Certification
Développeuse en Intelligence Artificielle

E4 : Développer une app

Projet “Problem Management”

Octobre 2024

- Rapport rédigé par : Manon Platteau

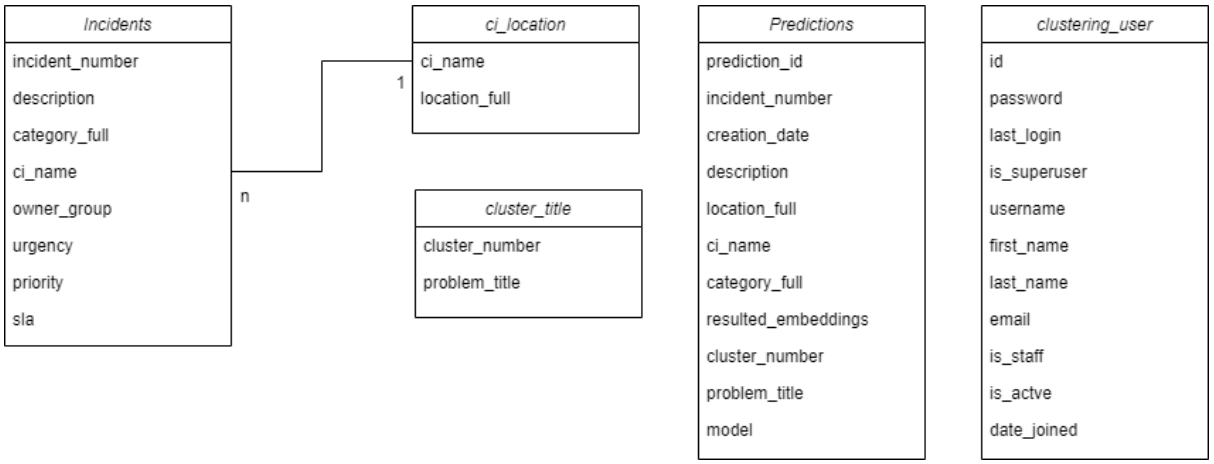
Sommaire

I. Analyser le besoin d'application d'un commanditaire intégrant un service d'intelligence artificielle

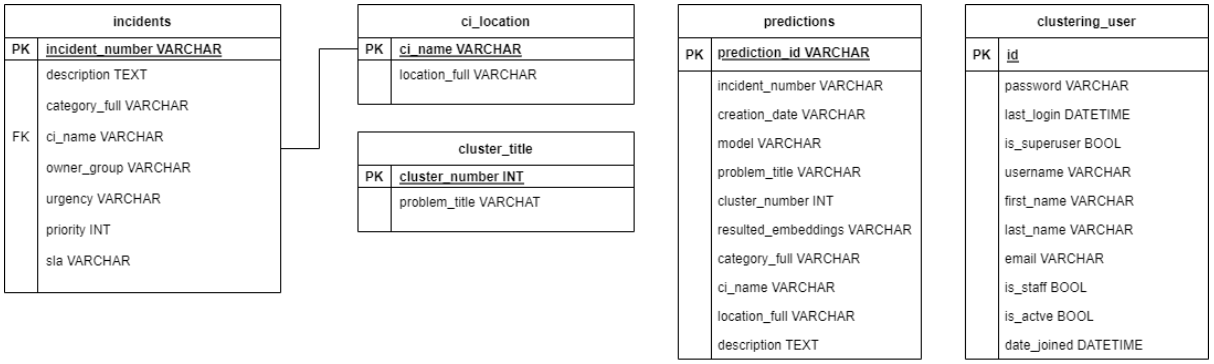
A. Modélisation des données

La base de données utilisée pour l'application est la même que celle utilisée pour la gestion des données en E1. Il s'agit d'une base de données SQL Server hébergée sur Azure.
Pour la gestion des utilisateurs de l'application, une table "clustering_user" a été ajoutée automatiquement par Django. Les noms des problèmes générés par le LLM pendant la phase d'entraînement sont stockés et récupérés dans la table "cluster_title". Les prédictions générées dans l'application sont stockées dans la table "predictions".

Modèle conceptuel des données



Modèle physique des données



B. Modélisation du parcours utilisateur et spécifications fonctionnelles

La documentation qui couvre les spécifications fonctionnelles est accessible ici : Documentation fonctionnalités .

L'objectif global de l'application est de faire gagner du temps à l'équipe qui gère les incidents informatiques. Ils réalisent actuellement manuellement le regroupement de ces incidents en problèmes (groupes d'incidents semblables) afin d'optimiser leur traitement, ce qui est chronophage.

