

Rapport projet tutoré n°2

2024 - 2025

**SUJET : IMPLÉMENTATION
D'UNE APPLICATION DE
VISUALISATION DE DONNÉES
MASSIVES AVEC DJANGO**

Tutrice : Claudia VASONCELLOS

Quentin COUZINET
Sacha Gicquel
Nabil SABRY
Toky ANDRIANJAFY

— CONTEXTE

Dans le cadre du projet tutoré de Licence Professionnelle, nous avons eu comme objectif de développer une application web avec Django permettant de visualiser les données massives liées à l'affaire Enron, en mettant l'accent sur les manipulations comptables et les transactions suspectes.

L'application permettrait d'analyser les interactions entre les employées internes et externes d'Enron avant sa faillite, ainsi que de mettre en évidence les pratiques frauduleuses.

Grâce à des outils de visualisation de données comme Chart.js, l'application offrirait des graphiques interactifs permettant une exploration dynamique des données, facilitant ainsi la compréhension de l'ampleur de la fraude et de ses conséquences. Ce projet vise à démontrer l'impact de ces pratiques sur l'entreprise tout en offrant une plateforme d'analyse claire et efficace des données massives associées à l'affaire.

SOMMAIRE

ENRON	p.5
• Présentation	
• Objectifs du projet	
<hr/>	
TECHNOLOGIES	p.6
<hr/>	
CRÉATION DU PROJET	p.7
• Installation de Django	
• Création du projet	
• Structure du projet (page6)	
<hr/>	
INSTALLATION DES MODULES	p.9
<hr/>	
CRÉATION DU MODÈLE DE BASE DE DONNÉES	p.10
<hr/>	
Migration vers la base de données	p.11
<hr/>	
JEU DE DONNÉES & SCRIPT DE PEUPLEMENT	p.13

INTERFACE DE L'APPLICATION

p.19

TEMPLATES HTML

p.23

PRÉSENTATION DES PAGES

p.25-37

CONCLUSION

p.40



ENRON

PRÉSENTATION

Enron était une entreprise américaine de courtage en énergie, fondée en 1985, qui s'est rapidement imposée comme l'un des leaders du secteur de l'énergie et des commodités. L'entreprise était reconnue pour son innovation dans le marché de l'énergie, en utilisant des contrats à terme et d'autres instruments financiers pour spéculer sur les prix de l'énergie. Enron s'est diversifiée dans de nombreux domaines, allant des pipelines de gaz naturel aux projets d'énergie renouvelable, avant de devenir un acteur majeur dans les marchés financiers.



La faillite d'Enron, survenue en 2001, a été déclenchée par la découverte de pratiques comptables frauduleuses qui consistaient à dissimuler des dettes massives et à gonfler artificiellement les profits de l'entreprise. Enron avait utilisé des sociétés écrans et des techniques financières complexes pour masquer sa véritable situation financière. Lorsque ces manipulations ont été révélées, la confiance des investisseurs s'est effondrée, entraînant la chute de l'entreprise. La faillite a eu des conséquences dramatiques, affectant des milliers d'employés, investisseurs et partenaires commerciaux, et a conduit à des réformes majeures de la réglementation financière.

OBJECTIFS DU PROJET

L'objectif principal est l'implémentation d'une application web pour visualiser des informations pertinentes contenues les e-mails échangés entre les employés d'Enron (dont quelques externes).

Un deuxième objectif est le développement d'un script d'automatisation du peuplement de la base de données, à partir des fichiers (texte plain) contenus dans le jeu de données (20 Go approximativement).

TECHNOLOGIES

DJANGO

- Le framework principal pour gérer le backend de l'application, la gestion des bases de données. Django permettra de créer l'architecture de l'application web et d'intégrer les données nécessaires pour la visualisation.

PYTHON

- Le langage principal de l'application, qui sera utilisé pour manipuler et analyser les données massives avant de les envoyer au frontend.

JAVASCRIPT

- Langage qui permettra d'afficher les données de manière claire et dynamique sur le frontend, notamment grâce à l'utilisation de bibliothèques de visualisation de données permettant de créer des graphiques interactifs (courbes, diagrammes, cartes, etc).

HTML / TAILWIND CSS

- Technologies utilisées pour construire l'interface utilisateur de l'application et rendre la présentation des données visuellement attrayante et réactive.

TRELLO

- Trello permettra à l'équipe d'organiser et de suivre l'avancement du projet en créant des tableaux et des cartes pour chaque tâche. Cela facilitera la gestion, la collaboration et le suivi des étapes du projet.

GIT / GITHUB

- Git et GitHub seront utilisé pour gérer le code source du projet, permettant à l'équipe de collaborer efficacement. Chaque membre pourra travailler sur des branches séparées, proposer des modifications via des pull requests et suivre l'historique des versions du projet. Cela facilitera le contrôle de version, la révision du code et l'intégration des contributions de chacun.

CRÉATION DU PROJET

INSTALLATION DE DJANGO

Avant de créer un projet, il est nécessaire d'installer Django. Nous pouvons utiliser pip pour l'installer :

- Linux / macOS: `python -m pip install Django==5.1.6`
- Windows: `py -m pip install Django==5.1.6`

CRÉATION DU PROJET

Une fois Django installé, nous pouvons créer un nouveau projet avec la commande `django-admin startproject`. Cela crée un répertoire contenant la structure de base d'un projet Django : `django-admin startproject base_project`. Ensuite, il faut créer une application Django qui est un module fonctionnel du projet. Cela peut être fait avec la commande `startapp` : `python manage.py startapp base_app`.

```

    ✓ base_project
        > __pycache__
        ⚡ __init__.py
        ⚡ asgi.py
        ⚡ settings.py
        ⚡ urls.py
        ⚡ wsgi.py
        ⚡ .gitignore
        ≡ db.sqlite3
        ⚡ manage.py

```

Figure 1: Dossier `base_project`

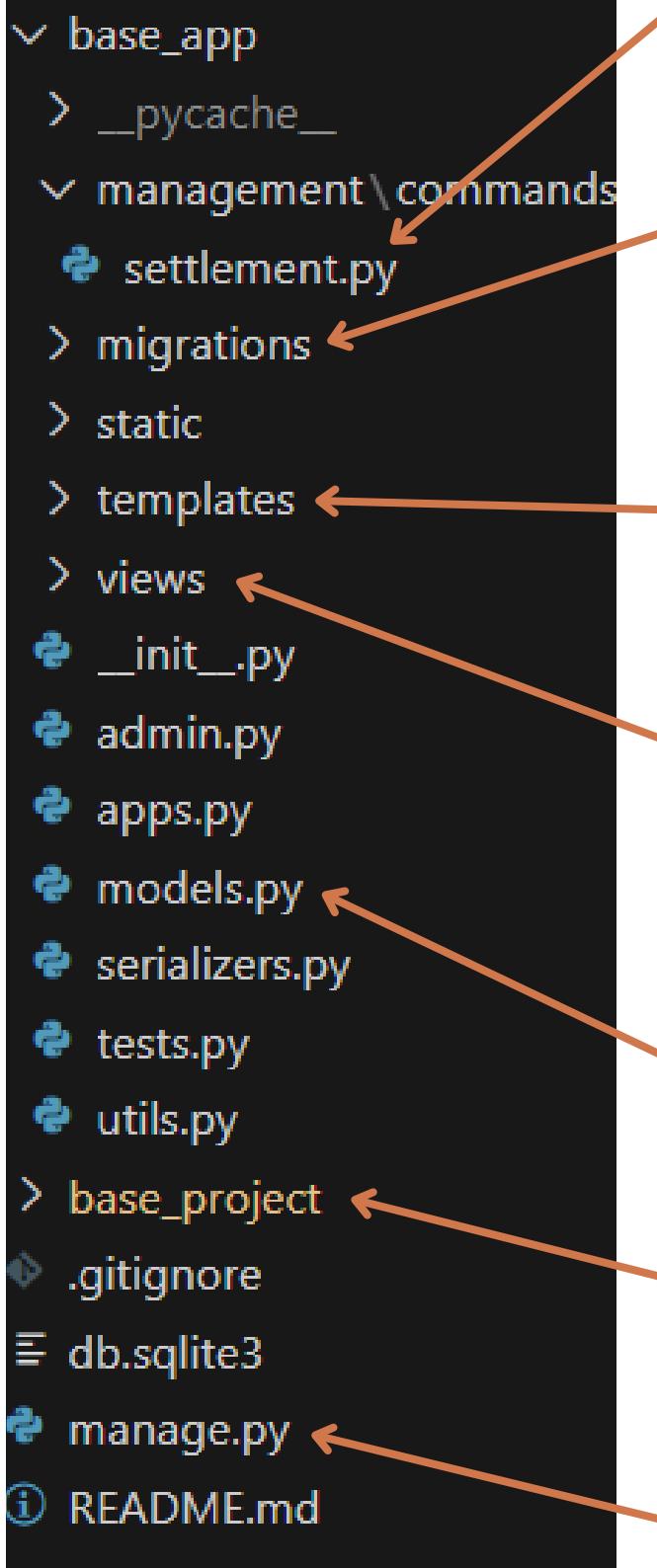
```

    ✓ base_app
        > __pycache__
        > management
        > migrations
        > static
        > views
        ⚡ __init__.py
        ⚡ admin.py
        ⚡ apps.py
        ⚡ models.py
        ⚡ serializers.py
        ⚡ tests.py
        ⚡ utils.py

```

Figure 2 : Dossier `base_app`

STRUCTURE DU PROJET



Le fichier **settlement.py** contient le script de peuplement

Le dossier **migrations** générera les évolutions de la base de données au fil du développement de notre projet. Ce dossier contient des fichiers de migration qui permettent de maintenir la synchronisation entre les modèles de données (définis dans **models.py**) et la structure réelle de la base de données.

Le dossier **templates** contient les fichiers HTML qui définissent la structure des pages web de notre projet Django et qui peuvent être dynamiquement rendus avec les données provenant des vues.

Le dossier **views** gère la logique de traitement des requêtes et la récupération des données. Le lien entre les vues et les templates se fait principalement par la fonction `render()`, qui prend les données et les insère dans un template pour générer une réponse HTML que l'utilisateur voit dans son navigateur.

Le fichier **models.py** permet de définir les modèles de données, qui correspondent aux tables de la base de données.

Le dossier **base_project** contient tout le nécessaire pour développer notre application Django : gestion des bases de données, configuration, routage des URLs,...

Le fichier **manage.py** simplifie l'exécution des commandes courantes, en particulier pour le développement, la gestion des migrations, le lancement du serveur de développement, l'administration, et bien plus encore.

