées aux assurés.
- **PIECE :** Organisation structurelle des logements.
- **BIEN :** Inventaire détaillé incluant les catégories, dates et prix d'achat.
- **SINISTRE & IMPACTE :** Historique des événements et évaluation financière des pertes.

4. Règles de Gestion et Validation

4.1 Intégrité des données

Le système impose des validations strictes via des formulaires (forms.py) :

- **Âge :** Un utilisateur doit avoir au moins 18 ans pour s'inscrire.
- **Format :** Les numéros de téléphone doivent comporter 10 chiffres.
- **Dates :** La date d'achat d'un bien ou la date d'un sinistre ne peut pas être dans le futur.
- **Justificatifs :** L'ajout d'un bien nécessite obligatoirement l'upload d'un fichier (facture/preuve d'achat) au format PDF ou image.

4.2 Calculs

- **Valeur Actuelle :** Calculée en soustrayant la vétusté au prix d'achat initial.
- **Dégâts :** Lors d'un sinistre, l'utilisateur précise si la perte est totale (100% de la valeur actuelle) ou partielle (50% par défaut dans le code).

5. Interface Utilisateur

L'application propose deux tableaux de bord distincts :

1. **Assuré :** Vue synthétique sur le nombre de logements, de biens et la valeur totale du patrimoine.
2. **Assureur :** Liste des assurés rattachés, suivi des sinistres en attente et validation des montants d'indemnisation.

6. Démarche d'Éco-conception et Optimisation

6.1 Points critiques dans le code au niveau de l’éco-conception

Une analyse du code source a permis d'identifier plusieurs domaines où nous pouvions optimisé le code :

- **Gaspillage CPU Client :** Initialement, le script de recherche searchTable.js déclenchait un recalcul de l'affichage à chaque caractère saisi, sollicitant inutilement le processeur du terminal utilisateur.
- **Impact Stockage et Bande Passante :** L'absence de redimensionnement lors de l'upload d'images entraînait un stockage de fichiers volumineux (plusieurs Mo).
- **Duplication de code :** La duplication de la logique de validation dans forms.py nuisait à la maintenabilité du code, augmentant indirectement le coût énergétique de maintenance logicielle.

6.2 Solutions Implémentées

A. Optimisation du stockage des images (Pillow)

Afin de réduire le stockage pris par les images, un traitement des images a été ajouté dans la fonction ajouter_bien :

1. **Détection intelligente :** Le système vérifie l'extension du fichier (PNG, JPG, JPEG, GIF, BMP) pour appliquer le traitement uniquement sur les images.
2. **Redimensionnement (Thumbnail) :** Utilisation de la bibliothèque **Pillow** pour redimensionner les images dans un carré de 800x800 pixels maximum tout en conservant le ratio hauteur/largeur.
3. **Résultat :** Cela réduit le poids de quelques Mo à quelques Ko sans perte de lisibilité.

B. Optimisation CPU et Maintenabilité

- **Optimisation Client :** Le script de recherche a été optimisé pour ne se déclencher qu'après une pause de 200ms dans la saisie.
- **Refactoring de forms.py :** Les fonctions de validation (nom, prénom, adresse, texte simple) ont été factorisées dans des fonctions réutilisables (check_name_format, check_address_format, etc.), ce qui réduit la redondance du code

6.3 Indicateurs de performance et suivi

Pour évaluer l'efficacité de ces mesures, nous nous sommes appuyés sur les indicateurs suivants :

- **Outils d'évaluation :** Utilisation de l'extension GreenIT-Analysis (pour le score CO2 et eau) pour valider que le score de performance garantit une consommation minimale, actuellement toutes nos pages ont un score de A.

6.4 Limites de GreenIt et améliorations possibles

Limites : GreenIt ne prend en compte que le Front-End : L'outil analyse uniquement ce qui arrive dans le navigateur (poids des fichiers, requêtes HTTP, complexité du DOM). Il ignore totalement la consommation énergétique du serveur (Back-End), comme l'efficacité des requêtes SQL ou des calculs Python.

Améliorations possibles : Utilisé CodeCarbon. C'est un outil que l'on peut intégrer directement dans le code Flask. Il permet d'estimer en temps réel la quantité de CO2 produite par l'exécution de nos fonctions en fonction de la consommation du CPU et de la localisation du serveur.