Diapyr : Documentation technique

Diapyr Beta : Version : 0.5

Auteurs : Aïssa Pansan | Killian Van Seuningen | Celian Wirtz | Saad Roty

Edition : 08/04/2025

Table des matières

I – Introduction et Préambule

Nous avons le plaisir de vous partager cette documentation technique au nom de toute l'équipe de développement **Diapyr !** Le but de cette présente documentation est de pouvoir expliquer les choix techniques qui ont été utilisés pour réaliser Diapyr. Le but est aussi de proposer un document afin d'aider les autres développeurs à modifier notre projet et d'en assurer la maintenance.

Diapyr est une application de débats pyramidaux, basée sur la plateforme de collaboration en ligne **Zulip**. Diapyr propose une manière innovante de débattre à contre-courant des débats classiques avec son système de vote. Le principe est de répartir un grand nombre d'utilisateurs en plusieurs petits groupes de débat. De cette façon, chaque individu peut faire entendre son avis dans chaque groupe. Par la suite, dans chaque groupe sera rédigé une synthèse des débats internes et des représentants seront désignés afin de représenter la voix du groupe. Plusieurs petits groupes composés exclusivement de représentants sont alors formés. Ces groupes vont perpétuer une nouvelle phase de débat qui va fonctionner sur le même principe que la phase précédente. Ce mécanisme a lieu jusqu'à ce qu'il ne reste qu'un groupe chargé de prendre une décision finale basée sur toutes les synthèses et débats ayant eu lieu jusqu'ici.

Le but d'un tel processus est d'incorporer la vision de chaque participant en faisant en sorte que l'avis de chaque groupe est représenté dans un groupe de niveau supérieur. Cela est particulièrement crucial dans des débats de plusieurs centaines, voire plusieurs milliers de personnes, là où la voix de beaucoup pourrait être écartée au profit de la majorité, dans le cas d'un débat classique dans un vote ou un débat centralisé.