Figure 3 : Structure du projet

INSTALLATIONS DES MODULES

Il nous faudra ensuite installer les modules suivants :

- psycopg (ou psycopg2) : Comme nous utilisons PostgreSQL comme base de données dans ce projet, ce module est essentiel pour assurer la communication entre Django et PostgreSQL. La version avec l'option [binary] installe une version binaire précompilée de psycopg, ce qui simplifie l'installation en évitant la compilation manuelle.
- nltk : Si le projet implique la manipulation ou l'analyse de textes (par exemple, l'analyse de mails ou de contenu textuel dans des données massives), nltk peut être utile pour effectuer des traitements linguistiques ou des analyses textuelles sur les données collectées.
- dateutil : Ce module peut être utile pour manipuler des dates et des heures de manière complexe, par exemple pour gérer des dates de mails, des horaires d'envoi, ou des intervalles temporels dans les données massives.
- djangorestframework : Si nous avons besoin de créer une API pour permettre à des utilisateurs ou des applications externes de consulter, manipuler ou visualiser des données massives, DRF sera essentiel. Il nous permettra de construire facilement des points d'accès pour interagir avec nos modèles de données, en exposant des endpoints HTTP sécurisés.

Voici les commandes pour les installer :

- pip install "psycopg[binary]"
- pip install nltk
- pip install python-dateutil
- pip install djangorestframework

CRÉATION DU MODÈLE DE BDD

Nous allons maintenant créer le modèle nécessaire à l'implémentation d'une application de visualisation de données massives avec Django. Dans le cadre de ce projet, le modèle est conçu pour gérer des informations sur les employés, leurs adresses e-mail, ainsi que les mails eux-mêmes et leurs destinataires.

Dans le fichier `base_app/models.py`, chaque modèle utilise la classe interne `Meta` pour spécifier un nom de table personnalisé dans la base de données, garantissant ainsi une structure de données bien définie et facilitant la gestion des employés, des e-mails et des mails dans l'application.

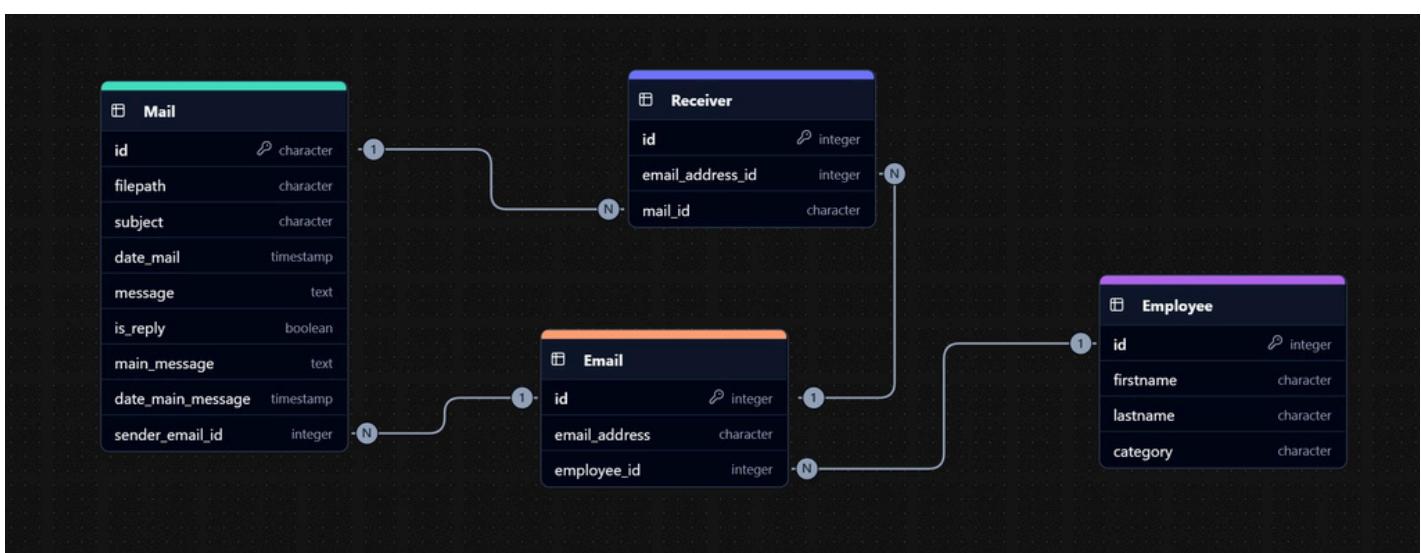


Figure 4 : Schéma fait avec ChartDB

Ce modèle Django contient quatre classes principales :

- **Employee** : Le modèle `Employee` représente un employé, avec des champs pour son prénom (`firstname`), nom (`lastname`), et une catégorie optionnelle (`category`). Le champ `id` est une clé primaire auto-incrémentée. Cela permet de lier plusieurs adresses e-mail à un seul employé.
- **Email** : Le modèle `Email` contient l'adresse e-mail d'un employé et fait référence à l'employé via une clé étrangère (`employee_id`). Cette relation permet de lier un employé à ses différentes adresses e-mail.
- **Mail** : Le modèle `Mail` représente un mail, avec des champs pour le fichier attaché (`filepath`), le sujet du mail (`subject`), le contenu du message (`message`), et un champ booléen (`is_reply`) pour indiquer si le mail est une réponse. Il contient aussi des informations relatives au message principal (`main_message` et `date_main_message`). Ce modèle fait référence à l'adresse e-mail de l'expéditeur via une clé étrangère (`sender_email_id`).
- **Receiver** : Le modèle `Receiver` relie un mail à son destinataire. Il fait référence à l'adresse e-mail du destinataire (`email_address_id`) et au mail lui-même (`mail_id`), permettant ainsi d'établir une relation entre les mails et leurs destinataires.

MIGRATION VERS LA BDD

Dans ce projet, nous utiliserons PostgreSQL comme système de gestion de base de données (SGBD). PostgreSQL est une base de données relationnelle robuste et performante, idéale pour gérer de grandes quantités de données, ce qui est particulièrement utile pour une application de visualisation de données massives. Pour que Django puisse interagir correctement avec PostgreSQL, il est nécessaire de configurer les paramètres de la base de données dans le fichier `base_project/settings.py` de notre projet.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'enron',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Figure 5 : Configuration du fichier `settings.py`

Il faudrait au préalable avoir construit la base de données sur PostgreSQL avec la requête suivante : `CREATE DATABASE enron`.

Maintenant que les fichiers `models.py` et `settings.py` sont correctement configurés, et que la base de données a été créée, nous pouvons passer à l'étape de migration des modèles vers la base de données.

COMMANDES :

- `python manage.py makemigrations` : Cette commande va générer un fichier de migration dans `base_app/migrations` qui contient les instructions pour créer les tables correspondantes à nos modèles dans la base de données.
- `python manage.py migrate` : Cette commande va appliquer toutes les migrations en attente et créer les tables correspondantes dans notre base de données PostgreSQL.



Nous allons utiliser pgAdmin pour le traitement des données PostgreSQL, un outil graphique qui facilite l'administration et la gestion des bases de données PostgreSQL. Si la migration des données s'est bien déroulée, nous devrions obtenir les résultats attendus directement dans la base de données, avec toutes les informations correctement intégrées et accessibles via pgAdmin.



The screenshot shows the pgAdmin interface with the 'Tables (14)' node expanded. Below it, four tables are listed: Email, Employee, Mail, and Receiver. Each table has its own expandable section showing its columns.

- Email:** Contains 3 columns: id, email_address, and employee_id.
- Employee:** Contains 4 columns: id, firstname, lastname, and category.
- Mail:** Contains 9 columns: id, filepath, subject, date_mail, message, is_reply, main_message, date_main_message, and sender_email_id.
- Receiver:** Contains 3 columns: id, email_address_id, and mail_id.

Figure 6 : Tables après migration

Les tables en dessous des 4 tables initiales font partie du système de gestion des permissions et des groupes dans Django. Elles sont utilisées pour contrôler l'accès à différentes parties de l'application en fonction des rôles des utilisateurs. Comme nous avons utilisé une application basée sur Django, ces tables peuvent être présentes pour gérer les rôles et les autorisations d'accès.

JEU DE DONNÉS ET SCRIPT DE PEUPLEMENT

Une fois toutes ces étapes terminées, nous sommes passés à la création du script de peuplement. Ce script, écrit en Python et conçu comme une commande personnalisée pour Django, a pour objectif de peupler la base de données de l'application en insérant des employés et des mails à partir de fichiers XML et de répertoires contenant des mails bruts.

Nous avions deux types de données de base : d'abord un fichier employees.xml de la forme suivante :

```
<employees>
  <employee>
    <lastname>Heard</lastname>
    <firstname>Marie</firstname>
    <email address="marie.heard@enron.com"/>
    <mailbox>heard-m</mailbox>
  </employee>
  <employee category="Employee">
    <lastname>Taylor</lastname>
    <firstname>Mark</firstname>
    <email address="mark.e.taylor@enron.com"/>
    <email address="mark.taylor@enron.com"/>
    <email address="e.taylor@enron.com"/>
    <mailbox>taylor-m</mailbox>
  </employee>
</employees>
```

Figure 7 : Format du fichier employee.xml

On retrouve le nom, le prénom, une ou plusieurs adresses email et enfin le nom de son dossier dans le dossier « maildir ».

Le dossier « maildir » est un dossier qui contient un ensemble de sous-dossiers (150) qui contiennent eux-mêmes des dossiers représentant l'ensemble des données de la boîte mail de chaque employé.