US 1 :

- En tant qu'utilisateur, j'ai besoin d'un espace personnel sécurisé pour accéder aux résultats des clusterings.
- Critères :
  - page de connexion accessible
  - accès à l'application limité aux utilisateurs authentifiés
- Accessibilité (conformément aux standards d'accessibilité WCAG) :
  - s'assurer d'une formalisation lisible
  - ajouter des textes de remplacement aux images potentielles

US 2 :

- En tant qu'utilisateur, j'ai besoin de déclencher manuellement le processus de clustering des incidents informatiques, en fournissant un fichier csv.
- Critères :
  - un bouton et un formulaire pour lancer facilement le processus de clustering
  - une notification indique clairement que le processus est terminé
- Accessibilité (conformément aux standards d'accessibilité WCAG) :
  - s'assurer d'une formalisation lisible
  - ajouter des textes de remplacement aux images potentielles

US 3 :

- En tant qu'utilisateur, j'ai besoin de visualiser le résultat du clustering sous la forme d'un tableau.
- Critères :
  - un tableau apparaît
  - le tableau est lisible et intelligible
  - chaque ligne du tableau correspond à un incident
- Accessibilité (conformément aux standards d'accessibilité WCAG) :
  - s'assurer d'une formalisation lisible

US 4 :

- En tant qu'utilisateur, j'ai besoin d'enregistrer les résultats du clustering réalisé avec la possibilité de les consulter ultérieurement.
- Critères :
    - un onglet historique est disponible

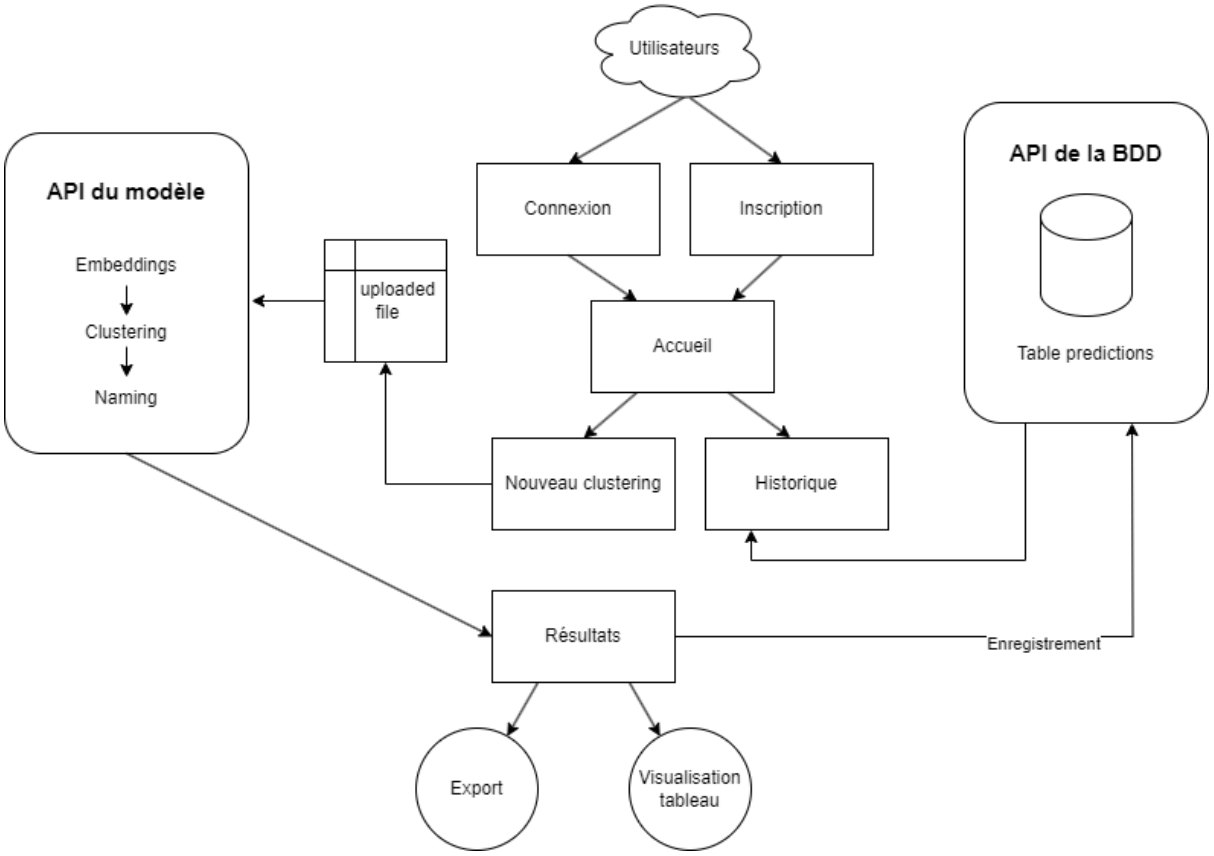
US 5 :

- En tant qu'utilisateur, j'ai besoin qu'un nom soit attribué à chaque cluster pour synthétiser son contenu.
- Critères :
    - tous les clusters ont un nom
    - le nom est pertinent au regard du contenu

US 6 :

- En tant qu'utilisateur, j'ai besoin d'exporter les résultats du clustering, y compris les noms donnés aux clusters.
- Critères :
    - une solution est apportée pour exporter les résultats en dehors de l'application et les utiliser selon les besoins de l'utilisateur

Schéma fonctionnel de l'application



II. Concevoir le cadre technique d'une application intégrant un service d'intelligence artificielle

La documentation concernant l'environnement technique de l'application est versionnée à la racine du dépôt github et disponible ici : Documentation environnement technique

L'architecture de l'application suit une logique de microservices afin de réduire la complexité et faciliter l'évolution indépendante de chacun des éléments. Ainsi, l'application comprend :

- la base de données hébergée sur Azure (contenant des données issues de l'agrégation entre fichiers pour l'entraînement du modèle, les tables intermédiaires de l'entraînement du modèle, la table de prédictions, la table des utilisateurs de l'application, la table de correspondance entre numéro de cluster et nom du problème),
- une API permettant de faire appel au modèle de clustering hébergée sur Azure Container Instances,
- une API permettant d'interagir avec la base de données hébergée sur Azure Container Instances,
- une ressource Azure OpenAI utilisée pour le naming des problèmes,
- l'application web hébergée sur Azure Container Instances.