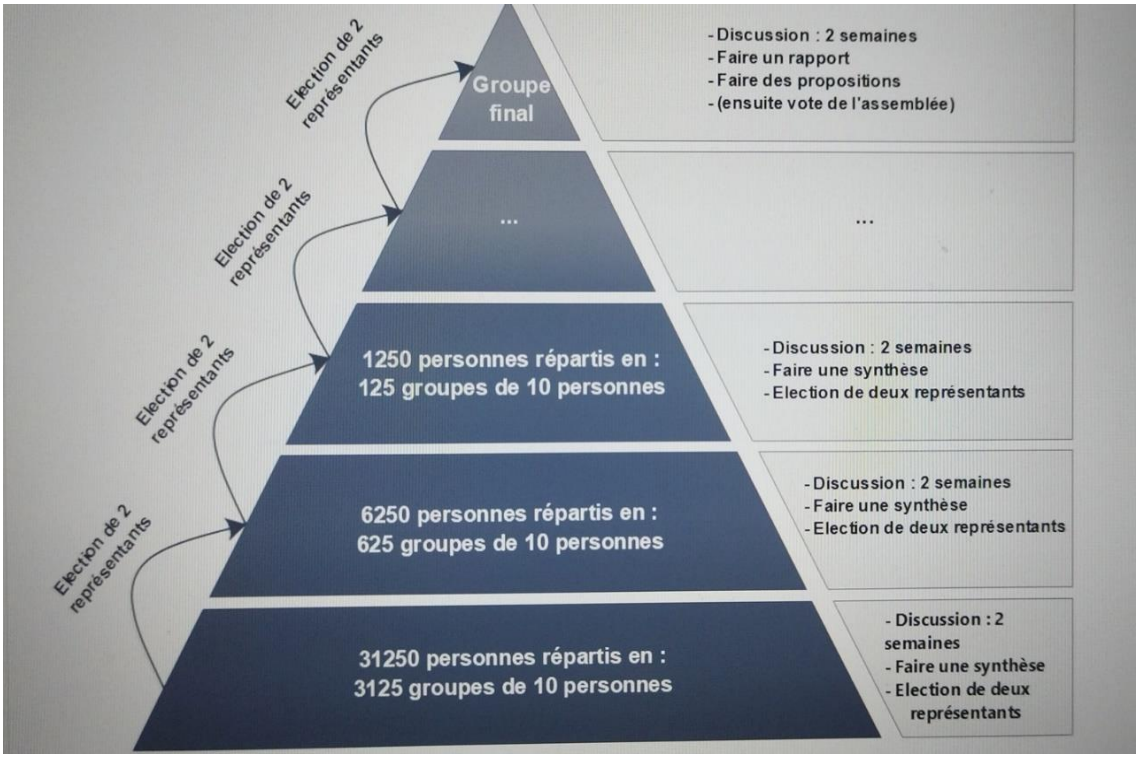


Illustration d'un schéma de débat

Au niveau du fonctionnement de notre application, nous avons fait en sorte de laisser l'immense majorité du code source de **Zulip** intacte. Afin de faciliter le processus de développement, nous avons plutôt préféré ajouter des fonctionnalités au-dessus du fonctionnement habituel de **Zulip**. C'est notamment le cas pour le gestionnaire de débat (Voir section Diapyr_bot).

II – Documentation techniques

1. Description brève de Zulip

Zulip est une plateforme de discussion collaborative basé sur Django. Destinée aussi bien au milieu public que privé, l'application est pourvue de nombreuses fonctionnalités, supporte une bonne intégration d'outils externes comme la suite Google et s'intègre facilement dans la plupart des navigateurs internet.

Un serveur **Zulip** contient une ou plusieurs organisations (appelées « realms »). C'est depuis une organisation que les utilisateurs vont pouvoir se rassembler pour discuter et ainsi débattre. Une organisation est structurée en plusieurs chaînes de discussion (channel).

Ci-dessous un résumé de l'architecture employée dans **Zulip**

Web App(FrontEnd) :

- Rendu backend : Jinja2 (HTML avec intégration de données issues de Django)
- Rendu Front-end : Handlebars (Moteur de template JavaScript)
- Autres Front-end : CSS/JS/TypeScript

Backend :

- **Base** --- > Python3, Framework : Django
- **WebSocket** (Pour voir les messages et plus en temps réel, c'est-à-dire, sans recharger les pages) : Tornado. Le code est localisé dans zerver/tornado
- **BDD** : PostgreeSQL
- **Serveur Web** : NGINX (Comme Apache) → Utilise les requêtes HTTP conformément aux règles décrites dans puppet/zulip/files/nginx/ et puppet/zulip/templates/nginx/.
- **Superviseur** : Supervisord →Gère le lancement de tous le projet, des process en arrière-plan et ceux qui tournent au premier-plan.
- **Mémoire cache** : memcached → Met en cache les modèles de base de données dans zerver/lib/cache.py et zerver/lib/cache_helpers.py

- **Message Broker :** RabbitMQ -> Gère de multiples tâches de fond (Envoyer des e-mails, notifications de messages, etc.) via un système de FIFO. L'interfaçage se fait en Python via la bibliothèque **pika.**