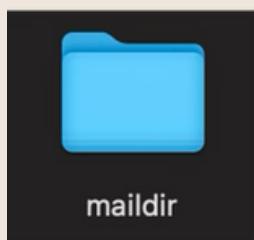


Figure 8 : Dossier maildir

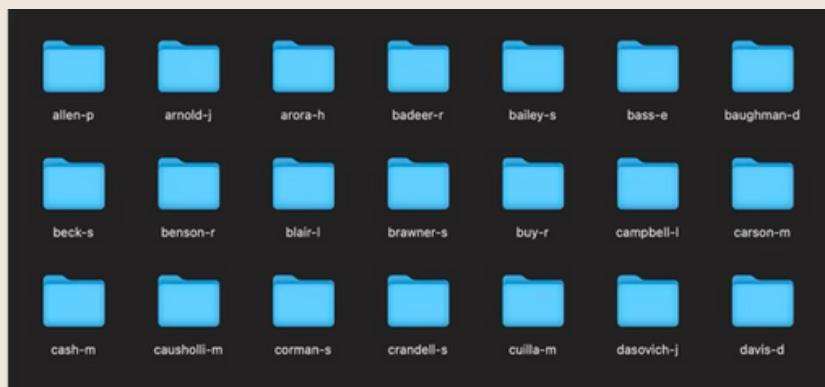


Figure 9: Contenu du dossier maildir

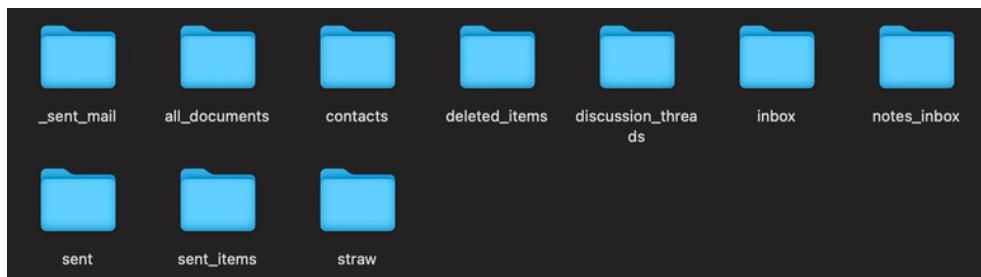


Figure 10 : Contenu du dossier (allen-p)

Dans chaque dossier on retrouve des fichiers avec un mail de la forme:

```

Message-ID: <22701292.1075855726133.JavaMail.evans@thyme>
Date: Thu, 8 Mar 2001 06:46:00 -0800 (PST)
From: ina.rangel@enron.com
To: information.management@enron.com
Subject: Mike Grigsby
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
X-From: Ina Rangel
X-To: Information Risk Management
X-cc:
X-bcc:
X-Folder: \Phillip_Allen_June2001\Notes Folders\'sent mail
X-Origin: Allen-P
X-FileName: pallen.nsf

Please approve Mike Grigsby for Bloomberg.

Thank You,
Phillip Allen

```

Figure 11 : Format d'un mail

On constate donc dans l'ordre un message-ID unique pour chaque mail, il nous servira d'id dans notre base de données, la date et l'heure à laquelle le mail a été envoyé, l'adresse mail de la personne qui a envoyé ce mail et enfin le sujet du mail, le reste des informations ne sont pas très importantes pour l'instant.

En tout, dans le dossier maildir, on retrouve 3 499 dossiers et 517 401 fichiers. Il est donc primordial d'avoir un script de peuplement robuste et efficace pour traiter un tel volume de données le plus rapidement possible.

Le script de peuplement est une classe Command qui hérite de BaseCommand (c'est une classe de base utilisée dans le framework Django pour créer des commandes personnalisées) et constitue le point d'entrée de la commande. La méthode « handle » orchestre les différentes étapes du peuplement.

```

def handle(self, *args, **options):
    """
    Méthode principale exécutée lorsqu'on lance la commande Django.
    """
    self.stdout.write()
    self.stdout.write(self.stylize("SUPPRESSION DES DONNÉES PRÉCÉDENTES", "CYAN"))
    self.deleteAll()

    self.stdout.write()

    self.stdout.write(self.stylize("PEUPLEMENT DES EMPLOYÉS", "CYAN"))
    self.stdout.write(f"Traitement de {self.stylize('employees.xml', 'PURPLE')}...")
    self.populateEmployees()
    Employee.objects.create(firstname="Personne", lastname="Externe", category="Externe")

    self.stdout.write()

    self.stdout.write(self.stylize("PEUPLEMENT DES MAILS", "CYAN"))
    self.startPopulateMails(options)

```

Figure 12 : Méthode handle

Dans cette méthode, on retrouve 3 grandes étapes, qui constituent les 3 grandes étapes du script de peuplement :

1. SUPPRESSION DES DONNÉES PRECEDENTES

Tout d'abord la méthode « `deleteAll()` » est appelée pour effacer toutes les données existantes. Cela nous permet de partir sur une base vide pour un peuplement fiable.

2. PEUPLEMENT DES EMPLOYÉS

Dans cette étape, on appelle la méthode « `populateEmployees()` », qui traite le fichier `employees.xml` pour insérer chaque employé dans la base de données. À noter que pour toutes les adresses mails externes, donc qui ne respectent pas la terminaison « `@enron.com` », on crée un employé externe pour regrouper les adresses mails dites externes.

Cette méthode permet le traitement des employés. Pour cela, elle lit le fichier XML, extrait les informations de chaque employé (prénom, nom, emails, catégorie) et les insère dans la base de données.

Pour ce faire, on utilise le module `xml.etree.ElementTree` et sa méthode `parse()` pour parcourir correctement le fichier XML.

Chaque élément parent est analysé, et pour chacun de ses enfants on enregistre l'information en fonction du tag de l'élément (`firstname`, `lastname`, `email`, `mailbox`). S'il s'agit du tag “`email`” on enregistre la liste des emails.

```
# Parcourir les éléments du fichier
for child in root:
    # Incrémenter le compteur
    initial_emp_count += 1

    # Récupérer la catégorie de l'employé si celle-ci est définie
    category = child.attrib['category'] if (child.attrib != {}) else None

    emails = []
    # Récupérer le reste des informations de l'employé (nom, prénom, liste des emails, nom de la boîte mail)
    for subchild in child:
        match subchild.tag:
            case "firstname":
                firstname = subchild.text
            case "lastname":
                lastname = subchild.text
            case "email":
                emails.append(subchild.attrib['address'])
            case "mailbox":
                mailbox = subchild.text
```

Figure 13 : extrait de la méthode populateEmployees

Une fois que les informations de l'employé sont enregistrées, on crée un nouvel objet employée ainsi qu'un nouvel objet Email pour chaque email de la liste. C'est ensuite enregistré en base de données.

```
# Créer une instance Employee et l'enregistrer dans la base
try:
    new_emp = Employee(firstname=firstname, lastname=lastname, category=category)
    new_emp.save()
    inserted_emp_count += 1

    # Gérer la liste des emails de l'employé (+mailbox?)
    for email in emails:
        new_email = Email(email_address=email, employee_id=new_emp)
        new_email.save()
except Exception as e:
    print(f"{self.stylize('Erreur lors de l'insertion de l'employé :', 'ERROR')}{e}")
```

Figure 14 : enregistrement des données

Pour finir, lorsque chaque élément a été traité, on affiche le résultat grâce au compteur qui a été incrémenté à chaque parcours de la boucle.

À noter que tout ce qui peut générer une erreur est pris en compte par un try/catch pour gérer les exceptions.

3. PEUPLEMENT DES MAILS

Enfin, la plus grosse partie de ce script de peuplement est l'insertion des mails via la méthode « startPopulateMails() » qui se charge de parcourir les dossiers et traiter les mails. Cette dernière est divisée en plusieurs fonctions pour rendre le code plus lisible et éviter les répétitions.

Chaque mail présent dans maildir est analysé et inséré dans la base de données en suivant un processus structuré :

1. Lecture et extraction des en-têtes du mail :

Pour chaque fichier trouvé, le script extrait les informations essentielles du mail, telles que :

- L'identifiant unique du mail (Message-ID) qui devient la clé primaire dans la base.
- L'expéditeur (From), qui est ensuite recherché dans la table Email pour voir s'il existe déjà.
- Les destinataires (To), qui sont également vérifiés avant insertion dans Receiver.
- La date d'envoi (Date), qui est convertie en format datetime.
- L'objet (Subject), qui est limité en taille si nécessaire.
- Le contenu principal du mail (Message).

2. Insertion du mail dans la table Mail

Une fois toutes les informations extraites, une nouvelle instance de Mail est créée et insérée dans la base de données.

3. Gestion de l'expéditeur

- Si l'adresse de l'expéditeur existe déjà dans la table Email, elle est récupérée.
- Sinon, une nouvelle entrée est créée dans Email et associée à un Employee.
- Si l'expéditeur ne fait pas partie des employés listés dans employees.xml, il est classé dans un employé générique "Personne Externe".
- L'extraction du nom de l'expéditeur est effectuée à partir du champ X-From, qui peut contenir plusieurs formats ("John Doe", "Doe, John", <email>...).

4. Gestion des destinataires (Receiver)

- Chaque adresse présente dans le champ To est vérifiée.
- Si l'adresse est connue, elle est simplement associée au mail dans la table Receiver.
- Si elle est inconnue, elle est ajoutée dans la table Email, puis rattachée à un employé déjà existant ou, si elle est externe, à "Personne Externe".

OPTIMISATION ET ROUSTESSE

- Étant donné le grand volume de données (517 401 fichiers), le script a été optimisé pour minimiser les accès à la base de données :
- Utilisation d'un cache en mémoire (email_cache) pour éviter des requêtes répétées sur les emails déjà rencontrés.
- Utilisation de bulk_create() pour insérer les mails et destinataires en masse au lieu de faire une requête par élément.
- Gestion des erreurs et des cas particuliers grâce à des blocs try/except pour éviter des interruptions du script.

AFFICHAGE DES RÉSULTATS

Pendant toute l'exécution du script, des affichages sont faits dans la console pour suivre l'avancement.

On utilise une fonction qui permet de styliser la sortie console en ajoutant des couleurs grâce à des codes ANSI. Cela rend l'affichage plus clair et agréable à lire.