L'application est développée en langage python avec le framework Django. Celui-ci est le plus maîtrisé par l'équipe projet. Elle est déployée sur un serveur gunicorn.

Les échanges entre l'application et les API se font via des requêtes HTTP.

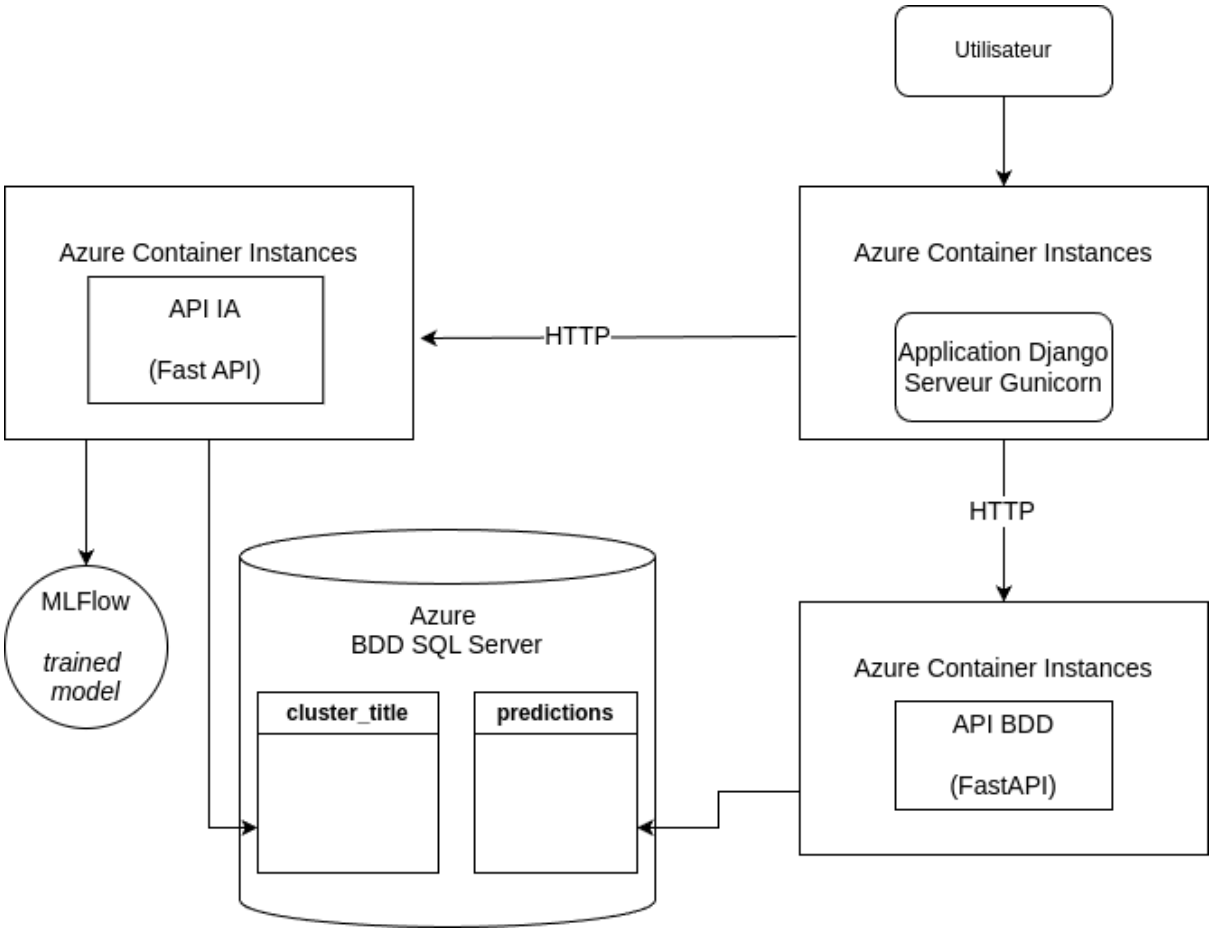
Le front utilisé dans le framework Django mobilise les langages HTML et CSS.

Les tests sont déployés grâce à la librairie pytest et à TestCase intégré directement à Django.

Le déploiement est réalisé grâce à docker et le CI/CD s'exécute avec Github actions.

Le monitoring de l'application est réalisé avec Azure Monitor et Opentelemetry.