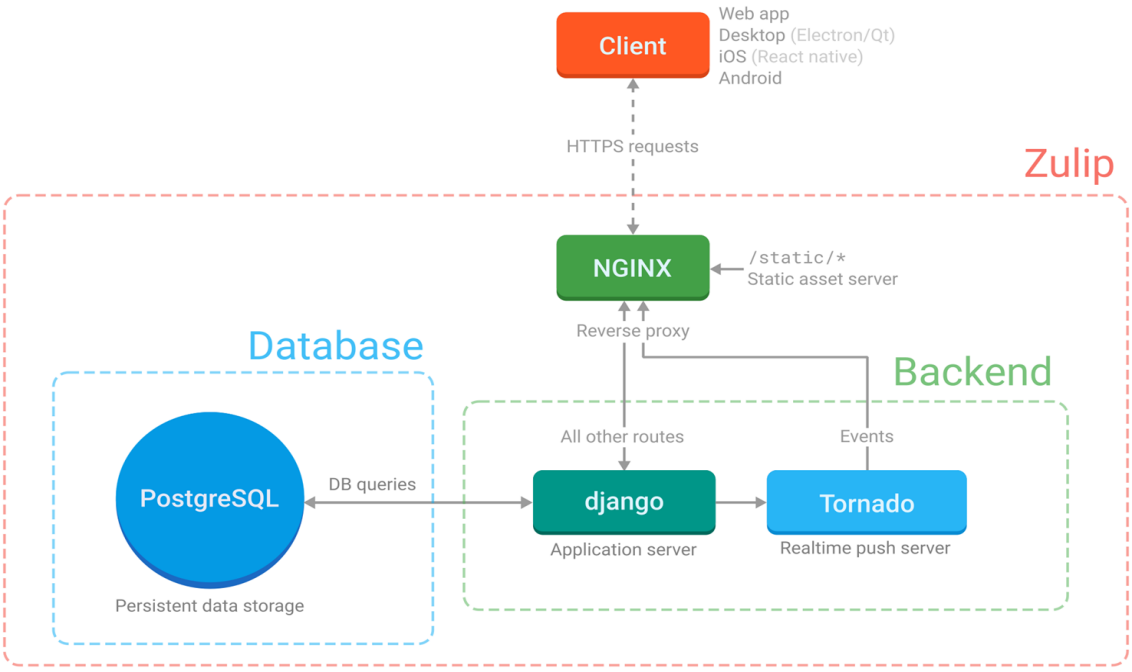
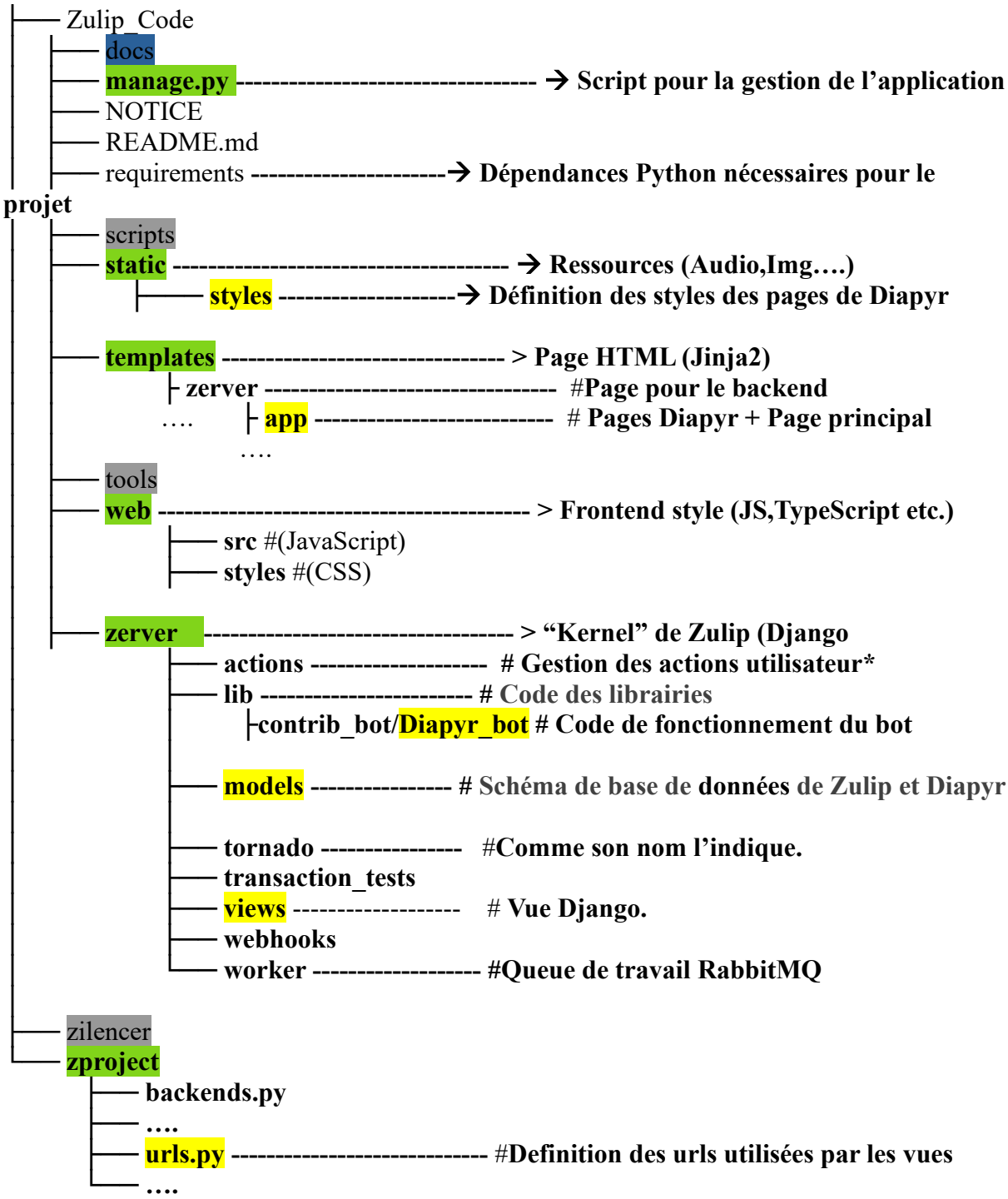