Et voici le rendu final dans le terminal, avec l'affichage du nombre de mails et de destinataires insérés dans la base pour chaque personne et les statistiques globales de l'insertion avec le nombre de fichiers total, le nombre de fichiers traités et enfin le nombre de fichiers ignorés.

```
SUPPRESSION DES DONNÉES PRÉCÉDENTES
Toutes les données ont été supprimées.

PEUPLEMENT DES EMPLOYÉS
Traitement de employees.xml...
149/149 employés insérés dans la base de données.

PEUPLEMENT DES MAILS
Traitement d'un seul dossier: /Users/gicquelsacha/Documents/Projet cours/Projet2/data/maildir/allen-p
3034 mails insérés dans la base.
3510 destinataires insérés dans la base.
Traitement d'un seul dossier: /Users/gicquelsacha/Documents/Projet cours/Projet2/data/maildir/arnold-j
4898 mails insérés dans la base.
5400 destinataires insérés dans la base.
Traitement d'un seul dossier: /Users/gicquelsacha/Documents/Projet cours/Projet2/data/maildir/lokay-m
5568 mails insérés dans la base.
9119 destinataires insérés dans la base.
```

Figure 15 : Retour terminal suite à l'exécution du script (début)

```
Traitement du dossier : C:\Users\quent\Documents\LICENCE-PRO\Projet-2-ENRON\data\maildir\zufferli-j
557 mails insérés dans la base.
727 destinataires insérés dans la base.

STATISTIQUES DU PEUPLEMENT DES MAILS
Nombre de fichiers total: 517401
Fichiers traités: 517401
Fichiers ignorés (en-tête manquant): 0
```

Figure 16 : Retour terminal suite à l'exécution du script (fin)

INTERFACE DE L'APPLICATION

PROCESSUS DE CRÉATION DE L'INTERFACE

Pour mettre au point l'interface de notre application de visualisation des données Enron, on a suivi une approche plutôt pragmatique et collaborative :

- Brainstorming initial : On a commencé par organiser plusieurs séances de réflexion pour lister toutes les idées possibles. Qu'est-ce qu'on voulait montrer ? Comment rendre ces milliers d'emails explorables ? On a noté vraiment toutes les idées qui nous venaient en tête, sans se limiter.
- Tri des idées réalisables : Après ce remue-ménages, on a dû faire un choix. On s'est concerté pour discuter de ce qui était faisable dans le temps qu'on avait. Certaines fonctionnalités étaient essentielles, d'autres plus secondaires. On a négocié entre nous pour trouver un bon équilibre.
- Maquettes papier : Ensuite, on a dessiné plusieurs versions de l'interface sur papier. C'était rapide à faire et ça permettait de visualiser concrètement comment seraient organisées les différentes pages. On a partagé ces croquis avec toute l'équipe pour avoir leur avis, et on a fait quelques allers-retours jusqu'à ce que tout le monde soit d'accord.
- Passage à Figma : Une fois qu'on était satisfaits des maquettes papier, on les a redessinées sur Figma. C'était l'occasion de préciser les couleurs, la typographie, l'aspect visuel global. Ça a donné une bien meilleure idée du rendu final et permis d'affiner les détails.

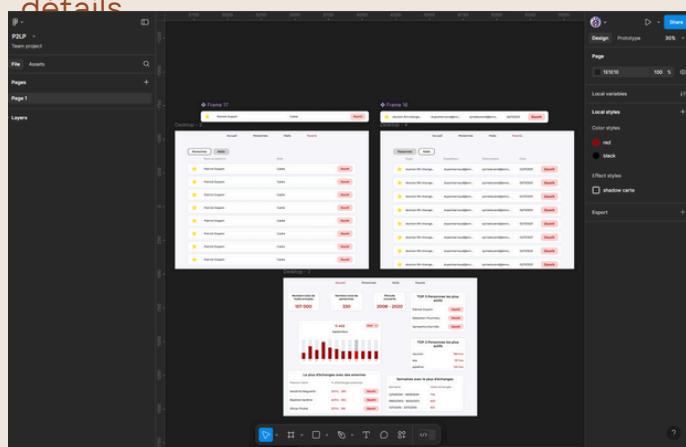


Figure 17 : Maquette Figma

- Développement et ajustements : Quand on est passés à l'implémentation avec Django, on a dû faire quelques compromis et ajustements. La théorie, c'est bien, mais confrontée à la réalité du code et des données, il a fallu adapter certains éléments. Les changements étaient généralement mineurs mais nécessaires pour que tout fonctionne correctement.

COMMENT SE PRÉSENTE L'INTERFACE

L'application est organisée en cinq grandes sections, chacune avec son propre objectif :

La page d'accueil - le tableau de bord

C'est un peu la vitrine de l'application. Dès qu'on arrive, on peut voir :

- Un compteur qui indique combien d'emails sont dans la base
- Le nombre de personnes impliquées dans les échanges
- La période couverte par les données
- Un graphique qui montre l'évolution des emails par mois
- Les personnes qui communiquent le plus avec l'extérieur
- Un top 3 des mots qui reviennent le plus souvent dans les mails
- Les trois adresses email les plus actives

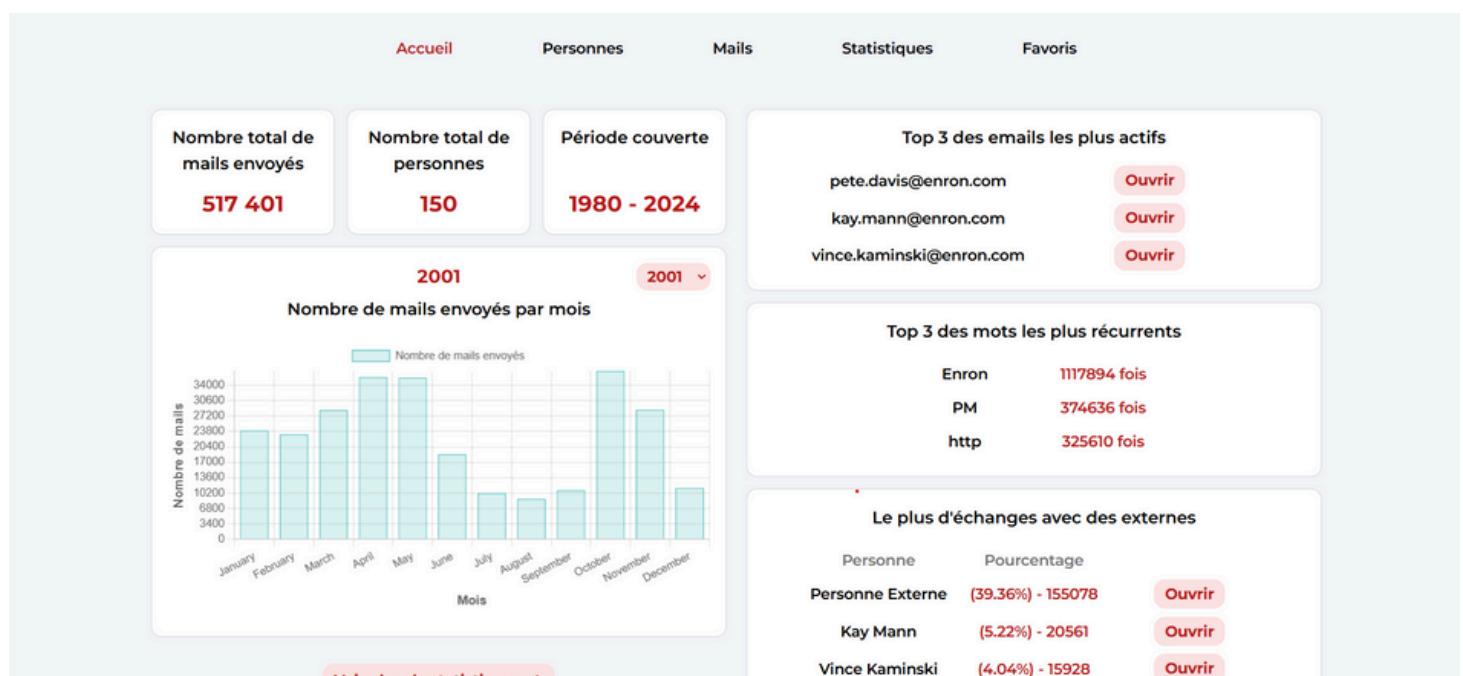


Figure 18 : Page Home

La page des personnes

Cette page permet d'explorer tous les acteurs impliqués dans les échanges :

- On peut voir la liste complète des personnes
- Une barre de recherche permet de trouver quelqu'un rapidement
- On peut filtrer par catégorie pour affiner les résultats
- Pour chaque personne, on a un petit résumé de son activité

The screenshot shows a web application interface for managing contacts. At the top, there is a navigation bar with five items: Accueil, Personnes (highlighted in red), Mails, Statistiques, and Favoris. Below the navigation bar, the title "Liste des personnes" is displayed. Underneath the title is a search and filter section. It includes a dropdown menu labeled "Filtrer par catégorie" with the option "None" selected, and two input fields for "Rechercher" (one for "Prénom" and one for "Nom"). The main content area displays a table of contacts with columns: Prénom, Nom, Catégorie, and Favoris (which contains a star icon). The contacts listed are: Marie Heard (None), Mark Taylor (Employee), Lindy Donoho (Employee), Lisa Gang (None), and Jeffrey Skilling (CEO). Each contact row has a star icon in the "Favoris" column.

Prénom	Nom	Catégorie	Favoris
Marie	Heard	None	★
Mark	Taylor	Employee	★
Lindy	Donoho	Employee	★
Lisa	Gang	None	★
Jeffrey	Skilling	CEO	★

Figure 19 : Page des personnes

La page des favoris

C'est une fonction pratique qu'on a ajoutée :

- On peut marquer certaines personnes comme favorites
- Elles apparaissent alors dans cette page dédiée
- Ça permet d'accéder rapidement aux profils qu'on consulte souvent

C'est particulièrement pratique quand on suit régulièrement certaines personnes importantes dans l'affaire.

The screenshot shows a web application interface for managing favorites. At the top, there is a navigation bar with five items: Accueil, Personnes, Mails, Statistiques, and Favoris (highlighted in red). Below the navigation bar, the title "Employés favoris" is displayed. The main content area displays a table of favorite contacts with columns: Prénom, Nom, Catégorie, and Retirer des favoris (which contains a star icon). The contacts listed are: Mark Taylor (Employee) and Lisa Gang (null). Each contact row has a star icon in the "Retirer des favoris" column.

