**Roadmap de Développement**

**1. Recherche et Planification**

* **Choix des technologies**
  + Sélection du langage de programmation, du framework backend et frontend, de la base de données.
* **Architecture du système**
  + Élaboration d'un modèle d'architecture pour le backend, le frontend, et la base de données.

**2. Développement du Backend**

* **Mise en place de la base de données**
  + Conception du schéma de la base de données pour les contacts, les campagnes, les séquences de mails, etc.
  + Implémentation des tables et des relations dans la base de données.
* **Développement des APIs**
  + API pour l'importation de contacts : Charger un fichier (. Txt, csv, Xlxs) et stocker les données dans la base de données.
  + API pour l'envoi d'emails : API pour envoyer des emails via un service SMTP.
  + API pour la réception d'emails : Intégration d'IMAP pour recevoir des emails.

**3. Développement du Frontend**

* **Interface Utilisateur**
  + Créer des wireframes pour les pages principales comme le tableau de bord, la page d'importation de contacts, l'éditeur de campagne, etc.
  + Développer ces interfaces avec le framework frontend choisi.
* **Intégration des fonctionnalités**
  + Intégration avec les APIs du backend pour l'importation de contacts et la création de campagnes.

**4. Intégration SMTP/IMAP**

* **Configuration SMTP/IMAP**
  + Permettre aux utilisateurs de saisir leurs paramètres SMTP et IMAP.
  + Intégrer ces paramètres dans les APIs d'envoi et de réception d'email.

**5. Éditeur de Texte et Interface de Campagne**

* **Intégration de l'Éditeur de Texte**
  + Intégrer un éditeur de texte riche pour la rédaction des emails.
* **Fonctionnalités de la Campagne**
  + Ajouter la possibilité de créer, éditer et supprimer des campagnes.
  + Implémenter les séquences de mails et les paramètres de la campagne.

**6. Tracking et Analytics**

* **Suivi des Emails**
  + Intégrer des pixels de suivi dans les emails envoyés pour suivre les ouvertures, les clics, etc.
* **Tableau de Bord Analytics**
  + Créer un tableau de bord pour afficher les statistiques et les métriques clés.

**7. Tests et QA**

* **Tests Unitaires**
  + Écrire des tests pour chaque fonctionnalité.
* **Tests d'Intégration**
  + Tests pour vérifier si les différentes parties de l'application fonctionnent ensemble comme prévu.
* **Tests Utilisateur**
  + Tests manuels pour vérifier l'expérience utilisateur.

**8. Lancement Beta**

* Lancer une version beta pour un groupe restreint d'utilisateurs et recueillir des feedbacks.

**9. Feedback et Itération**

* Analyser les feedbacks et faire les ajustements nécessaires.

**10. Lancement Officiel**

* Lancement au grand public avec marketing et support.

**11. Mises à Jour et Maintenance**

* Déployer des nouvelles fonctionnalités, des correctifs de bugs, et des améliorations basées sur le feedback et les données d'utilisation.

1. **Recherche et Planification ajustée à votre équipe**

**(VERSION PYTHON/JS)**

**Choix des Technologies**

1. **Langage de Programmation**
   * **Backend**: Python (Django ou Flask)
   * **Frontend**: JavaScript (React.js)
   * **Raison**: Ces langages tirent parti de l'expertise de votre équipe, ce qui pourrait accélérer le développement.
2. **Framework Backend**
   * **Choix**: Django (Python)
     + **Raison**: Django est extrêmement robuste, dispose d'un panneau administratif intégré et d'un ORM puissant, ce qui est idéal pour les applications CRUD.
3. **Framework Frontend**
   * **Choix**: React.js
     + **Raison**: La flexibilité, la facilité d'utilisation des composants et la grande communauté en font un excellent choix.
4. **Base de Données**
   * **Choix**: PostgreSQL
     + **Raison**: Compatible avec Django, robuste et offre des fonctionnalités avancées.
5. **Serveur de Messagerie**
   * **Choix**: SMTP via SendGrid
     + **Raison**: Facile à intégrer avec Python et offre des fonctionnalités comme le suivi d'email.
6. **Librairies et Outils**
   * **Backend**:
     + **celery** pour les tâches asynchrones.
     + **djangorestframework** pour les API RESTful.
   * **Frontend**:
     + **axios** pour les requêtes HTTP.
     + **redux** pour la gestion de l'état.
7. **APIs à utiliser**
   * Stripe API pour les paiements, si nécessaire.