Schéma de l'organisation de Zulip

2. Structure du projet

La compréhension de la structuration du projet est essentielle afin de pouvoir modifier ou ajouter des fonctionnalités dans **Zulip**. Cette application étant très riche, il nous a fallu un certain avant d'apprivoiser l'arborescence du projet. Voici les dossiers essentiels de l'application



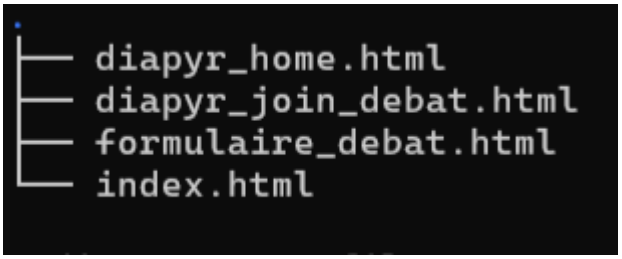
*Par action utilisateur, on entend le fait d'envoyer des messages ou des réactions, etc.

En jaune, sont représentés tous les dossiers ou ont a ajouté/modifié un fichier afin de rajouter les fonctionnalités de Diapyr.

Cette architecture de projet prend source dans la structure même d'un projet Django, le principal Framework utilisé pour le cœur de ce projet. Il est ainsi recommandé d'avoir une bonne connaissance de Python et de son Framework Django afin de pouvoir modifier plus facilement l'application. Ci-dessous, une liste de lien utile pour se familiariser à l'environnement

- Cours sur Django (OpenClassrooms) : https://openclassrooms.com/fr/courses/7172076-debutez-avec-le-framework-django
- Site officiel : https://www.djangoproject.com/
- Documentation Zulip : https://zulip.readthedocs.io/en/latest/index.html
- Base de Python : https://openclassrooms.com/fr/courses/7168871-apprenez-les-bases-du-langage-python , https://openclassrooms.com/fr/courses/7150616-apprenez-la-programmation-orientee-objet-avec-python

3. Pages HTML et styles



Les nouvelles pages du projet **Diapyr**, sont localisées dans le dossier **templates/zerver/app** à l'exception de index.html qui est la page principale de **Zulip**.

Ces pages sont chargées d'afficher le rendu final de Diapyr, à savoir l'interface principale pour voir les débats en cours(**diapyr_home.html**), le formulaire pour créer un débat (**formulaire_debat.html**) et le formulaire pour accéder à un débat (**diapyr_join_debat.html**).

Page d'accueil

Ces pages utilisent le moteur de template **Jinja2 (Associé au framework Flask)**. Ce moteur permet d'associer du HTML classique à une syntaxe singulière afin d'incorporer des éléments dynamiques enregistrés dans le SGBD (**PostgreeSQL**).

```
25  <h1>Bienvenue dans Diapyr !</h1>
26
27  <main>
28      <div class="Main-link">
29          <h2> Liste des débats</h2><br>
30          <ul class="debate-list">
31              {% if (debat|length) == 0 %}
32                  <p>Aucun débat n'a encore été créé.</p>
33              {% endif %}
34              {% for deb in debat%}
35                  <li class="debate-item">
36                      <strong>{{ deb.title}} - </strong>
37                      Description: {{ deb.description }}
38                      Date de debut: {{ deb.end_date }}
39                  </li>
40              {% endfor %}
41          </ul>
42      </div>
43  </main>
```

Exemple 1 : Extrait de code pour la génération de la liste des débats.

Le style CSS est directement lu depuis le dossier **static/styles.** Il est nécessaire d'incorporer la syntaxe suivante pour lier le CSS à l'HTML.

```
href="{{ static('styles/[Nom].css') }
```

4. Modèle de l'application

Le dossier modèles définit les tables qui seront générées dans **PostgreeSQL**. C'est dans ces fichiers qu'on va pouvoir ajouter des attributs ou des relations dans notre base de données. Dans notre cas, nous avons préféré générer un nouveau modèle (Nommé **debat.py**) plutôt que de modifier un modèle existant.