Prénom	Nom	Catégorie	Retirer des favoris
Mark	Taylor	Employee	★
Lisa	Gang	null	★

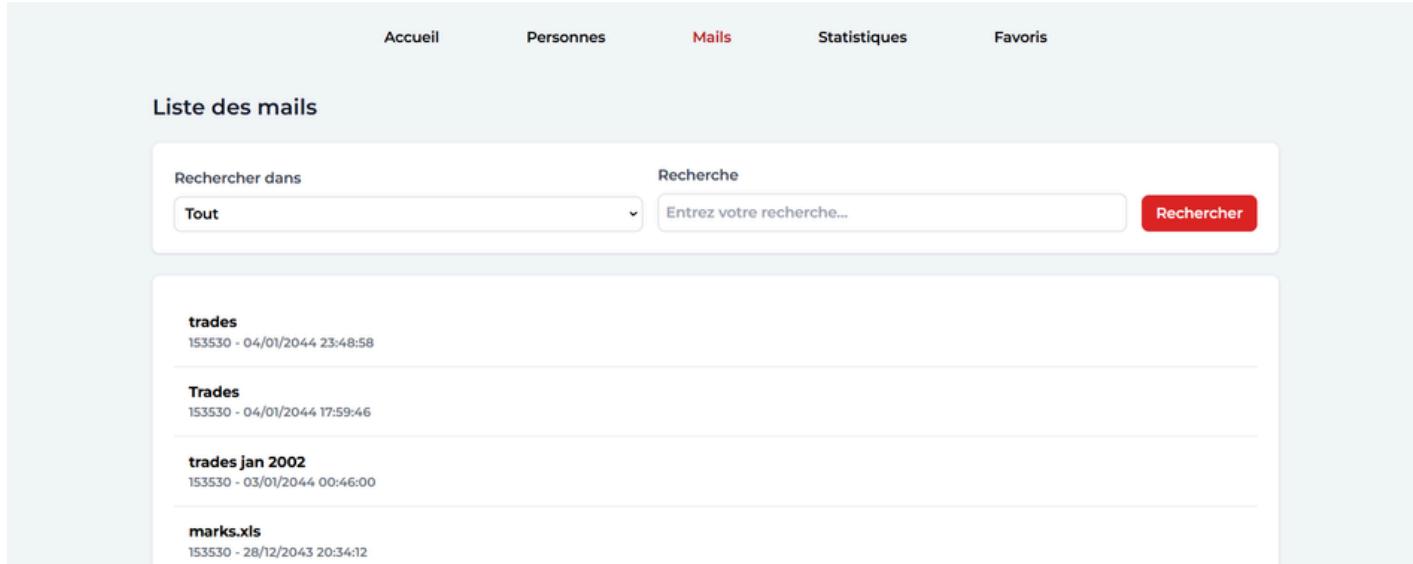
Figure 20 : Page des favoris

La page des mails

C'est vraiment le cœur de l'application :

- On y trouve tous les emails envoyés et reçus
- On peut chercher par mot-clé
- Les filtres permettent de trier par date, expéditeur, etc.
- On voit clairement les discussions entre personnes
- On peut afficher le contenu complet des messages

On a passé pas mal de temps sur cette page pour qu'elle reste fluide malgré la quantité énorme de données.



The screenshot shows the 'Mails' section of the application. At the top, there is a navigation bar with tabs: Accueil, Personnes, Mails (which is highlighted in red), Statistiques, and Favoris. Below the navigation bar, the title 'Liste des mails' is displayed. Underneath the title is a search bar with two input fields: 'Rechercher dans' (with a dropdown menu showing 'Tout') and 'Recherche' (with a placeholder 'Entrez votre recherche...'). To the right of the search bar is a red 'Rechercher' button. The main content area displays a list of email entries, each with a subject and a timestamp. The entries are:

- trades

153530 - 04/01/2044 23:48:58
- Trades

153530 - 04/01/2044 17:59:46
- trades jan 2002

153530 - 03/01/2044 00:46:00
- marks.xls

153530 - 28/12/2043 20:34:12

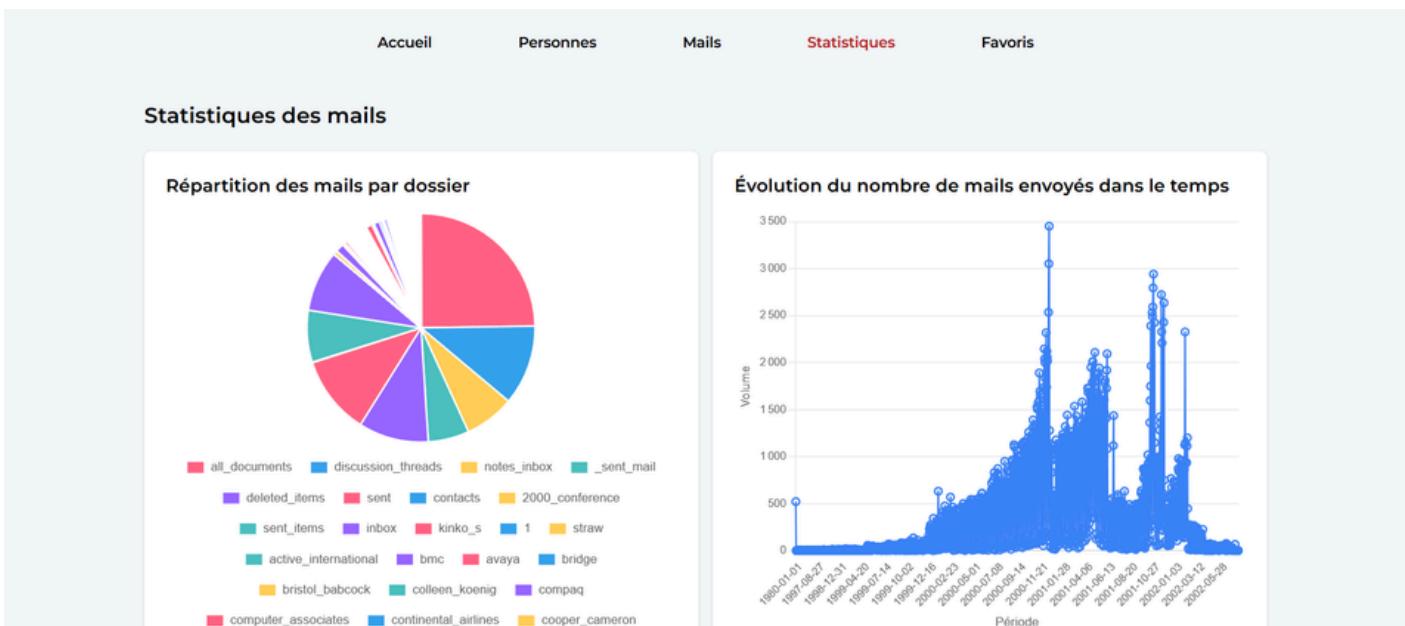
Figure 21 : Page des mails

La page des statistiques

C'est là qu'on entre dans l'analyse plus poussée :

- Différents types de graphiques pour visualiser les données

On a essayé de proposer des visualisations variées pour permettre différents types d'analyses.



The screenshot shows the 'Statistiques' section of the application. At the top, there is a navigation bar with tabs: Accueil, Personnes, Mails, Statistiques (which is highlighted in red), and Favoris. Below the navigation bar, the title 'Statistiques des mails' is displayed. The page features two main data visualizations: a pie chart titled 'Répartition des mails par dossier' and a scatter plot titled 'Évolution du nombre de mails envoyés dans le temps'. The pie chart shows the distribution of emails across various categories, with the largest proportion being 'all_documents'. The scatter plot shows the volume of sent emails over time, with data points plotted from January 1989 to May 2002. The x-axis is labeled 'Période' and the y-axis is labeled 'Volume'.

Figure 22 : Page des statistiques

TEMPLATES HTML

Dans Django, les fichiers dans le dossier `views` et `models.py` jouent un rôle crucial dans la gestion des données et leur présentation. Comme vu précédemment, le fichier `models.py` est chargé de définir les modèles de données, qui sont en quelque sorte les plans représentant les tables de la base de données. De son côté, un fichier de `views` contient les vues qui s'occupent des requêtes HTTP et interagissent avec ces modèles pour récupérer, manipuler ou afficher les données. Les vues vont chercher des objets dans la base de données à travers les modèles et les envoient aux templates HTML pour qu'ils soient affichés à l'utilisateur.

```
class Personne(models.Model):
    nom = models.CharField(max_length=100)
    prenom = models.CharField(max_length=100)
    email = models.EmailField()

    def __str__(self):
        return f"{self.prenom} {self.nom}"
```

Figure 23: Modèle Django
“Personne”

```
from django.shortcuts import render
from models import Personne

def liste_personnes(request):
    personnes = Personne.objects.all()
    return render(request, 'personnes/liste.html', {'personnes': personnes})
```

Figure 24 : Exemple simple d'envoi de la liste des personnes de la vue vers le template

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Liste des personnes</title>
</head>
<body>
    <h1>Liste des personnes</h1>
    <ul>
        {% for personne in personnes %}
            <li>{{ personne.prenom }} {{ personne.nom }} - {{ personne.email }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Figure 25 : Affichage de la liste des personnes dans le template

PRÉSENTATION DES TEMPLATES DU PROJET

Dans notre projet Django, nous avons mis en place 5 templates principaux pour structurer l'application et offrir une navigation fluide aux utilisateurs.

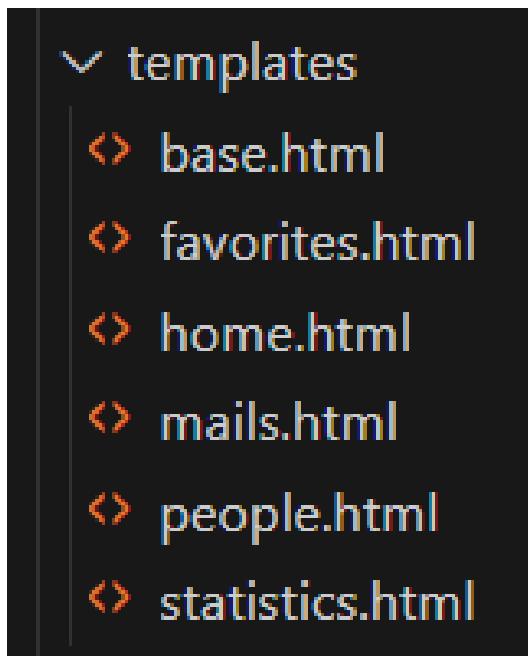


Figure 26 : Dossier templates

- La page d'accueil affiche des statistiques sur les mails envoyés, le nombre de personnes, et la période couverte, avec un graphique interactif. Elle présente également les 3 personnes les plus actives, les mots récurrents et les employés ayant le plus d'échanges externes.
- La page "Personnes" permet de consulter une liste filtrable des employés par catégorie et nom. Elle inclut un système de recherche par prénom et nom, ainsi qu'une fonctionnalité pour ajouter ou retirer des employés de la liste des favoris.
- La page "Favoris" affiche une liste des employés ajoutés aux favoris, avec la possibilité de retirer un employé de la liste.
- La page "Mails" affiche une liste d'emails récupérés dynamiquement avec des options de recherche, permettant de filtrer par ID, chemin de fichier, sujet, message ou date.
- La page "Statistiques" génère une page affichant des graphiques interactifs de statistiques, comme la répartition des mails ou l'évolution dans le temps, avec l'utilisation de Chart.js.

PAGE HOME.HTML

La page home.html est la page principale de l'application. Elle offre un aperçu rapide et efficace des principales statistiques analysées concernant les échanges d'e-mails dans l'affaire Enron.

La vue home.py est responsable de l'affichage de la page d'accueil. Elle récupère plusieurs statistiques à partir de la base de données et les transmet au template home.html pour affichage.

