Vous : bjr

Bjr 🙂 Ça va ?

Tu veux que je t'aide sur quoi aujourd'hui ?

admin.py
Python
apps.py
Python
json_transfer.py
Python
models.py
Python
tests.py
Python
urls.py
Python
views.py
Python
Voici le contenu du dossier app_acceuil
Réflexion durant quelques secondes

Parfait — j'ai bien reçu le contenu de ton app **`app_acceuil`** (admin, models, views, urls, json_transfer, etc.). Voilà un **résumé clair de ce que fait chaque fichier** + les points importants.

1) `models.py` — les données du portfolio

Tu as 7 modèles principaux :

models

- **`Projetscards`** : témoignages (nom, image, description) + **anti-doublon** via `fingerprint` unique (SHA256 nom+description normalisés).
- **`Projetsfirstspeciality`** : une "photo de spécialisation" (image + description).
- **`Projetmesrealisations`** : projets/réalisations (nom, image, description) + compteur `compteur_demo_live`.
- **`Projetphotodeprofil`** : photo de profil (image).

- **`ProjetAproposDeMoi`** : texte “à propos”.
- **`MesCompetencesCles`** : compétences (nom).
- **`ReseauSocial`** : réseaux sociaux (nom unique, url, visible).

🟢 Plusieurs modèles ont `est_visible` pour afficher/masquer sur le site.

2) **`admin.py`** — configuration Django Admin

Tu déclares des classes `ModelAdmin` pour afficher/éditer rapidement `est_visible`, et tu gères le côté “singleton” (une seule ligne possible) pour certains modèles :

admin

- `Projetsfirstspeciality`, `Projetphotodeprofil`, `ProjetAproposDeMoi` : **bloque l’ajout** si une ligne existe déjà (`has_add_permission`).
- `MesCompetencesCles` et `ReseauSocial` sont enregistrés via `@admin.register`.

3) **`views.py`** — page publique + dashboard admin “custom”

Page publique

`aff_acceuil_PAGE_PUBLIQUE_Vue_par_les_VISITEURS` :

- récupère uniquement ce qui est `est_visible=True`
- envoie au template `index.html` (témoignages, spécialités, projets, compétences, réseaux, photo profil, a propos)
- récupère aussi des menus visibles selon le rôle (via `RoleMenuAcces`)

  views

Dashboard (page_administration.html)

`page_Mon_Espace_Administration` :

- accès réservé `Administrateur` ou `Moderateur` (via session)
- gère *plein* d’actions POST : upload photo, changer visibilités, CRUD projets, CRUD témoignages, compétences, réseaux sociaux, reset compteur, etc.

views

- pour **témoignages**, tu fais un **upsert** via fingerprint (évite doublons) avec `update_or_create`.

Compteur “Démo Live”

- `incrementer_compteur_demo_live` incrémente `compteur_demo_live` puis redirige vers une page de démo.

  views

- `reinitialiser_compteur_projet` remet à zéro pour un projet donné.

4) `urls.py` — routes

- routes **export/import JSON** (admin)
- route publique `/` et `/index.html`
- dashboard `/dashboard/`
- démo live `realisations/<id>/demo-live/`
- quelques pages annexes + redirections d’ancres `/#projets`, etc.

  urls

5) `json_transfer.py` — export/import JSON + fichiers en base64

Très complet :

json_transfer

Export

- `export_portfolio_json` : export des modèles de `app_acceuil`
- `export_users_json` : export rôles/users/menus/etc (depuis `app_contact` + `app_user`)
- sérialise les images/fichiers en **base64** + metadata (mimetype, filename…)

Import

- `import_json_bundle` : reçoit 2 fichiers (portfolio et users) + options “replace”
- si replace = purge, sinon **upsert intelligent**
- gère les FK en exportant des ids, puis en les restaurant proprement

6) `apps.py` et `tests.py`

- config standard `AppAcceuilConfig`

  apps

- tests vides

  tests

Point important (petit bug potentiel)

Dans `json_transfer.py`, tu as :

```
"app_acceuil.Projetmesrealisations": ["nom"],
```

mais dans `models.py` un commentaire dit que ton import utilisait `"nom_projet"` avant. Là c’est **OK** parce que ton champ est bien `nom`, mais si tu as des anciens JSON qui contiennent `nom_projet`, l’upsert ne matchera pas (et tu risques des doublons à l’import).

Si tu me dis maintenant ce que tu veux faire avec ce dossier (ex: corriger un bug, ajouter une API REST, sécuriser l’import/export, refactor dashboard, etc.), je te propose directement la solution concrète (code + modifications exactes).

