ChatGPT

Plan pour insérer et vérifier des données dans Konnaxion v14

1. Insertion manuelle de données dans la base de données Neon

Préparation de l'accès à la base de données : Assurez-vous que la connexion PostgreSQL vers Neon est correctement configurée dans l'environnement. La variable `DATABASE_URL` dans le fichier **.env** (ou fichier d'environnement Docker) doit pointer vers l'URL Neon. Par exemple, le fichier `.envs/.local/.postgres` utilisé en développement local contient uniquement la variable `DATABASE_URL` de Neon [1]. Le paramétrage Django utilise cette variable pour définir la base de données par défaut [2]. Vérifiez que votre instance Neon est accessible (adresse IP autorisée ou tunnel configuré si nécessaire).

Choix de la méthode d'insertion : Vous pouvez insérer des données manuellement de plusieurs façons. Voici deux méthodes recommandées :

- *Via le shell SQL (psql)* : Utilisez la commande Django `dbshell` pour ouvrir un terminal psql connecté à la base de données Neon avec les bons identifiants [3]. Par exemple, lancez la commande :

  ```
  python manage.py dbshell
  ```

  Cela ouvrira `psql` avec la connexion définie dans les settings (moteur PostgreSQL). Vous pouvez alors exécuter des requêtes SQL `INSERT` classiques. Par exemple, pour ajouter une nouvelle **catégorie de débat** (modèle *EthikosCategory*), vous pourriez exécuter dans `psql` :

  ```
  INSERT INTO ethikos_ethikoscategory (name, description)
  VALUES ('Nouveau thème', 'Description de la catégorie');
  ```

  (Remarque : les noms de table Django sont généralement formés de `<nom_app>_<nom_modèle>` *en minuscules.)*
  Après exécution, utilisez une requête `SELECT` ou `\x` pour vérifier que la ligne a bien été insérée, ou quittez `psql` (`\q`) pour continuer.

- *Via le shell Python Django (ORM)* : Alternativement, vous pouvez utiliser l'ORM Django pour créer des objets, ce qui gère automatiquement les champs. Lancez le shell Django avec `python manage.py shell` puis, par exemple :

  ```
  from konnaxion.ethikos.models import EthikosCategory
  EthikosCategory.objects.create(name="Nouveau thème",
  description="Description de la catégorie")
  ```

Ceci insérera une nouvelle catégorie dans la base. Vous pouvez de même créer d'autres objets (débats, projets, etc.) via l'ORM, ce qui évite d'écrire du SQL manuel.

- *Via l'interface d'administration Django* : Si le modèle est enregistré dans l'admin, vous pouvez également passer par l'interface **Django Admin** (accessible à l'URL `/admin/`) pour créer des objets manuellement. Connectez-vous avec un compte superutilisateur (à créer via `manage.py createsuperuser` si ce n'est pas déjà fait) et utilisez les formulaires d'administration du modèle concerné. Cette méthode assure aussi le respect des validations du modèle.

Respect des contraintes des modèles : Avant l'insertion, identifiez les champs obligatoires et les relations du modèle cible. Par exemple, le modèle **EthikosTopic** (débat) requiert au minimum un titre et un statut (`open/closed/archived`) [4], tandis qu'un **Project** (projet collaboratif) nécessite un titre *et* un créateur lié à un utilisateur existant [5]. Assurez-vous donc de fournir des valeurs valides : - Pour les champs **ForeignKey** (par ex. *creator* d'un Project ou *author* d'une suggestion), utilisez l'identifiant d'une instance existante (ex. l'ID d'un utilisateur valide). Créez d'abord les objets référencés si nécessaire (ex. créer un utilisateur ou utiliser le superuser existant). - Pour les champs **ENUM/choices** (par ex. statut d'un débat), utilisez une valeur reconnue par l'application (ex. `"open"` pour un débat actif).
- Pour les champs avec des valeurs par défaut ou auto-générées (timestamps, PK auto-incrémenté), vous pouvez généralement les omettre dans l'INSERT manuel pour laisser la base ou l'ORM les remplir.

Après insertion, **vérifiez dans la base** que les données sont présentes. Vous pouvez utiliser une requête SELECT dans `dbshell` ou actualiser la vue correspondante dans l'admin Django pour voir le nouvel enregistrement.