**Architecture du Système**

1. **Backend**
   * **Modèle MTV (Modèle-Template-Vue) de Django**
     + **Raison**: Naturellement adapté à Django et facilite la séparation des préoccupations.
2. **Frontend**
   * **Architecture basée sur Composants**
     + Utiliser le Hook API et le Context API pour la gestion de l'état.
3. **Base de Données**
   * **Modélisation des Données**
     + Utiliser le modèle de données Django pour définir les tables et les relations.
4. **Serveur et Déploiement**
   * Utiliser Docker pour la conteneurisation.
   * Utiliser Docker Compose pour orchestrer les services localement.
5. **CI/CD**
   * Utiliser GitHub Actions ou Jenkins avec des scripts de build et de déploiement spécifique à Python et JavaScript.
6. **Sécurité**
   * Mettre en œuvre l'authentification JWT en utilisant les packages Python dédiés.
   * Utiliser HTTPS.
   1. **Recherche et Planification Ultra-Détaillée**

Choix des Technologies

1. **Langage de Programmation**
   * **Backend**: Python avec Django comme framework
   * **Frontend**: JavaScript avec React.js comme framework

Framework Backend - Django

1. **Modules et Packages essentiels**
   * **Django REST Framework**: Pour créer une API RESTful robuste.
   * **Celery + Redis**: Pour les tâches asynchrones et les files d'attente.
   * **django-cors-headers**: Pour gérer les CORS.
   * **django-environ**: Pour gérer les variables d'environnement.
   * **django-allauth**: Pour l'authentification.
2. **Structure du projet**
   * Découpage en plusieurs apps Django (**users**, **campaigns**, **emails** etc.)
3. **APIs à Intégrer**
   * **SendGrid**: Pour l'envoi d'email.
   * **Stripe**: Pour les paiements.
   * **OAuth Providers**: Pour l'authentification via Google, Facebook, etc.
   * **Cloudinary**: Pour le stockage d'images et de médias.

Framework Frontend - React.js

1. **Bibliothèques essentielles**
   * **axios**: Pour les requêtes HTTP.
   * **redux + redux-thunk**: Pour la gestion de l'état.
   * **react-router**: Pour le routage.
   * **styled-components**: Pour le CSS-in-JS.
2. **Structure du projet**
   * Découpage en plusieurs composants (**Header**, **Footer**, **CampaignList**, etc.)
   * Utilisation du dossier **assets** pour les images et du dossier **utils** pour les fonctions utilitaires.
3. **APIs à Intégrer**
   * **Mapbox ou Google Maps**: Si vous avez besoin de fonctionnalités liées à la localisation.

Base de Données - PostgreSQL

1. **Tables Principales**
   * **Users**
   * **Campaigns**
   * **Emails**
   * **Contacts**
2. **Relations**
   * Clés étrangères, index, et possiblement des tables de jointure pour des relations many-to-many.
3. **ORM**
   * Utilisation de l'ORM Django pour les requêtes et les migrations.

Serveur et Déploiement

1. **Docker + Docker Compose**
   * Chaque service (Django, React, PostgreSQL) dans son propre conteneur.
2. **CI/CD**
   * Utilisation de GitHub Actions pour tester et déployer automatiquement le code.
3. **Cloud Providers**
   * AWS, Azure, ou Google Cloud pour le déploiement.