Dans le fichier **Debat.py,** on définit deux tables en relations ManyToMany c'est-à-dire qu'une table peut être en relation avec plusieurs autre représentant de l'autre table et vice-versa. Tout d'abord, on définit la table **Participant** qui permet de constituer un profil et une identité spécifique à un débat précis. Enfin, la table **Debat** permet d'enregistrer toutes les informations essentielles à un débat et de conserver une trace de la liste des participants inscrits aux débats.

Ces modèles sont primordiaux notamment dans la définition des vues et dans l'exécution du **Diapyr_bot**.

[Image en relation]

5. Définition des vues et url

Avant de pouvoir accéder à n'importe quelle page de l'application cette dernière doit être répertoriée dans le fichier **url.py** dans **zproject/url.py.** Le fichier **url.py** associe des adresses URL à des vues à exécuter pour traiter la requête http d'un utilisateur. Dans le cadre, d'un projet **Zulip,** il est nécessaire d'inclure ces URL dans la liste **i18n_urls** afin qu'elle soit visible pour l'utilisateur finale.

```
i18n_urls = [
    path("", home, name="home"),
    path("desktop_home/", desktop_home), #--> C'est la page d'accueil de Zulip
    …
    path("diapyr_debat/",formulaire_debat,name="diapyr_debat_form"),
    path("diapyr_home/",diapyr_home,name="diapyr_home"),
    path("diapyr_join_debat/",diapyr_join_debat,name="diapyr_join_debat"),
]
```

Les vues permettent de gérer la logique d'affichage dès que le serveur reçoit une requête http. Nous avons défini trois quoi pour gérer nos 3 pages HTML.

. Diapyr_Home

La vue **diapyr_home** permet d'afficher la page d'accueil de Diapyr. Elle va envoyer à la page HTML sous forme de réponse http(via la fonction **render**) les données issues de la BDD concernant les débats, afin de les afficher dans la page correspondante.

```
def diapyr_home(request: HttpRequest) -> HttpResponse:
    debat = Debat.objects.all()
    return render(request, 'zerver/app/diapyr_home.html', {'debat': debat})
```

. Formulaire_debat

La vue **formulaire_debat** va dans un premier temps enregistrer les informations saisies par l'utilisateur s'il a effectué une requête **POST**. Ensuite, on instancie un objet **Debat** qu'on initialise avec les valeurs du formulaire. Ces étapes d'enregistrement seront cruciales, pour le bot. Une fois l'opération effectuée, on redirige l'utilisateur dans la page d''accueil. Si l'utilisateur avait fait une requête **GET,** on le renvoie vers la page du formulaire.

. Join_debat

La vue diapyr_join_debat a un fonctionnement très similaire à la vue précédente. On va récupérer les informations du formulaire, générer un nouvel objet **Participant** et l'ajouter dans la BDD.

6. Diapyr_bot

Le bot Diapyr est un des composants essentiels au fonctionnement de Diapyr. Ce bot réutilise les fonctionnalités d'un bot Zulip auquel on ajoute plusieurs fonctionnalités spécifiques à **Diapyr**

Ce bot organise des débats en groupes successifs sur Zulip :

- Les utilisateurs s'inscrivent.
- Ils sont répartis en groupes (stream).
- À intervalles réguliers, une sélection est faite pour passer à l'étape suivante.
- Le débat continue jusqu'à ce qu'un seul groupe reste.

Composants principaux dans Diapyr_bot.py :

. **Méthodes de classe Debat(ObjectD) :** Gère un débat donné (paramètres, abonnés, étapes, groupes...).

__init__ :

Entrées :

- **name**: nom du débat
- **creator_email**: email du créateur
- **max_per_group**: max de personnes par groupe
- **end_date**: date limite d'inscription
- **time_between_steps**: délai entre les étapes
- **num_pass**: nombre de personnes à retenir entre les étapes

Sortie : None

Fonctionnement : Initialise un débat, en construisant un objet **ObjectD.**

add_subscriber :

Entrées : ObjectD self, **String** user_email, **String**

Sortie : None

Fonctionnement : Ajoute un participant au débat, dans la liste **subscriber d'ObjetD**.

split_into_groups :

Entrée : ObjectD self

Sortie : String[][] groups

Fonctionnement : Divise les participants (hors créateur) en groupes aléatoires selon l'attribut **max_per_group**. Le créateur est ajouté fictivement à chaque groupe. Renvoie la liste des utilisateurs triés.

create_streams_for_groups :

Entrée : ObjectD self, **: String[][]** groups

Sortie : String []