2. Vérification de l'affichage des données dans le front-end

Une fois les données insérées en base, l'étape suivante est de s'assurer qu'elles apparaissent dans l'interface utilisateur de Konnaxion (module front-end). Pour cela :

- **Lancer le backend et le frontend en mode développement :** Démarrez le serveur Django local (via `python manage.py runserver` sur le port 8000, ou en lançant `docker-compose up` si vous utilisez Docker). Assurez-vous que le backend est bien connecté à la base Neon (vérifiez la console pour s'assurer qu'aucune erreur de connexion n'apparaît). Ensuite, lancez le serveur frontend (application Next.js) en développement, par exemple avec `npm run dev` ou `yarn dev` dans le dossier `frontend`. Le frontend Konnaxion v14 est une application Next.js 13+ qui devrait démarrer (par défaut sur http://localhost:3000). Cette application est configurée pour appeler le backend sur les endpoints commençant par `/api` (le `baseURL` de l'API est paramétré soit via `NEXT_PUBLIC_API_BASE`, soit par défaut sur `/api` ce qui, en développement, est généralement proxifié ou redirigé vers le port du backend [6]).

- **Accéder à l'interface et naviguer vers la section appropriée :** Connectez-vous sur l'application frontend avec un utilisateur possédant les droits nécessaires (un simple compte utilisateur suffit généralement pour voir les données publiques, mais pour certaines données restreintes un compte admin ou spécifique peut être requis). Utilisez le menu principal pour trouver la page liée aux données que vous avez insérées. L'application Konnaxion est organisée par modules thématiques, comme indiqué par la cartographie des routes : par exemple, un débat Ethikos (*EthikosTopic*) inséré devrait apparaître sur la page **Debate Hub** (`/debate`) dans l'onglet **Open** si son statut est "open" [7]. De même, une nouvelle consultation (*Consultation*) serait visible sur **Consultation Hub** (`/consult`) sous l'onglet *Live*, ou un projet (*Project*) apparaîtrait dans **Project Studio** (`/projects`) sous l'onglet *Browse* ou *My Projects*.

- **Exemple :** si vous avez inséré un nouveau débat via la base de données, allez sur la page **Debate Hub** du module *ethiKos* (URL `/debate`). Par défaut, cette page affiche la liste des débats **ouverts**. Si votre objet EthikosTopic a un statut `open`, vous devriez le voir listé ici avec le titre que vous avez fourni. S'il a un statut `closed`, basculez vers l'onglet *Archived* (débats archivés) dans l'interface pour le retrouver [7]. Vérifiez que toutes les informations pertinentes (titre, etc.) s'affichent correctement.

- **Autre exemple :** pour une **catégorie de débat** (*EthikosCategory*) insérée manuellement, vous pourriez la voir apparaître dans les formulaires de création de débat (p.ex. menu déroulant des catégories si implémenté) ou dans l'admin. Si elle n'est pas immédiatement visible côté utilisateur, vérifiez via l'admin Django que la catégorie existe, puis tentez de créer un nouveau débat depuis l'UI en choisissant cette catégorie pour confirmer qu'elle est bien disponible.

- **Vérifications techniques si les données n'apparaissent pas :** Si vous ne voyez pas les données dans le front-end, procédez à quelques vérifications:

- **Requête API directe :** Utilisez le **navigateur API DRF** ou `curl`/Postman pour interroger l'endpoint correspondant du backend et voir si les données sont renvoyées. Par exemple, l'endpoint GET `/api/ethikos/topics/` devrait retourner la liste des débats [8]. Si votre entrée insérée figure dans la réponse JSON, cela signifie que le backend l'envoie bien. Dans le cas contraire, inspectez les configurations backend – assurez-vous que le **ViewSet/serializer** de l'API inclut bien cet objet. Par exemple, un ViewSet peut filtrer par utilisateur courant ou statut : vérifiez la présence de filtres ou permissions dans le code DRF (fichiers `api_views.py` et `serializers.py` du module concerné). Assurez-vous aussi que l'objet respecte ces filtres (ex: un projet **privé** pourrait ne pas être visible par un utilisateur lambda).
- **Console du navigateur et logs** : Sur le front-end, ouvrez la console de développement. Rechargez la page et voyez si une requête réseau part vers l'API. Vérifiez qu'elle aboutit (code 200) et ne renvoie pas d'erreur ou une liste vide. En cas d'erreur (ex. 403 Forbidden ou 500 Server Error), cela indique un problème de permission ou de sérialisation. Consultez alors les logs du serveur Django pour plus de détails sur l'erreur.
- **Django Admin** : Comme solution de contournement rapide, vérifiez via l'admin Django si l'objet existe bien et si ses champs sont corrects. L'admin vous permettra de confirmer la présence de l'ID, et éventuellement de corriger un champ (ex. statut mal défini, absence d'un champ obligatoire) qui empêcherait l'affichage côté frontend.