Sécurité

1. **Authentification**
   * Utilisation de JWT avec des tokens d'accès et de rafraîchissement.
2. **HTTPS**
   * Utilisation de certificats SSL pour sécuriser la communication.
3. **Rate Limiting et Throttling**
   * Pour limiter les requêtes API par utilisateur.
4. **Développement du Backend**

**Mise en place de la base de données**

1. **Choix de la Base de Données**: PostgreSQL, étant donné sa flexibilité, sa fiabilité et sa compatibilité avec Django.

Pourquoi PostgreSQL?

1. **Flexibilité**: PostgreSQL supporte des types de données JSON et JSONB, ce qui vous permet de stocker des données semi-structurées.
2. **Fiabilité**: C'est une base de données éprouvée qui est utilisée par de grandes entreprises.
3. **Compatibilité**: Django a un excellent support pour PostgreSQL, ce qui facilite l'intégration.
4. **Installation et Configuration**: Utilisation de Docker pour encapsuler la base de données PostgreSQL. Cela permet une portabilité et une facilité de déploiement.

Étapes à suivre:

1. **Installation de Docker**: Allez sur le site Web de Docker et suivez les instructions pour installer Docker Desktop sur votre machine.
   * Site Web: [Docker](https://www.docker.com/products/docker-desktop)
2. **Configuration du fichier docker-compose.yml**: Créez un fichier appelé **docker-compose.yml** dans le répertoire racine de votre projet Django.



**Démarrage du conteneur**: Ouvrez un terminal, accédez au répertoire où se trouve votre fichier **docker-compose.yml** et exécutez la commande suivante:

**docker-compose up -d**

1. **Connexion à Django**: Utilisation du fichier **settings.py** dans Django pour configurer la connexion à la base de données PostgreSQL.

**Installation du package psycopg2**: Ce package est nécessaire pour que Django puisse interagir avec PostgreSQL.

**pip install psycopg2-binary**

**Configuration du fichier settings.py**: Ouvrez votre fichier **settings.py** dans votre projet Django et trouvez la section **DATABASES**. Modifiez-le pour qu'il ressemble à ceci:



**Migration de la base de données**: Une fois que le fichier **settings.py** est configuré, vous pouvez créer la base de données en exécutant les migrations Django :

**python manage.py makemigrations**

**python manage.py migrate**

**4. Hébergement de la Base de Données**

Pour héberger votre base de données, vous pouvez choisir parmi plusieurs fournisseurs tels que:

1. **AWS RDS**: Amazon Web Services offre un service de base de données relationnelle qui supporte PostgreSQL.
   * Site Web: [AWS RDS](https://aws.amazon.com/rds/)
2. **Google Cloud SQL**: Un autre choix viable pour le stockage de bases de données PostgreSQL.
   * Site Web: [Google Cloud SQL](https://cloud.google.com/sql)
3. **Heroku**: Une solution plus simple et moins technique.
   * Site Web: [Heroku](https://www.heroku.com/postgres)

**Conception du schéma de la base de données**

1. Tables Principales et leurs Colonnes

**Table: Users**

* **id**: Clé primaire, entier auto-incrémenté.
* **username**: Chaîne de caractères, unique.
* **email**: Chaîne de caractères, unique.
* **password\_hash**: Chaîne de caractères, hashé.
* **created\_at**: Timestamp.
* **updated\_at**: Timestamp.

**Table: Campaigns**

* **id**: Clé primaire, entier auto-incrémenté.
* **name**: Chaîne de caractères.
* **user\_id**: Clé étrangère vers la table Users.
* **created\_at**: Timestamp.
* **updated\_at**: Timestamp.

**Table: Emails**

* **id**: Clé primaire, entier auto-incrémenté.
* **subject**: Chaîne de caractères.
* **body**: Texte.
* **campaign\_id**: Clé étrangère vers la table Campaigns.
* **created\_at**: Timestamp.
* **updated\_at**: Timestamp.