Voici les principales fonctionnalités :

Statistiques générales

- Nombre total d'e-mails envoyés
- Nombre total de personnes identifiées
- Période couverte par les e-mails

Cette donnée est définie en fonction du mail le plus récent et du mail le plus ancien :

```
# Récupérer la période couverte
min_year = Mail.objects.filter(date_mail__year__range=YEARS_RANGE).earliest("date_mail").date_mail.year
max_year = Mail.objects.filter(date_mail__year__range=YEARS_RANGE).latest("date_mail").date_mail.year
covered_period = f"{min_year} - {max_year}"
```

Top 3

- Employés ayant envoyé le plus d'e-mails
- Mots les plus utilisés (les plus récurrents)

Pour cette analyse, nous avons utilisé le module NLTK (Natural Language Toolkit), qui permet un traitement avancé du texte. Ce module fournit une liste de mots vides (stopwords), c'est-à-dire des mots qui apportent peu ou pas d'information au texte, comme les articles, déterminants, prépositions, etc.

En excluant ces stopwords courants (ex. : "the", "and", "of") ainsi qu'une liste personnalisée de mots ajoutés manuellement en fonction des redondances spécifiques aux e-mails, nous obtenons une analyse plus pertinente du contenu des messages.

Cela permet d'identifier les thèmes majeurs des échanges en mettant en avant les mots réellement significatifs.

Le code a été séparé et nous avons créé une fonction dans un fichier utils.py pour améliorer la lisibilité (voir ci-contre)

```
def getExcludedWords():
    """
    Renvoie un ensemble de mots à exclure (stopwords).

    Utile pour analyser de manière cohérente un message.
    """
    nltk.download('stopwords')
    stop_words = set(stopwords.words('english'))
    custom_excluded_words = {"ect", "hou", "com", "subject"}
    excluded_words = stop_words | custom_excluded_words
    return excluded_words
```

- Employés ayant le plus d'échanges avec l'extérieur

Graphique des e-mails envoyés par mois chaque année

Le template affiche un graphique dynamique (via Chart.js) qui représente le nombre d'e-mails envoyés chaque mois. Dans home.py on récupère les données pour le graphique :

```
# Récupérer le nombre de mails envoyés par mois pour l'année choisie dans le graphique
mail_per_month = (
    Mail.objects.filter(date_mail__year=selected_year)
    .annotate(month=TruncMonth('date_mail'))
    .values('month')
    .annotate(count=Count('id'))
    .order_by('month')
)

# Préparer les données pour le graphique avec les mois en français
mail_data = {
    "labels": [m["month"].strftime("%B").capitalize() for m in mail_per_month], # Mois en français
    "values": [m["count"] for m in mail_per_month], # Nombre de mails par mois
}
```

Puis elles sont envoyées au template home.html, qui les affiche sous forme de graphique à barres. L'affichage est géré en JavaScript.

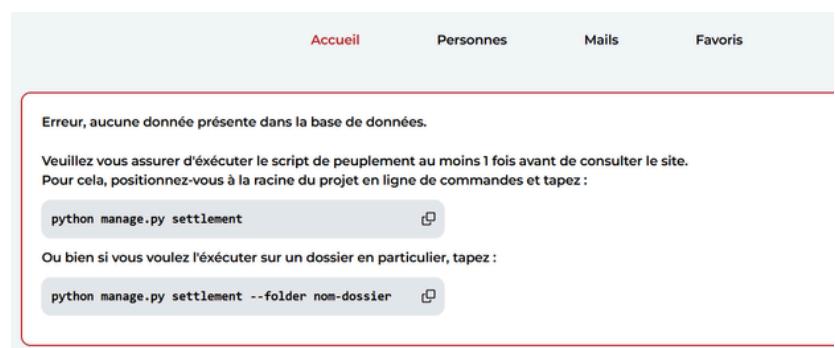


```
// GRAPHIQUE AVEC CHART.JS
// Configuration
const ctx = document.getElementById('mailChart').getContext('2d');
new Chart(ctx, {
    type: 'bar', // Type de graphique (barres verticales)
    data: {
        labels: mailData.labels, // Mois (labels)
        datasets: [{
            label: 'Nombre de mails envoyés',
            data: mailData.values, // Nombre de mails par mois
            backgroundColor: 'rgba(75, 192, 192, 0.2)', // Couleur de fond
            borderColor: 'rgba(75, 192, 192, 1)', // Couleur des bordures
            borderWidth: 1 // Epaisseur des bordures
        }]
    },
});
```

Gestion des erreurs

Si aucune donnée n'est disponible, un message d'alerte est affiché pour inviter l'utilisateur à exécuter le script de peuplement de la base de données.

Ce même message est affichée sur les autres pages le cas échéant.



PAGE PEOPLE.HTML

La page people.html inclut des éléments visuels comme les listes d'employés, le formulaire de recherche et les différentes catégories. Ce modèle s'appuie sur les données fournies par la vue pour créer l'interface de manière dynamique.

```
from django.shortcuts import render
from base_app.models import Employee

def people(request):
    category_filter = request.GET.get('category', None)

    if category_filter:
        employees = Employee.objects.filter(category=category_filter)
    else:
        employees = Employee.objects.all()

    categories = Employee.objects.values_list('category', flat=True).distinct()

    context = {
        'employees': employees,
        'categories': categories,
    }

    return render(request, 'people.html', context)
```

Figure 27 : extrait de code
Fonction people

La vue people, qui est un fichier Python, prend en charge la requête de l'utilisateur et s'occupe de traiter les données nécessaires. Elle va chercher les employés et les catégories dans la base de données, en se basant sur les filtres (par catégorie) qui sont définis dans l'URL, ou par défaut, elle récupère tous les employés. Ensuite, elle crée un contexte qui contient toutes ces informations, comme les employés et les catégories.

Ce contexte est ensuite transmis au template HTML (people.html) grâce à la fonction render(). À ce stade, Django remplace les variables du template par les valeurs appropriées du contexte. Par exemple, la liste des employés sera affichée à l'aide de la balise `{% for employee in employees %}`, et les catégories apparaîtront dans le menu déroulant grâce à `{% for category in categories %}`.

```
{% if employees %}
    {% for employee in employees %}
        <a href="{% url 'mails' %}?person_id={{ employee.id }}">
            <li id="personne" class="border-b border-gray-200 p-4 cursor-pointer transition-colors duration-200 hover:bg-gray-100 flex items-center">
                <span class="flex-1 text-gray-800">{{ employee.firstname }}</span>
                <span class="flex-1 text-gray-800">{{ employee.lastname }}</span>
                <span class="flex-1 text-gray-800">{{ employee.category }}</span>
                <span class="flex-1 text-center text-gray-800">
                    <i class="fa{% if employee.id in favorites %} s{% else %} r{% endif %} fa-star fa-2x"
                        id="favorite-{{ employee.id }}"
                        data-employee-id="{{ employee.id }}"
                        onclick="toggleFavorite(this)"></i>
                </span>
            </li>
        </a>
    {% endfor %}
    {% else %}
        <li class="px-6 py-4 text-center text-gray-800">Aucun employé trouvé.</li>
    {% endif %}
```

PAGE FAVORITES.HTML

La page favorites.html est conçue pour montrer les employés préférés de l'utilisateur. Quand un utilisateur accède à cette page, il découvre une liste des employés qu'il a précédemment marqués comme favoris. Chaque employé est présenté sur une ligne avec son prénom, son nom, sa catégorie, et une étoile qui indique s'il fait partie des favoris. Cette étoile est interactive : si l'employé est déjà dans la liste des favoris, l'étoile brille en jaune, sinon elle est grise. L'utilisateur peut cliquer sur l'étoile pour ajouter ou retirer un employé de ses favoris. Cette action modifie l'état de l'étoile, et le changement est enregistré dans le localStorage du navigateur, ce qui permet aux favoris de rester en place même lorsque l'utilisateur navigue sur d'autres pages.

```
{% block content %}

<div class="container max-w-[1280px] mx-auto p-5">
    <div class="text-2xl font-semibold text-gray-900 mb-6">Employés favoris</div>

    <div class="bg-white rounded-lg shadow p-6">
        <ul class="min-w-full list-none justify-center">
            <li class="bg-gray-100 flex py-3 text-left border-b text-gray-600 px-4 rounded-t-lg">
                <span class="flex-1 text-left">Prénom</span>
                <span class="flex-1 text-left">Nom</span>
                <span class="flex-1 text-left">Catégorie</span>
                <span class="flex-1 text-center">Retirer des favoris</span>
            </li>
            <div id="favorites-list">
                </div>
        <ul>
    </div>
</div>
```

```
favoriteEmployees.forEach(employee => {
  const row = document.createElement('li');
  row.className = "border-b border-gray-200 p-4 cursor-pointer transition-colors duration-200 hover:bg-gray-100 flex items-center";

  row.innerHTML =
    `${employee.firstname}
    ${employee.lastname}
    ${employee.category}
    
      <i class="fa s fa-star"
        id="favorite-${employee.id}"
        data-employee-id="${employee.id}"
        onclick="toggleFavorite(this)"
        style="color: #ffce33;"></i>
    `;
  favoritesList.appendChild(row);
});
```

Les deux fichiers, people.html et favorites.html, interagissent entre eux grâce à l'utilisation du localStorage pour stocker les employés favoris, ainsi que par l'intermédiaire de la logique côté client définie dans le JavaScript.