En résumé, cette étape consiste à s'assurer que le **flux lecture** fonctionne : ce qui est en base est correctement exposé par le backend et consommé par le frontend. Si tout est configuré conformément au modèle, les données insérées manuellement doivent s'afficher dans l'UI aux emplacements prévus.

3. Utilisation du front-end pour créer ou modifier des données en base

Après avoir validé l'affichage, il est important de tester le **flux d'écriture** à travers l'application elle-même, c'est-à-dire utiliser les formulaires ou actions du front-end pour envoyer de nouvelles données ou modifier des données existantes dans la base de données. Ceci permet de vérifier l'intégration bout-en-bout (frontend → API → base de données) et de se familiariser avec l'architecture.

- **Création d'une nouvelle entrée via le front-end :** Identifiez dans l'interface utilisateur l'endroit où l'on peut créer le type de données que vous souhaitez tester. Par exemple, pour créer un nouveau débat, utilisez l'onglet **Start New Debate** sur la page Debate Hub (`/debate`) [7]. Pour

un nouveau projet, allez dans **Project Studio** (`/projects`) et ouvrez l'onglet **Create**. Pour soumettre une suggestion citoyenne dans une consultation, utilisez l'onglet **Suggest** sur la page Consultation Hub (`/consult`) [9] , etc. Remplissez le formulaire proposé avec des valeurs de test cohérentes (en respectant les contraintes du formulaire : longueurs max, formats d'email si requis, etc.) puis validez l'envoi.

- Au moment de la soumission, observez ce qu'il se passe en coulisses. Le front-end va appeler l'API REST correspondante (ex.: un POST sur `/api/ethikos/topics/` pour créer un débat). Ces endpoints sont définis dans le backend via des ViewSets DRF enregistrés par module (par ex., le **TopicViewSet** gère `/api/ethikos/topics` [8]). Si vous avez la console réseau ouverte, vous devriez voir la requête POST partir et la réponse du serveur (201 Created ou les données créées en JSON).

- Vérifiez que la nouvelle donnée apparaît bien dans l'interface immédiatement après la création. La plupart des formulaires frontend devraient mettre à jour la liste ou rediriger l'utilisateur vers l'élément créé. Par exemple, après avoir créé un débat, vous pourriez être redirigé vers la liste des débats ouverts où le nouveau débat est listé, ou vers la page détaillée du débat.

- **En base de données**, confirmez que l'objet a bien été persisté. Utilisez soit l'admin Django, soit `manage.py dbshell` avec une requête SELECT. Par exemple, si vous avez créé un débat *EthikosTopic*, exécutez dans dbshell :

```
SELECT title, status, start_date FROM ethikos_ethikostopic WHERE title
= 'TitreDeMonDebat';
```

  Vous devriez voir l'enregistrement correspondant. De même pour tout autre objet créé (projet, suggestion, etc.), vérifiez son existence et ses champs en base. Cela garantit que le **backend (ViewSet, Serializer)** a correctement traité la requête du front-end et enregistré les données.

- **Modification d'une entrée existante via le front-end :** Testez également la capacité à modifier ou mettre à jour des données depuis l'interface. Par exemple, éditez un objet que vous avez inséré manuellement pour voir si les changements se répercutent en base. Selon l'application :

- Il peut y avoir des formulaires d'édition dédiés (par exemple, éditer les détails d'un projet via l'onglet **Settings** ou **Edit** sur la page d'un projet, modifier son profil utilisateur sur `/profile` ou la réputation sur `/reputation` [10] , etc.). Utilisez ces interfaces pour changer un champ (ex. le titre ou la description) et soumettez la modification.
- Sur certaines listes, l'édition peut se faire inline ou via des actions (ex. changer le statut d'un élément via un bouton *toggle*, supprimer un élément via un bouton *Delete*, etc.). Effectuez une action de mise à jour ou suppression si disponible.