**Table: Contacts**

* **id**: Clé primaire, entier auto-incrémenté.
* **email**: Chaîne de caractères.
* **first\_name**: Chaîne de caractères.
* **last\_name**: Chaîne de caractères.
* **phone\_number**: Chaîne de caractères.
* **user\_id**: Clé étrangère vers la table Users.
* **created\_at**: Timestamp.
* **updated\_at**: Timestamp.

**Table de jointure: Campaigns\_Contacts**

* **campaign\_id**: Clé étrangère vers la table Campaigns.
* **contact\_id**: Clé étrangère vers la table Contacts.

2. Relations

* **Users à Campaigns**: Relation One-to-Many. Un utilisateur peut avoir plusieurs campagnes.
* **Campaigns à Emails**: Relation One-to-Many. Une campagne peut avoir plusieurs emails.
* **Users à Contacts**: Relation One-to-Many. Un utilisateur peut avoir plusieurs contacts.
* **Campaigns à Contacts**: Relation Many-to-Many. Une campagne peut avoir plusieurs contacts et un contact peut appartenir à plusieurs campagnes.

3. Attributs Supplémentaires et Indices

* Utilisez des indices sur les colonnes fréquemment recherchées comme **email** dans la table Contacts et **username** dans la table Users.
* Utilisez des types de données spécifiques pour le stockage efficace des données. Par exemple, utilisez le type de données "email" pour stocker les adresses e-mail si votre base de données le supporte.

Exemple de création de table en SQL (PostgreSQL)

Pour la table Users, la commande SQL pourrait ressembler à ceci :



Vous pouvez créer des tables similaires pour Campaigns, Emails, Contacts, et la table de jointure Campaigns\_Contacts en utilisant des commandes SQL similaires.

Cependant, si vous utilisez Django, la plupart de ces tables seront créées automatiquement lorsque vous exécuterez les migrations, à condition que vous ayez bien configuré vos modèles Django.

**Implémentation des tables et des relations dans la base de données**

1. **Modèles Django**: Création de modèles Django pour chaque table.

**Modèle Users**



Modèle Campaigns



Modèle Emails



Modèle Contacts



1. **Migrations**: Utilisation de la commande **python manage.py makemigrations** pour créer les fichiers de migration et **python manage.py migrate** pour appliquer ces migrations.

Après avoir créé vos modèles, il est temps de générer les fichiers de migration. Ouvrez votre terminal et naviguez vers le répertoire de votre projet Django. Exécutez la commande suivante :

**python manage.py makemigrations**

Cela générera les fichiers de migration dans le dossier **migrations** de chaque application. Ensuite, pour appliquer ces migrations et créer les tables dans votre base de données, exécutez :

**python manage.py migrate**

1. **ORM Django**: Utilisation de l'ORM Django pour les opérations CRUD (Create, Read, Update, Delete).

Django ORM vous permet de réaliser des opérations CRUD sans écrire de SQL. Voici quelques exemples :

Créer un nouvel utilisateur

****

Lire des utilisateurs

**all\_users = User.objects.all()**

Pour obtenir un utilisateur spécifique :

**user = User.objects.get(id=1)**

Mettre à jour un utilisateur

**user = User.objects.get(id=1)**

**user.email = "new\_email@example.com"**

**user.save()**

Supprimer un utilisateur

**user = User.objects.get(id=1)**

**user.delete()**

**Développement des APIs**

1. **API pour l'importation de contacts**:
   * Utilisation de Python **csv** ou **pandas** pour lire les fichiers **.txt**, **.csv**, **.xlxs**.
   * Stockage des données dans la table **Contacts**.

Vous pouvez utiliser la bibliothèque **csv** standard de Python ou la bibliothèque **pandas** pour lire les fichiers .txt, .csv, .xlsx.

**Avec la bibliothèque Pandas de Python :**



1. **API pour l'envoi d'e-mails**:
   * Utilisation de la bibliothèque Python **smtplib** ou d'un service comme AWS SES ou SendGrid pour l'envoi d'e-mails.
   * Mise en file d'attente des e-mails à envoyer en utilisant Celery + Redis pour les tâches asynchrones.