Schéma technique de l'application

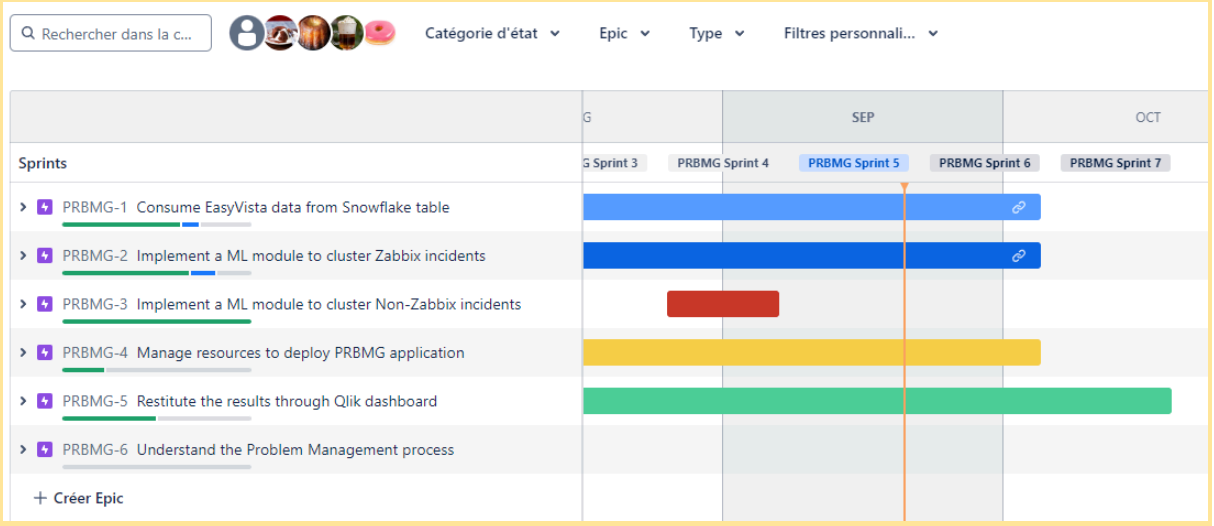


La preuve de concept de l'application est disponible en environnement de production et donne un aperçu concluant. Des améliorations pourront être apportées en termes de fonctionnalités proposées à l'utilisateur mais la conclusion est globalement positive.

III. Coordonner la réalisation technique d'une application d'intelligence artificielle en s'intégrant dans une conduite agile de projet

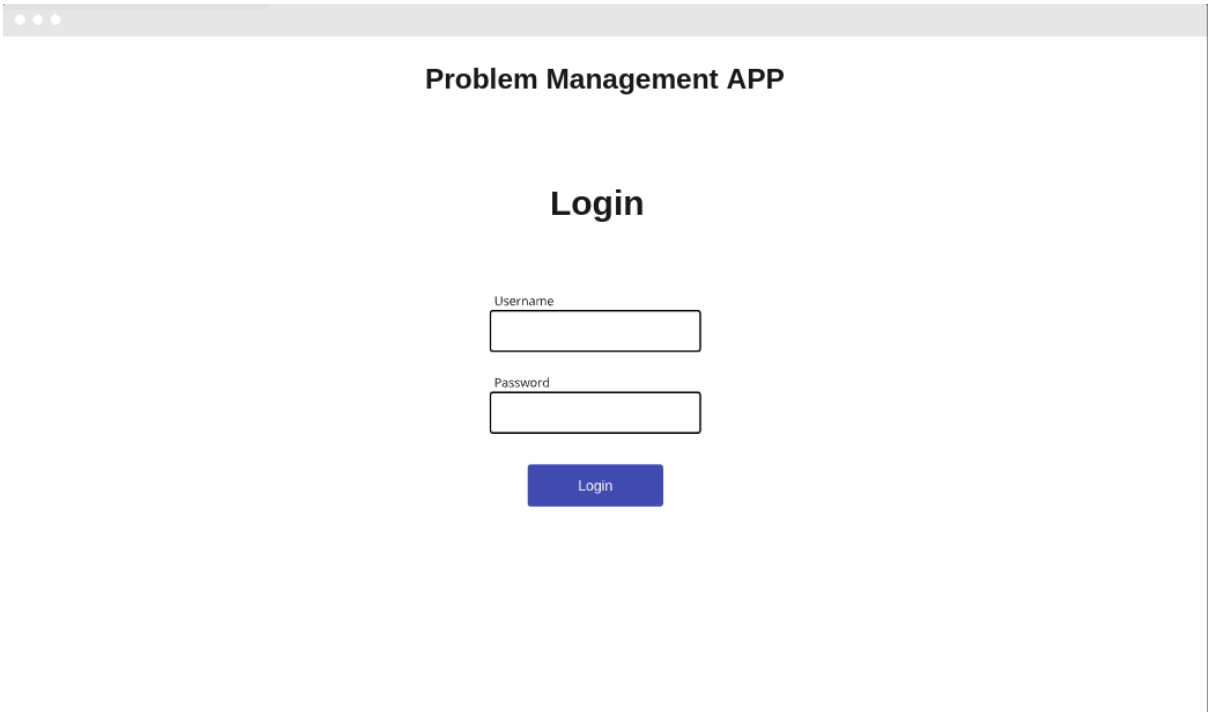
Afin d'organiser au mieux la réalisation technique de notre application, nous avons choisi de mettre en place une méthodologie agile. Cette méthode de gestion de projet, déjà appliquée dans l'entreprise, permet d'agir de façon itérative pour pouvoir s'adapter en permanence aux retours des clients, ou ici au retour du métier. Le rôle de Scrum Master est occupé par Elisa, celui de product Owner par le métier et la Dev Team par Manon. Conformément aux principes de la méthode Agile, nous travaillons en sprint (périodes de travail bien définies) dont la durée est fixée à deux semaines. Des rituels rythment ces sprints : des points d'équipe sont réalisés régulièrement (30min par semaine) entre data scientists afin de faire un point sur