Après chaque modification, **validez les résultats** : - Côté UI, assurez-vous que le changement est visible immédiatement (par ex., le nouveau titre s'affiche, l'objet n'apparaît plus s'il a été supprimé, etc.). - Côté base de données, interrogez la table pour voir la mise à jour. Par exemple, si vous avez modifié le titre d'un projet, vérifiez ce titre dans la table correspondante (`keenkonnect_project` par exemple). - Là encore, vous pouvez utiliser l'admin Django pour confirmer la modification. L'admin est utile pour voir

l'historique du champ avant/après (si vous avez l'audit log activé) ou simplement pour constater la mise à jour.

- **Vérifications techniques en cas de problème :** Si la modification ou création via l'UI ne fonctionne pas (aucun effet, ou erreur):
- Vérifiez la **réponse de l'API** dans la console réseau. Une erreur 400 (Bad Request) peut indiquer un champ manquant ou invalidé par le serializer (par ex., un champ obligatoire non rempli ou une valeur hors contrainte). Dans ce cas, le front-end devrait afficher un message d'erreur provenant du serializer DRF. Corrigez les valeurs saisies selon les messages (ex.: ajouter un titre si oublié, respecter la longueur maximale).
- Une erreur 403 (Forbidden) peut signifier que vous n'avez pas les permissions nécessaires (par ex., essayer de modifier un objet sans en être propriétaire). Assurez-vous de tester avec un compte approprié ou ajustez temporairement les permissions pour le test.
- Une erreur 500 indique une exception côté serveur. Consultez la trace dans les logs Django pour identifier la cause (ex.: bug dans la ViewSet, champ non géré par le serializer, etc.). Vous pourrez alors parcourir le code du **serializer** ou de la **vue DRF** correspondante pour corriger le problème. Les fichiers clés se trouvent dans le module de l'app concernée (par ex. `konnaxion/ethikos/api_serializers.py` et `api_views.py` pour le module Ethikos). Assurez-vous que tous les champs nécessaires y sont définis.

En utilisant les interfaces front-end pour créer et modifier des données, vous validez le bon fonctionnement de toute la chaîne. Vous aurez également identifié les fichiers importants côté backend : - Les **modèles Django** (`models.py`) qui définissent la structure des données. - Les **serializers DRF** qui contrôlent comment les modèles sont exposés et validés à travers l'API. - Les **ViewSets/Views DRF** (`views.py` ou `api_views.py`) qui définissent la logique d'API (list, création, modification, permissions, filtres). - L'**admin Django** (`admin.py`) qui permet la gestion manuelle des données pour l'administration ou les tests rapides.

N'hésitez pas à vous référer à ces fichiers lors de vos manipulations : par exemple, pour comprendre pourquoi un champ n'apparaît pas dans la réponse, vérifiez le serializer; pour savoir si une action d'édition est permise, vérifiez la view (elle peut avoir `permission_classes` ou des méthodes `update()` spécifiques). L'**architecture modulaire** de Konnaxion fait que chaque module (ethiKos, keenKonnect, etc.) a ses propres modèles et API endpoints correspondants [8], mais le principe reste le même pour tous.

En suivant ce plan pas à pas, un développeur découvrant l'architecture pourra insérer manuellement des données dans la base Neon, les voir apparaître dans l'application Konnaxion v14, puis utiliser l'application elle-même pour créer/modifier des données et confirmer leur persistance. Ces étapes couvrent à la fois la manipulation directe de la base et l'utilisation des couches API et front-end, offrant ainsi une compréhension complète du flux de données dans le système.

---

[1] Code_core_20251115_154246.txt
file://file_000000005f70720c8b6cda776790222e

[2] [8] Code_config_20251115_154246.txt
file://file_00000000e7c0722f8d7d34c527304dc5

[3] django-admin and manage.py | Django documentation | Django
https://docs.djangoproject.com/en/5.2/ref/django-admin/

4 5 Konnaxion v14 – Database Schema Reference (Custom Tables).md
file://file_0000000097a471f58d0917a533e01db3

6 frontend_services_20251115_151545.txt
file://file_00000000eed471f5a62bf1661ccd870d

7 9 10 Konnaxion v14 – Site Navigation Map (Top-Level Routes).md
file://file_0000000006d871f58f8bd4b01ecfa633