Fonctionnement : Ajoute les participants dans les streams correspondants aux groupes via la fonction **add_users_to_stream.** Envoie une notification à chacun utilisateur en message privé via la fonction **notify_users**.

next_step :

Entrée : ObjectD self

Sortie : Bool

Fonctionnement : Sélectionne aléatoirement des participants pour l'étape suivante, si le nombre de personnes restantes dans le débat est supérieur à **max_per_group.** Met à jour les abonnés et incrémente **step**. Renvoie **False** s'il n'y a pas assez de personnes et **True** le cas échant.

get_status :

Entrées : ObjectD self

Sorties : String

Fonctionnement : Renvoie une chaîne décrivant l'état actuel du débat (étape, groupes, timing...).

start_debate_process :

Entrée : ObjectD self

Sortie : None

Fonctionnement : Lance un thread qui :

- Attends time_between_steps,
- Appelle next_step(),
- Recrée des groupes et streams,
- Répète jusqu'à ce qu'un seul groupe reste.

. Fonctions utilitaires

Get_client() :

Entrée : None

Sortie : None

Fonctionnement : Récupère les informations des clients de Zulip. La configuration du bot est récupérée dans **zuliprc.txt**

add_users_to_stream :

Entrées : String stream_name, **String[]** user_emails

Sortie : Bool

Fonctionnement : Ajoute une liste d'utilisateurs à un stream donné. Renvoie **True** si l'opération a réussi et **False** le cas échant.

notify_users :

Entrées : String stream_name, **String[]** user_emails

Sortie : None

Fonctionnement : Envoie un message privé à chaque utilisateur pour les prévenir de leur affectation à un groupe.

get_user_id :

Entrée : String user_email

Sortie : String

Fonctionnement : Recherche l'user_id Zulip correspondant à un email.

Get_email_by_full_name:

Entrée : String full_name

Sortie : String

Fonctionnement : Retourne l'email de l'utilisateur en fonction du pseudo qu'il a choisi lors de la création de son compte*

* Cette étape est nécessaire, car dans Zulip, l'adresse-mail de l'utilisateur agit comme un identifiant. De plus, l'adresse n'est pas la même que celle renseignée lors de l'inscription dans Zulip.

. Gestion des messages (commandes utilisateur)

handle_message :

Entrée : dict[String : String] msg

Sortie : None

Fonctionnement : Réagis à des commandes envoyées depuis des messages directs dans Zulip à destination du bot :

- @créer : création rapide d'un débat.
- @configurer : création via JSON (plus flexible).
- @s'inscrire : inscription à un débat existant.
- @état : affiche l'état actuel du débat.

. Récupération des informations de la base de données

Create_debat :

Entrée : None

Sortie : None

Fonctionnement : Parcours les objets **Debat** de la base de données et générer un débat en instanciant un nouvel objet **ObjectD** s'il n'a pas été généré auparavant. On ajoute dans le même temps l'utilisateur dans **listeDebat**

Add_user :

Entrée : None

Sortie : None

Fonctionnement : Ajout d'un utilisateur en parcourant la liste des personnes inscrit à un débat depuis la table **Debat.** L'ajout est effectif seulement s'il n'a pas encore été inscrit à un débat. On ajoute l'utilisateurs dans **listeDebat** en récupérant son email via **get_email_by_full_name()** et par la méthode **add_subscriber().**

. Boucle principale et threads

check_and_create_channels :

Entrée : None

Sortie : None

Fonctionnement : Vérifie régulièrement si la date d'inscription est dépassée. Si oui :

- Forme les groupes via **split_into_groups**
- Crée les streams via **create_streams_for_groups**,
- Démarre le processus de débat via **start_debate_process()**.

message_listener :

Entrée : None

Sortie : None

Fonctionnement : Démarre l'écoute des messages entrants (commandes utilisateurs).

main_loop :

Entrée : None

Sortie : None

Fonctionnement : Boucle principale infinie qui exécute les actions suivantes toutes les 10 secondes.

- Génére des débats, si de nouveau ont été formé via le formulaire (**create_debat()**)
- Ajout d'utilisateurs dans débats, si non affeté via **add_user()**
- Appelle **check_and_create_channels()**

Fonctionne en parallèle de **message_listener()**

Exécution principale

```
if __name__ == "__main__":
    threading.Thread(target=message_listener).start()
    main_loop()
```

Le listener des messages fonctionne dans un thread séparé.

La logique du débat continue en parallèle dans la boucle principale.