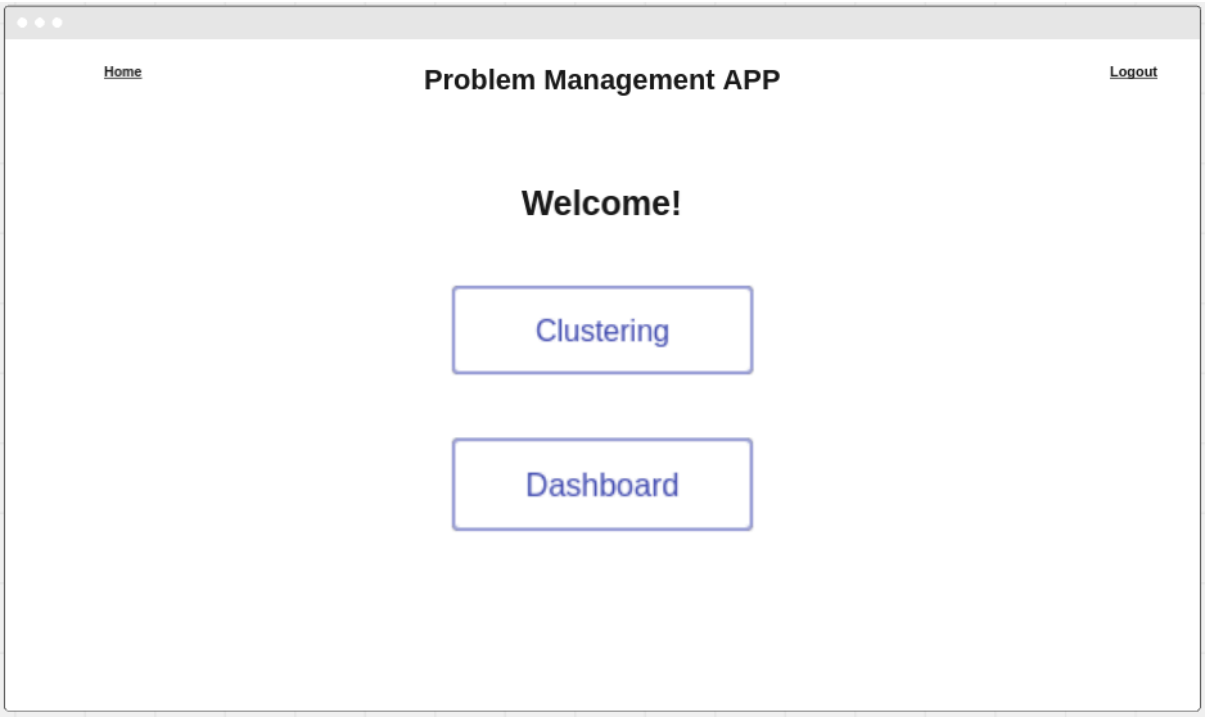
l’avancement du projet (ce qui a été fait, ce qu’il reste à faire, ce qui pose problème). Une sprint-review toutes les deux semaines permet de rendre compte de l’état d’avancement du projet au métier et de recueillir son feed back. Des project review sont également programmées afin de restituer l’avancement du projet à un niveau supérieur de la hiérarchie.
Nous utilisons l’outil de planification Jira qui nous permet de nous répartir les tâches et de planifier les sprints.