AJOUT EN FAVORIS

Lorsqu'on accède à la page people.html, une liste d'employés est affichée, et chaque employé possède une icône d'étoile (représentée par `<i class="fa-star">`). Chaque étoile est cliquable et nous permet de marquer ou de retirer l'employé des favoris. Le JavaScript associé à cette icône détecte chaque clic et, en fonction de l'état actuel de l'étoile (grise ou jaune), il ajoute ou retire l'ID de l'employé dans le localStorage. L'état de l'étoile (favori ou non) est mis à jour immédiatement dans l'interface, sans nécessiter un rechargement de la page.

```
function toggleFavorite(starElement) {
    const employeeId = parseInt(starElement.getAttribute('data-employee-id'));
    let favorites = JSON.parse(localStorage.getItem('favorites')) || [];

    // Si l'employé est déjà dans les favoris, on le retire, sinon on l'ajoute
    if (favorites.includes(employeeId)) {
        favorites = favorites.filter(id => id !== employeeId); // Retirer l'ID
        starElement.classList.remove('s');
        starElement.classList.add('r');
        starElement.style.color = 'gray';
    } else {
        favorites.push(employeeId); // Ajouter l'ID
        starElement.classList.remove('r');
        starElement.classList.add('s');
        starElement.style.color = '#ffce33';
    }

    // Mettre à jour le localStorage
    localStorage.setItem('favorites', JSON.stringify(favorites));
}
```

Quand nous naviguons vers la page favorites.html, un autre script JavaScript récupère les employés favoris depuis le localStorage en utilisant les IDs enregistrés. Ce script va ensuite parcourir la liste des employés et les afficher dans la section des favoris si l'ID d'un employé est présent dans le stockage local. Si aucun favori n'est trouvé, un message nous indique qu'il n'y a pas d'employé favori. De plus, chaque employé favori possède à nouveau une icône d'étoile qui, lorsqu'on clique dessus, permet de retirer l'employé des favoris (en mettant à jour le localStorage et en réaffichant la liste des favoris)

```
function toggleFavorite(starElement) {
    let employeeId = starElement.getAttribute('data-employee-id');
    let favorites = JSON.parse(localStorage.getItem('favorites')) || [];

    const index = favorites.indexOf(parseInt(employeeId));

    if (index !== -1) {
        // Si l'employé est déjà dans les favoris, on le retire
        favorites.splice(index, 1);
        starElement.classList.remove('s');
        starElement.classList.add('r');
        starElement.style.color = 'gray'; // Modifier la couleur de l'étoile
    } else {
        // Sinon, on l'ajoute aux favoris
        favorites.push(parseInt(employeeId));
        starElement.classList.remove('r');
        starElement.classList.add('s');
        starElement.style.color = 'yellow'; // Modifier la couleur de l'étoile
    }

    // Sauvegarder les favoris dans localStorage
    localStorage.setItem('favorites', JSON.stringify(favorites));

    // Mettre à jour les favoris affichés
    displayFavorites();
}
```

PAGE MAILS.HTML

Après avoir présenté en détail le script de peuplement et exposé une vue d'ensemble du projet, nous allons désormais nous concentrer sur l'analyse précise du fonctionnement de la page mails.html.

1. Structure Générale du Template

Le fichier débute par l'héritage d'un template de base via la directive Django '{% extends "base.html" %}'. Cela permet d'assurer une cohérence visuelle et structurelle à l'ensemble de l'application dans le template de base.

La page redéfinit ensuite plusieurs blocs spécifiques :

- Bloc 'title'

Le titre de la page est défini par '{% block title %}Enron | Mails{% endblock %}', ce qui permet d'afficher un titre contextuel dans l'onglet du navigateur.

- Bloc 'navbar'

Ce bloc personnalise la barre de navigation pour la page "Mails". On y retrouve une liste d'éléments '- ' comportant des liens vers les différentes sections de l'application (Accueil, Personnes, Statistiques, Favoris). L'item correspondant aux mails est mis en évidence (classe 'text-red-700') afin d'indiquer à l'utilisateur la page active.

```
% block navbar %
  # Définir quelle page est active #
  <li class="hover:text-red-700">
    <a href="{% url 'accueil' %}">Accueil</a>
  </li>
  <li class="hover:text-red-700">
    <a href="{% url 'personnes' %}">Personnes</a>
  </li>
  <li class="text-red-700">
    <a href="{% url 'mails' %}">Mails</a>
  </li>
  <li class="hover:text-red-700">
    <a href="{% url 'statistiques' %}">Statistiques</a>
  </li>
  <li class="hover:text-red-700">
    <a href="{% url 'favoris' %}">Favoris</a>
  </li>
% endblock %
```

Figure 28: Bloc navbar

- Bloc 'content'

L'ensemble du contenu spécifique à la page est placé dans ce bloc. Il est composé d'éléments HTML et de scripts JavaScript dédiés à l'affichage et la gestion dynamique des mails.

2. La Zone de Contenu Principal

2.1. Container Principal

La structure principale de la page est enveloppée dans un '`<div>`' avec des classes utilitaires (Tailwind CSS) telles que '`container max-w-[1280px] mx-auto p-5`', garantissant un affichage centré et une largeur maximale définie.

2.2. Titre et En-tête

Un titre "Liste des mails" est affiché dans une balise '`<div>`' avec une typographie mise en valeur (classe '`text-2xl font-semibold text-gray-900 mb-6`').

3. Le Formulaire de Recherche

Un composant essentiel de cette page est le formulaire de recherche, permettant à l'utilisateur de filtrer les mails selon différents critères :

- Structure du formulaire :

Le formulaire (`id="search-form"`) est défini avec la classe '`flex gap-4 items-end`', assurant une disposition en ligne des éléments.

- Sélecteur de type de recherche :

Un '`<select>`' (`id="search-type"`) offre plusieurs options telles que "Tout", "ID", "Filepath", "Subject", "Message" et "Date". Ce choix conditionne le type de recherche que l'utilisateur souhaite effectuer.

- Champ de recherche textuelle :

Lorsque le type de recherche est autre que "Date", un '`<input type="text">`' (`id="search-input"`) est affiché pour permettre la saisie du terme recherché.

- Recherche par date :

Pour une recherche basée sur la date, une section supplémentaire (div avec `id="date-search"`) est proposée. Elle contient deux champs de saisie (`id="start-date"` et `id="end-date"`) activés par Flatpickr pour sélectionner respectivement la date de début et la date de fin.

- Bouton de validation :

Le bouton "Rechercher" déclenche la soumission du formulaire, permettant de recharger dynamiquement la liste des mails selon les paramètres définis.

4. Affichage Dynamique des Mails

Après le formulaire, le template prévoit une zone destinée à l'affichage des mails :

- Conteneur des mails :

Le '`<div id="email-container">`' servira de conteneur dans lequel chaque mail, sous forme d'un élément cliquable, sera inséré dynamiquement par JavaScript.

- Loader :

Un élément '`<div id="loader">`' affiche un message "Chargement..." afin d'indiquer à l'utilisateur que le chargement des données est en cours.

5. Gestion de l'Affichage du Détail d'un Mail (Modal)

Notre page mails intègre également une structure modale pour afficher le contenu détaillé d'un mail :

- Overlay et Modal :

Le '`<div id="modal-overlay">`' représente l'arrière-plan semi-transparent, initialement masqué, qui recouvre la page lors de l'ouverture du détail d'un mail.

À l'intérieur se trouve le '`<div id="email-modal">`' qui contient le contenu du mail et un en-tête comportant deux icônes :

- Téléchargement () : Permet de télécharger le mail sous format texte.

- Fermeture (x) : Permet de fermer la modale.

- Contenu de la modale :

Le '`<div id="modal-content">`' est destiné à recevoir le détail complet du mail (ID, filepath, date, sujet, et message) une fois celui-ci chargé via une requête asynchrone.

6. Le Script JavaScript

L'ensemble de la dynamique de la page est géré par un script JavaScript intégré en fin de template. Voici les points essentiels :

6.1. Initialisation et Variables Globales

- Variables d'état :

Des variables telles que `lastDate`, `loading`, `noMoreEmails` et `currentSearchParams` permettent de gérer l'état du chargement et les paramètres de recherche.

- Initialisation de Flatpickr :

Les champs de date sont initialisés avec une configuration précise (format de date "Y-m-d", bornes minimales et maximales).

6.2. Gestion de l'Affichage des Champs de Recherche

Un écouteur d'événement sur le sélecteur de type de recherche ajuste dynamiquement l'affichage :

- Si le type sélectionné est "date", le champ de texte est masqué et le bloc de recherche par date est affiché.
- Sinon, c'est l'inverse qui se produit.

6.3. Soumission du Formulaire de Recherche

Lors de la soumission du formulaire :

- La fonction empêche le comportement par défaut de la soumission.
- Les paramètres de recherche sont collectés (type, requête textuelle, dates de début et de fin).
- La liste des mails affichés est réinitialisée et une nouvelle requête est lancée par l'appel à la fonction `loadEmails()`.

6.4. Chargement et Affichage des Mails

- Fonction `loadEmails()` :

Cette fonction asynchrone effectue une requête HTTP vers l'endpoint `/api/emails/` en ajoutant les paramètres de recherche et éventuellement la date du dernier mail chargé (`lastDate`) pour paginer les résultats.

- Si la réponse est positive, le JSON reçu est parcouru et chaque mail est transformé en un élément HTML par la fonction `createEmailItem(email)`.
- La variable `lastDate` est mise à jour pour permettre le chargement incrémental.
- En cas d'absence de nouveaux mails, un message adapté est affiché.

- Création d'un élément mail (`createEmailItem(email)`):

Un '`<div>`' est généré pour chaque mail avec une structure qui inclut :

- Le sujet (ou un texte par défaut si absent).

- Le nom de l'expéditeur et la date du mail, formatée de manière lisible.

- Un écouteur d'événement sur le clic qui appelle la fonction `openEmailDetail(email.id)` pour ouvrir la modale avec les détails du mail.

6.5. Affichage du Détail d'un Mail

- Fonction `openEmailDetail(emailId)`:

Elle effectue une requête vers l'API pour obtenir les détails du mail sélectionné.

En cas de succès, le contenu détaillé est injecté dans le bloc de la modale (`modal-content`), et le mail est stocké dans l'attribut `data-email` de l'icône de téléchargement pour une éventuelle sauvegarde en .txt.

6.6. Téléchargement d'un Mail en Format Texte

- Fonction `downloadEmailAsTxt()`:

Cette fonction récupère les données du mail depuis l'attribut `data-email`, construit un contenu textuel en concaténant les informations (ID, filepath, date, sujet et message), puis crée dynamiquement un lien de téléchargement pour générer et déclencher le téléchargement du fichier.

6.7. Fermeture de la Modale et Interactions Diverses

- Fonction `closeModal()`:

Elle masque la modale et rétablit le défilement normal de la page.

- Gestion par Intersection Observer :

Un observateur surveille l'élément `loader` afin de déclencher automatiquement le chargement de nouveaux mails lorsque celui-ci entre dans le champ de vision de l'utilisateur.

- Bouton "Back to Top" :

Un écouteur d'événement sur le défilement de la fenêtre permet d'afficher ou masquer le bouton de retour en haut en fonction de la position verticale, et d'effectuer un défilement fluide lors du clic.

PAGE STATISTICS.HTML

Après avoir collecté et traité l'ensemble des données issues de l'affaire Enron, il devient essentiel de les analyser sous différents angles. La création d'une page statistique offre ainsi un moyen intuitif de visualiser les tendances, de détecter les anomalies et de mieux comprendre la répartition et l'évolution des communications. Ce type de présentation permet de transformer un volume massif de données en informations exploitables et facilement interprétables, facilitant ainsi la prise de décisions éclairées.