Utilisation de smtplib

Vous pouvez utiliser la bibliothèque **smtplib** de Python pour envoyer des e-mails.



Utilisation de AWS SES avec boto3

Si vous utilisez AWS SES pour l'envoi d'e-mails, vous pouvez utiliser la bibliothèque **boto3**.



1. **API pour la réception d'e-mails**:
   * Utilisation de la bibliothèque Python **imaplib** pour intégrer IMAP.
   * Éventuellement, utilisation de webhooks si le fournisseur d'e-mail le supporte.

Pour recevoir des e-mails, vous pouvez utiliser la bibliothèque **imaplib**.

(CLICKER SUR L’OBJET (texte en jaune) CAR LE CODE FAIS PLUSIEURS PAGE)



* Connexion au serveur de messagerie: Le code utilise **imaplib** pour se connecter au serveur de messagerie Gmail via IMAP.
* Récupération des e-mails: Les ID de tous les e-mails dans la boîte de réception sont récupérés. Dans cet exemple, seuls les 10 derniers e-mails sont traités pour des raisons de démonstration.
* Décodage des e-mails: Chaque e-mail est décodé en une chaîne de caractères afin que les informations puissent être extraites.
* Extraction des informations: Le sujet, l'expéditeur, le destinataire, la date et le corps de chaque e-mail sont extraits.
* Stockage des e-mails: Une instance de la classe **Email** est créée pour chaque e-mail et la méthode **save()** est appelée. Cette méthode est une simulation de la manière dont les données seraient stockées dans une base de données.
* Envoi de réponses automatiques: Une réponse automatique est envoyée à l'expéditeur de chaque e-mail en utilisant SMTP.
* Classification des e-mails: Les e-mails sont classés en fonction de mots-clés dans le sujet ("urgent", "important", etc.) et la classification est affichée.
* Sécurité SSL: Le code utilise SSL pour sécuriser la connexion SMTP lors de l'envoi des e-mails.

Notez que pour que la fonction **send\_automatic\_reply** fonctionne, vous devrez remplacer **your\_email@gmail.com** et **your\_password** par vos informations d'identification Gmail.

Ce code est un point de départ et peut être étendu en fonction de vos besoins spécifiques. Vous pouvez ajouter des fonctionnalités plus avancées de classification, de tri, etc. Vous pouvez également remplacer la classe **Email** simulée par le modèle Django réel pour le stockage dans votre base de données.

**Bibliothèques et Modules Supplémentaires**

1. **Django Extensions**: Pour des fonctionnalités supplémentaires comme les graphes de modèles, des commandes shell supplémentaires, etc.

**Utilisation:**

Django Extensions ajoute un ensemble de fonctionnalités supplémentaires pour Django, notamment des graphiques de modèle, des commandes shell utiles, et plus encore.

**Installation:**

Pour installer Django Extensions, utilisez la commande pip :

**pip install django-extensions**

**Configuration:**

Ajoutez-le à votre **INSTALLED\_APPS** dans **settings.py** :

**INSTALLED\_APPS = [**

**...**

**'django\_extensions',**

**...**

**]**

**Exemples de Commandes Utiles:**

* **./manage.py show\_urls**: Affiche toutes les URL de votre application.
* **./manage.py shell\_plus**: Ouvre un shell interactif avec tous vos modèles Django déjà importés.

1. **Django Filter**: Pour ajouter des fonctionnalités de filtrage complexes aux API REST.

**Utilisation:**

Django Filter vous permet d'ajouter facilement des fonctionnalités de filtrage complexes à vos API REST.

**Installation:**

**pip install django-filter**

**Configuration:**

Ajoutez-le à votre **INSTALLED\_APPS** :

**INSTALLED\_APPS = [**

**...**

**'django\_filters',**

**...**

**]**

Et dans **settings.py**, ajoutez :

REST\_FRAMEWORK = {

...

'DEFAULT\_FILTER\_BACKENDS': ['django\_filters.rest\_framework.DjangoFilterBackend'],

...