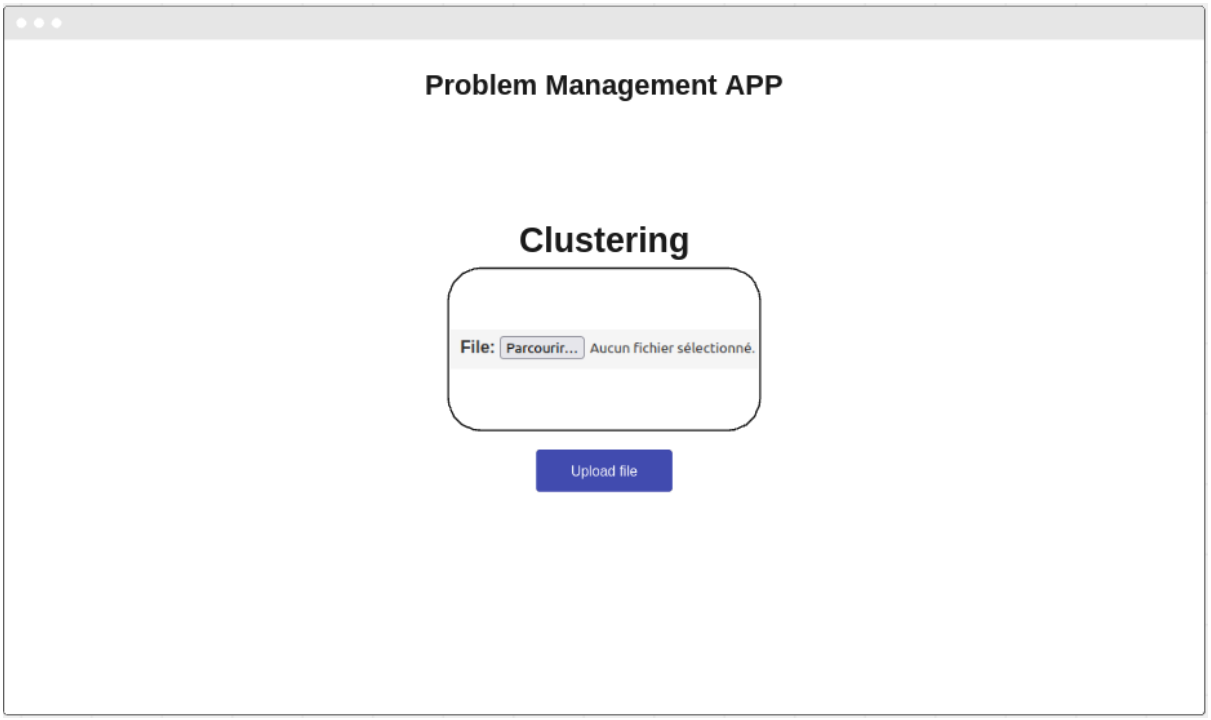


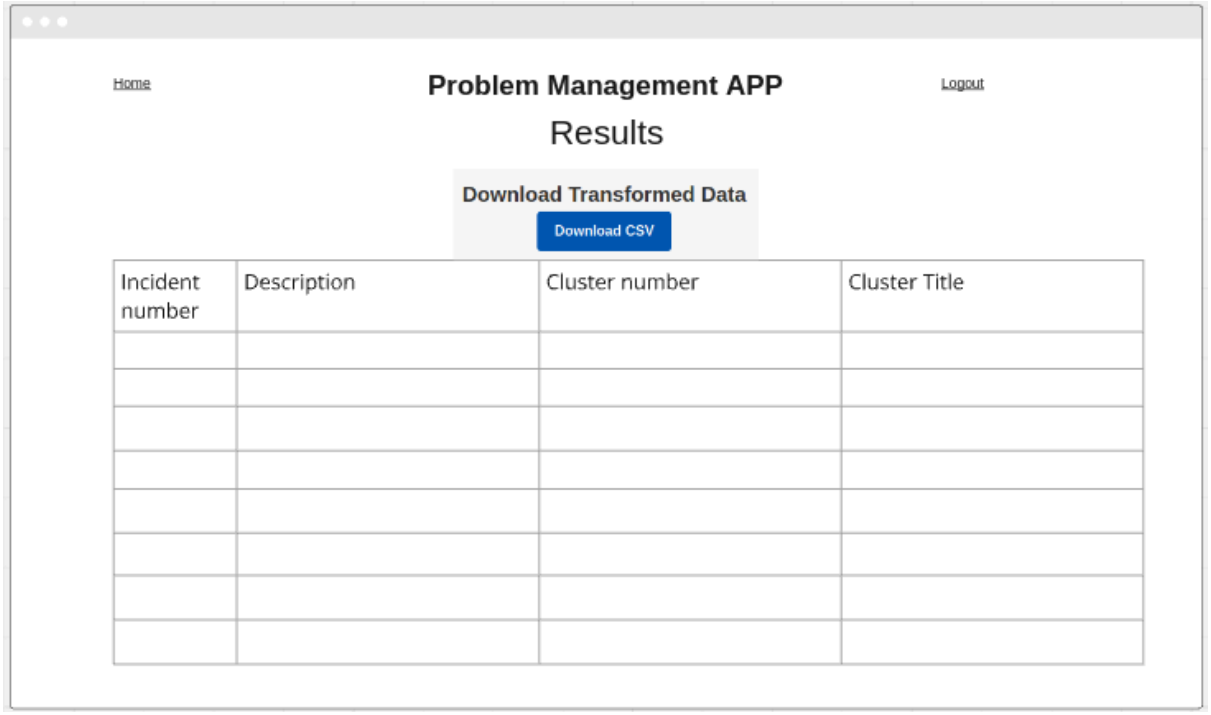
IV. Développer les composants techniques et les interfaces d'une application