Vue Globale de la Page

- Affichage Conditionnel :
 - Avant d'afficher les graphiques, le template vérifie s'il existe des données dans la base. Si ce n'est pas le cas (condition `if error == True`), un message d'erreur apparaît, invitant l'utilisateur à exécuter le script de peuplement (avec des extraits de commandes à copier).
- Disposition des Graphiques :
 - Lorsque des données sont disponibles, une grille responsive présente cinq graphiques dans des conteneurs interactifs. Chaque conteneur, doté d'un effet de survol (agrandissement) et d'un comportement onclick, permet d'ouvrir une fenêtre modale pour une analyse plus approfondie.
- Fenêtre Modale :
 - La modale, cachée par défaut, s'active lors d'un clic sur un graphique. Elle offre une vue agrandie du graphique sélectionné et, pour certains graphiques, la possibilité de choisir la granularité (heure, jour, mois, année ou plage de dates) via des sélecteurs et des champs de saisie.
- Intégration de Chart.js :
 - La bibliothèque Chart.js (version 3.9.1) est utilisée pour générer et configurer les graphiques. Un script JavaScript centralise l'initialisation, la configuration et la mise à jour dynamique des graphiques en fonction des interactions utilisateur.

Notre page de statistiques intègre cinq graphiques interactifs, chacun conçu pour mettre en lumière un aspect spécifique des données :

- Graphique 1 : Répartition des mails par dossier
- Permet d'identifier les dossiers les plus actifs en affichant la distribution des mails selon leur emplacement.

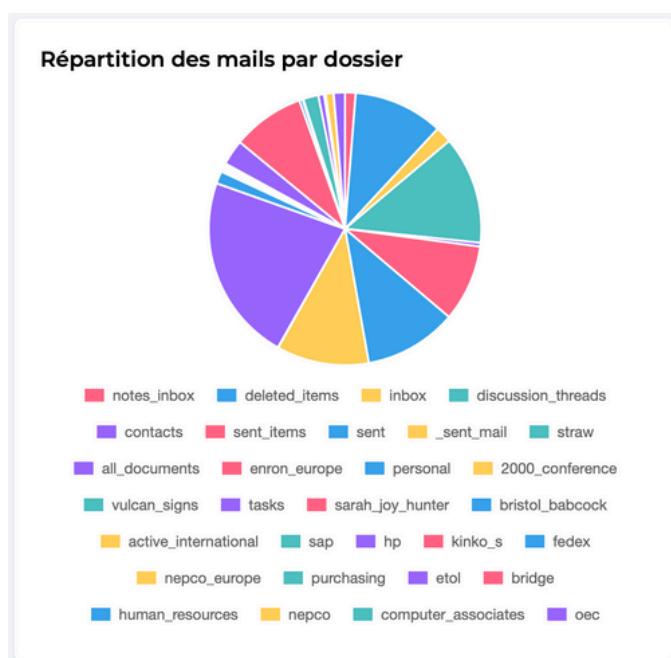


Figure 29 : Graphique 1, Répartition des mails par dossier

Graphique 2 : Évolution du nombre de mails envoyés dans le temps
 Offre une vue chronologique du volume des mails, facilitant l'analyse des tendances et des pics d'activité.



Figure 30 : Graphique 2 : Évolution du nombre de mails envoyés dans le temps

Graphique 3 : Originaux vs Réponses

Compare le nombre de mails initiaux aux réponses, fournissant une indication claire de la dynamique conversationnelle.

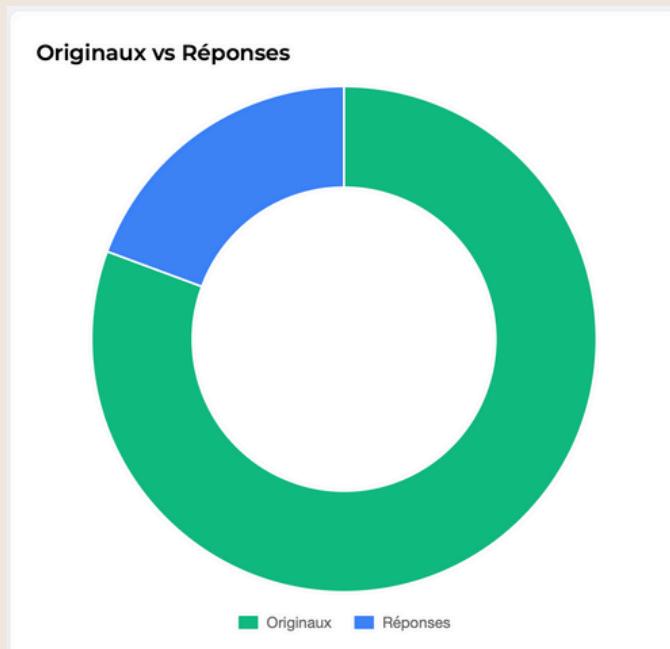


Figure 31 : Graphique 3 : Originaux vs Réponses

Graphique 4 : Activité par expéditeur

Met en avant les expéditeurs les plus actifs en affichant le nombre de mails envoyés par chacun.

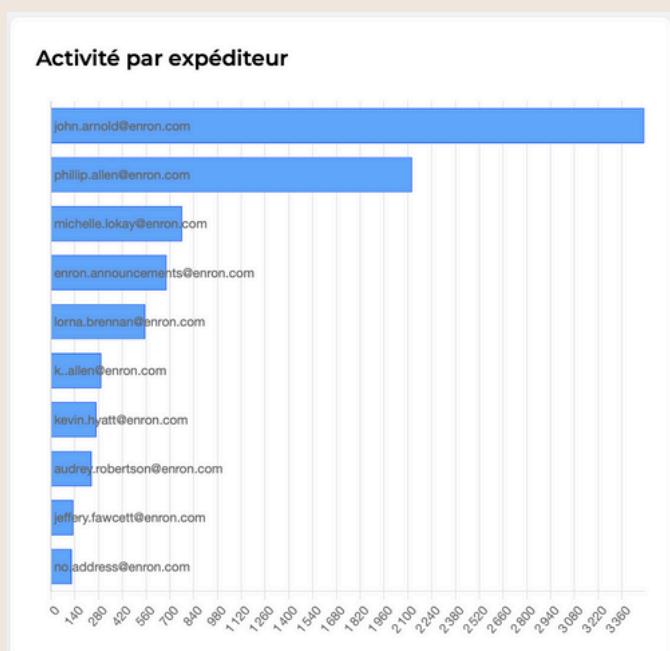


Figure 32 : Graphique 4 : Activité par expéditeur

Graphique 5 : Longueur des messages et des sujets

Analyse la longueur moyenne des messages et des sujets pour offrir un aperçu de la taille des communications.



Figure 33 : Graphique 5 : Longueur des messages et des sujets

Fenêtre modale et sélection de granularité:

Un des points forts de cette page est l'intégration d'une fenêtre modale destinée à approfondir l'analyse d'un graphique sélectionné. Lorsque l'utilisateur clique sur un graphique, une modale s'ouvre et affiche une version agrandie du graphique choisi. Pour le graphique représentant l'évolution temporelle, l'interface offre également une option de sélection de la granularité (heure, jour, mois, année ou plage de dates), permettant ainsi d'affiner l'analyse en fonction de la période souhaitée. Des champs de saisie pour sélectionner une date de début et une date de fin apparaissent lorsque l'option « Plage de dates » est sélectionnée.

Logique JavaScript et configuration de Chart.js:

Le script JavaScript intégré gère l'initiation et la mise à jour des graphiques à l'aide de la bibliothèque Chart.js (version 3.9.1). Plusieurs points sont à souligner :

- Initialisation des données : Les données des graphiques (telles que les labels et les valeurs) sont passées depuis le contexte Django, puis converties depuis un format JSON sécurisé grâce à une fonction de parsing.

- Fonction d'initialisation des graphiques : La fonction initChart crée les instances de graphique en récupérant le contexte 2D du canevas et en appliquant une configuration spécifique en fonction du type de graphique (pie, line, doughnut, horizontalBar ou bar).
- Configuration centralisée : La fonction getChartConfig regroupe les options communes (réactivité, légendes, tooltips, etc.) et définit des configurations particulières pour chaque type de graphique. Par exemple, pour les graphiques circulaires, des couleurs prédéfinies et un découpage central (cutout) sont appliqués pour le type « doughnut ».
- Mise à jour dynamique : La fonction updateGranularity réalise une requête asynchrone afin de récupérer de nouvelles données en fonction de la granularité sélectionnée par l'utilisateur. Elle met à jour en temps réel les graphiques affichés, tant dans la page principale que dans la fenêtre modale.
- Gestion de la modale : Les fonctions openModal et closeModal gèrent l'affichage et la destruction des instances de graphique dans la fenêtre modale, assurant ainsi une navigation fluide et un rafraîchissement des données lors des interactions utilisateur.

Enfin, on retrouve tout une partie de Tailwind CSS pour garantir que les conteneurs de graphiques disposent d'une hauteur minimale et maximale adaptée à différentes résolutions d'écran. Cette stylisation inclut également des réglages pour la fenêtre modale afin de garantir une bonne lisibilité et une interactivité optimale, notamment sur les appareils mobiles et les écrans plus petits.

CONCLUSION

Ce projet a permis de mettre en place une application web performante de visualisation des données massives liées à l'affaire Enron, en s'appuyant sur Django pour la gestion des données et sur des outils de visualisation pour leur analyse.

L'un des défis majeurs a été la gestion et l'intégration d'un volume de données considérable, avec plus de 517 401 fichiers et 3 499 dossiers à traiter.

Pour répondre à cette contrainte, nous avons développé un script de peuplement optimisé permettant d'insérer efficacement les employés et les mails dans la base de données. Grâce à des techniques comme l'utilisation d'un cache en mémoire, l'insertion en masse, et la gestion robuste des erreurs, nous avons pu améliorer significativement la performance du traitement.

L'interface de l'application a également été pensée pour offrir une navigation fluide et intuitive, avec des fonctionnalités avancées comme la recherche de mails, l'affichage des interactions entre employés, et l'analyse des tendances grâce aux graphiques interactifs.

Ce projet illustre ainsi la puissance de Django pour le traitement et la visualisation de données massives, tout en mettant en lumière l'importance d'une approche méthodique pour garantir l'efficacité, la robustesse et la lisibilité du code.

Il constitue une base solide pour d'éventuelles évolutions futures, notamment avec l'intégration d'outils d'analyse avancés comme l'intelligence artificielle ou l'apprentissage automatique pour détecter des schémas de fraude encore plus précis.