}

**Exemple de Code:**

Supposons que vous ayez un modèle **Contact** et que vous vouliez filtrer par **id** et **date**. Vous pouvez créer un **filters.py** :

**import django\_filters**

**class BookFilter(django\_filters.FilterSet):**

**class Meta:**

**model = Book**

**fields = ['id', 'date']**

Dans votre **views.py** :

from django\_filters.rest\_framework import DjangoFilterBackend

class BookViewSet(viewsets.ModelViewSet):

queryset = Book.objects.all()

serializer\_class = BookSerializer

filter\_backends = (DjangoFilterBackend,)

filterset\_class = BookFilter

1. **Django Cache**: Pour la mise en cache et améliorer les performances.

**Utilisation:**

Django Cache est utilisé pour la mise en cache afin d'améliorer les performances de votre application.

**Configuration:**

Dans **settings.py**, vous pouvez ajouter :

CACHES = {

'default': {

'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',

'LOCATION': '127.0.0.1:11211',

}

}

**Exemple de Code:**

Pour utiliser le cache dans une vue :

from django.core.cache import cache

def my\_view(request):

cache\_key = 'my\_key'

data = cache.get(cache\_key)

if data is None:

data = 'This is data'

cache.set(cache\_key, data, 60\*15) # Cache pendant 15 minutes

return HttpResponse(data)

Ces modules et bibliothèques sont très utiles et ajouteront beaucoup de fonctionnalités supplémentaires à votre projet Django, tout en améliorant également les performances.

**APIs à Intégrer (en + de celles mentionnées)**

* 1. RabbitMQ: Comme broker pour Celery

**Utilisation:**

RabbitMQ agit comme un courtier de messages pour Celery, facilitant la mise en file d'attente et la distribution des tâches asynchrones dans votre application.

**Installation:**

Vous pouvez utiliser Docker pour installer RabbitMQ:

docker run -d --hostname my-rabbit --name some-rabbit -p 5672:5672 rabbitmq:3

**Configuration:**

Dans votre fichier **settings.py**, ajoutez ou modifiez la configuration de Celery comme suit:

CELERY\_BROKER\_URL = 'amqp://localhost'

**Intégration:**

Intégrez RabbitMQ dans votre application en initialisant Celery dans votre fichier **\_\_init\_\_.py** de votre projet Django.

* 1. Elasticsearch: Si vous avez besoin de fonctionnalités de recherche avancées

**Utilisation:**

Elasticsearch est un moteur de recherche basé sur Lucene. Il fournit un moyen distribué, multi-locataire capable de rechercher des données en temps réel.

**Installation:**

Encore une fois, Docker est votre ami:

docker run --name elasticsearch -d -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" docker.elastic.co/elasticsearch/elasticsearch:7.10.0

**Configuration:**

Installez la bibliothèque Python Elasticsearch :

pip install elasticsearch

Puis configurez Elasticsearch dans votre projet Django. Vous pouvez créer un fichier **search\_indexes.py** pour gérer cela.

* 1. Swagger ou Postman: Pour la documentation et le test des API

**Utilisation:**

Ces outils vous aident à documenter vos API et à effectuer des tests d'API en temps réel.

**Swagger**

**Installation:**

Installez **drf-yasg** pour intégrer Swagger à Django :

pip install drf-yasg

**Configuration:**

Ajoutez **drf\_yasg** à votre **INSTALLED\_APPS** dans **settings.py** et configurez vos URL pour inclure les routes Swagger.

**Postman**

**Installation:**

Téléchargez et installez Postman à partir du [site officiel](https://www.postman.com/downloads/).

**Utilisation:**

Créez une nouvelle collection et ajoutez vos routes d'API pour les tests. Vous pouvez même écrire des scripts de test et automatiser vos tests d'API avec Postman.

Chacun de ces services et bibliothèques ajoute des fonctionnalités essentielles et des améliorations de performance à votre application, assurant que votre projet est robuste, évolutif et facile à maintenir.