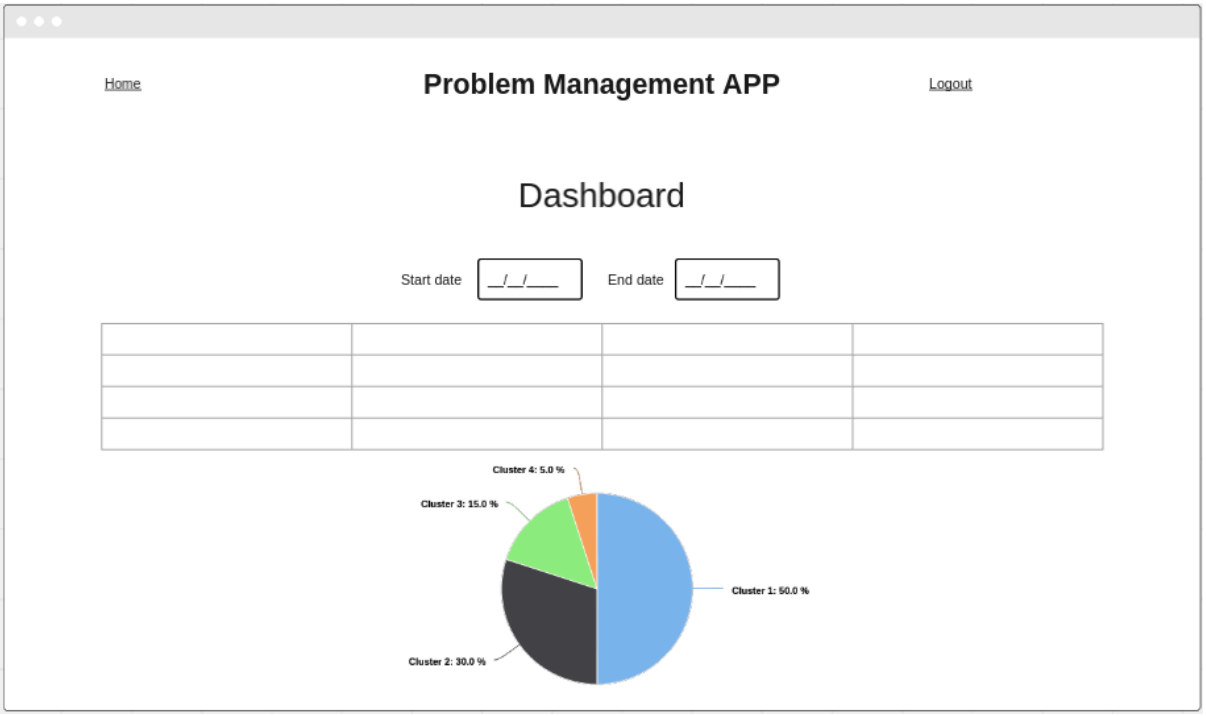
Des maquettes de l'application ont été réalisées en amont :






Problem Management APP
Clustering
File: Parcourir... Aucun fichier sélectionné.
Upload file


Home
Problem Management APP
Logout
Results
Download Transformed Data
Download CSV
Incident number
Description
Cluster number
Cluster Title



Toutes les pages ont été développées et fonctionnent comme prévu :

- page d'accueil ;
- page de connexion ;
- page d'inscription ;
- page de modification du compte utilisateur ;
- page de prédiction où un fichier csv peut être uploadé ;
- page d'historique des prédictions.

Le framework de l'application est Django 5.0.6. Les interfaces sont réalisées avec les templates et les vues correspondantes. Un fichier de style .css est ajouté au projet pour produire un ensemble harmonieux.

L'accès à l'historique des prédictions et à la page de réalisation des prédictions est réservé aux utilisateurs identifiés grâce au décorateur @login_required.

Des tests permettent de tester toutes les vues de l'application ainsi que l'accès limité à l'application et sont disponibles dans le fichier **tests.py**. Ils sont rédigés grâce au framework django.test.

Par exemple, voici les tests permettant de tester la vue du dashboard de l'historique des prédictions :

```python
class DashboardPredictionsViewTests(TestCase):
    def setUp(self):
        """
        Set up the test environment for the dashboard predictions view.

        - Initializes the test client
        - Creates and logs in a user for testing
        - Sets the URL for the dashboard predictions view
        """
        self.client = Client()
        self.url = reverse('dashboard_predictions')
        self.user = User.objects.create_user(username='testuser', password='password')
        self.client.login(username='testuser', password='password')

    def test_dashboard_predictions_view_status_code(self):
        """
        Test the HTTP status code for the dashboard predictions view.

        - Asserts that the dashboard predictions view returns a status code of 200
        """
        response = self.client.get(self.url)
        self.assertEqual(response.status_code, 200)

    def test_dashboard_predictions_view_template(self):
        """
        Test the template used by the dashboard predictions view.

        - Asserts that the dashboard predictions view uses the 'dashboard_predictions.html' template
        """
        response = self.client.get(self.url)
        self.assertTemplateUsed(response, 'dashboard_predictions.html')

    def test_dashboard_predictions_view_data(self):
        """
        Test the context data provided by the dashboard predictions view.

        - Asserts that the context contains 'predictions' and 'data_points'
        """
        response = self.client.get(self.url)
        self.assertIn('predictions', response.context)
        self.assertIn('data_points', response.context)
```

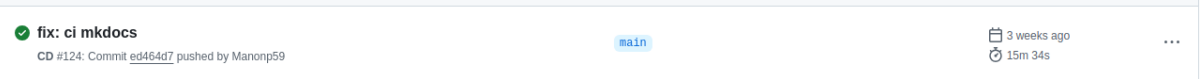
La documentation concernant les tests de l'application Django est versionnée à la racine du dépôt github et accessible ici : [Documentation tests application.]

V. Automatiser les phases de test du code source

Les tests de l'application présentés précédemment sont automatisés et inclus dans le pipeline de Continous integration qui se déclenche à chaque push sur la branch main du projet grâce à la ligne :

```yaml
      - name: Run Django Tests
        working-directory: web_app
        run: python manage.py test
```

Il s'exécute dans github actions :



La documentation concernant le pipeline de continuous integration de l'application web est versionnée à la racine du dépôt github et accessible ici : [Documentation CI](#).

VI. Créer un processus de livraison continue d'une application

L'application Django est déployée de manière automatique grâce à un pipeline de Continuous Deployment disponible dans le fichier **cd.yaml** qui est déclenché à chaque push sur la branche main. Il s'appuie sur le dockerfile :

```
# Définition des variables d'environnement
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Définition du répertoire de travail dans le conteneur
WORKDIR /app

# Copie du fichier requirements.txt dans le conteneur
COPY ./requirements-web_app.txt /app/requirements.txt

# Installation des dépendances
RUN pip install --no-cache-dir --upgrade -r requirements.txt

# Copie de l'ensemble du projet dans le conteneur
COPY . /app

# Installer les dépendances système requises
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    unixodbc \
    unixodbc-dev \
    gcc \
    && rm -rf /var/lib/apt/lists/*

ENV ACCEPT_EULA=Y
RUN apt-get update -y && apt-get update \
  && apt-get install -y --no-install-recommends curl gcc g++ gnupg unixodbc-dev

# Add SQL Server ODBC Driver 17 for Ubuntu 18.04
RUN curl https://packages.microsoft.com/keys/microsoft.asc | apt-key add - \
  && curl https://packages.microsoft.com/config/debian/10/prod.list > /etc/apt/sources.list.d/mssql-release.list \
  && apt-get update \
  && apt-get install -y --no-install-recommends --allow-unauthenticated msodbcsql17 mssql-tools \
  && echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bash_profile \
  && echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bashrc

# Exporter la variable d'environnement pour la configuration Django
ENV DJANGO_SETTINGS_MODULE=prbmg.settings
ENV PYTHONUNBUFFERED=1

# Accept the build argument and set it as an environment variable
ARG APPLICATIONINSIGHTS_CONNECTION_STRING
ENV APPLICATIONINSIGHTS_CONNECTION_STRING=${APPLICATIONINSIGHTS_CONNECTION_STRING}

# Exécuter la commande collectstatic pour rassembler les fichiers statiques
RUN python manage.py collectstatic --noinput

# Exposition du port 8002 vers l'extérieur
EXPOSE 8002

# Commande pour exécuter Gunicorn avec votre application WSGI
CMD ["gunicorn", "--bind", ":8002", "--timeout", "300", "prbmg.wsgi:application"]
```

Nous utilisons un docker compose qui permet de push les images docker des API et de l'application en une seule commande :

```yaml
version: '3.8'

services:

  api_database:
    build: ./api_database
    image: manon29/api_database:latest
    ports:
      - 8000:8000

  api_ia:
    build: ./api_ia
    image: manon29/api_ia:latest
    ports:
     - 8001:8001

  app_web:
    build:
      context: ./web_app
      args:
        APPLICATIONINSIGHTS_CONNECTION_STRING: ${APPLICATIONINSIGHTS_CONNECTION_STRING}
    image: manon29/web_app:latest
    ports:
      - 8002:8002
    depends_on:
      - api_ia
      - api_database
```

Le pipeline de CD comprend les étapes de :

- build and push sur le docker hub de la Docker Image à partir des DockerFile et du DockerCompose :

```yaml
- name: Build and Push Docker Images using Docker Compose
  env:
    DOCKERHUB_USERNAME: ${{ secrets.DOCKERHUB_USERNAME }}
    APPLICATIONINSIGHTS_CONNECTION_STRING: ${{ secrets.APPLICATIONINSIGHTS_CONNECTION_STRING }}
  run: |
    echo "${{ secrets.DOCKERHUB_PASSWORD }}" | docker login -u ${{ secrets.DOCKERHUB_USERNAME }} --password-stdin
    docker compose build
    docker compose push
```

- déploiement de l'application dans une Azure Conteneur Instance :

```yaml
- name: Deploy Web app  to Azure Container Instances
  uses: azure/aci-deploy@v1
  with:
    resource-group: ${{ secrets.RESOURCE_GROUP }}
    dns-name-label: prbmg-web-app
    image: docker.io/manon29/web_app:latest
    registry-login-server: docker.io
    registry-username: ${{ secrets.DOCKERHUB_USERNAME }}
    registry-password: ${{ secrets.DOCKERHUB_PASSWORD }}
    name: prbmg-web-app
    location: francecentral
    ports: '8002'
    secure-environment-variables: |
      API_IA_SECRET_KEY=${{ secrets.API_IA_SECRET_KEY }}
      API_DATABASE_SECRET_KEY=${{ secrets.API_DATABASE_SECRET_KEY }}
      AZURE_DATABASE_NAME=${{ secrets.AZURE_DATABASE_NAME }}
      AZURE_DATABASE_PASSWORD=${{ secrets.AZURE_DATABASE_PASSWORD }}
      AZURE_DATABASE_USERNAME=${{ secrets.AZURE_DATABASE_USERNAME }}
      AZURE_SERVER_NAME=${{ secrets.AZURE_SERVER_NAME }}
      DRIVER=${{ secrets.DRIVER }}
      APPLICATIONINSIGHTS_CONNECTION_STRING=${{ secrets.APPLICATIONINSIGHTS_CONNECTION_STRING}}
```

Il s'exécute dans github actions :

La documentation concernant le pipeline de continuous deployment de l'application web est versionnée à la racine du dépôt github et accessible ici : Documentation CD.