Sources
0003_alter_projetscards_images.py
Python
0004_projetscards_created_at.py
Python
0005_projetsfirstspeciality.py
Python

0006_projetmesrealisations.py
Python
0007_projetphotodeprofil.py
Python
0008_projetaproposdemoi.py
Python
0009_mescompetencescles.py
Python
0010_mescompetencescles_est_visible_and_more.py
Python
Voici le contenu du dossier app_acceuil/migrations
Réflexion durant une seconde

Parfait — j'ai bien reçu le contenu du dossier **`app_acceuil/migrations`**. Voici ce que tes migrations font, dans l'ordre, et ce qu'elles apportent à ton schéma.

Chronologie des migrations

`0003_alter_projetscards_images.py`

- Modifie le champ `images` de `Projetscards` pour qu'il soit **nullable** et **optionnel** (`blank=True, null=True`) et stocké dans `static/images/`.

  0003_alter_projetscards_images

`0004_projetscards_created_at.py`

- Ajoute un champ `created_at` à `Projetscards` : `DateTimeField(auto_now_add=True, null=True)`.

  0004_projetscards_created_at

  `null=True` permet de ne pas casser les anciennes lignes déjà présentes.

`0005_projetsfirstspeciality.py`

- Crée le modèle `Projetsfirstspeciality` avec :
  - `images` (optionnel)
  - `description_speciality` (texte optionnel)
  - `created_at` (auto)

0005_projetsfirstspeciality

0006_projetmesrealisations.py

- Crée le modèle `Projetmesrealisations` avec :
  - `nom` (CharField 100, optionnel)
  - `images` (optionnel)
  - `description` (optionnel)
  - `created_at` (auto)

0006_projetmesrealisations

0007_projetphotodeprofil.py

- Crée le modèle `Projetphotodeprofil` avec :
  - `images` (optionnel)
  - `created_at` (auto)

0007_projetphotodeprofil

0008_projetaproposdemoi.py

- Crée le modèle `ProjetAproposDeMoi` avec :
  - `description` (optionnel)
  - `created_at` (auto)

0008_projetaproposdemoi

0009_mescompetencescles.py

- Crée le modèle `MesCompetencesCles` avec :
  - `nom` unique (optionnel, max 100) avec `help_text`
  - `created_at` (auto)
  - options `ordering = ['-created_at']` + verbose names

0009_mescompetencescles

0010_mescompetencescles_est_visible_and_more.py

- Ajoute le champ `est_visible` (booléen, default True) à **plusieurs** modèles :
  - `MesCompetencesCles`, `Projetscards`, `Projetsfirstspeciality`, `Projetmesrealisations`, `Projetphotodeprofil`, `ProjetAproposDeMoi`

    0010_mescompetencescles_est_vis…

- Ajuste aussi le `help_text` de `MesCompetencesCles.nom` (ex “Python” au lieu de “Python, Gestion de projet”).

  0010_mescompetencescles_est_vis…

Ce que ça signifie côté application

- Tu as progressivement construit ton “portfolio” : témoignages → spécialités → réalisations → photo profil → à propos → compétences.
- Puis tu as ajouté `est_visible` partout pour **piloter l’affichage sur le site** (cohérent avec ta vue publique qui filtre sur `est_visible=True`).

Si tu veux, je peux aussi te dire **si l’ordre des dépendances est cohérent** avec ton `models.py` actuel (et repérer d’éventuels écarts “modèle ≠ migrations”, par ex champ manquant type `fingerprint`, `compteur_demo_live`, `url`, etc.).

Sources
admin.py
Python
apps.py
Python
models.py
Python
tests.py
Python
urls.py
Python
views.py
Python
Voici le contenu du dossier app_contact
Réflexion durant quelques secondes

Parfait — j’ai bien reçu le contenu de **`app_contact`**. Voici le résumé fichier par fichier, avec ce que fait l’app concrètement.

1) `models.py` — Contact + Auth maison (Role/User)

Tu as 3 grandes parties :

models

`Contact`

- Champs : `nom`, `email`, `message`, `date_envoi` (auto).
- Le `save()` force **email en minuscule** (`strip().lower()`) pour éviter les doublons “Test@…” vs “test@…”.

  models

`Role`

- `nom_role` unique (ex : “Administrateur”, “Moderateur”, “Utilisateur”).

  models

`User`

- `nom`, `email` unique, `password` (hash), et `role` en `ForeignKey` vers `Role` (avec `on_delete=models.CASCADE`).

  models

- `save()` normalise aussi l’email en minuscule.

  models

⚠ Point important : `on_delete=models.CASCADE` signifie **si tu supprimes un rôle, tu supprimes tous les users** liés. (C’est parfois voulu, mais souvent risqué.)

---

2) `views.py` — inscription + login + logout (sessions)

Tu fais une auth “maison” (pas Django auth), avec hash des mots de passe et session.

views

Pages

- `page_inscription` : rend `page_inscription.html`.

  views

- `aff_contact` : affiche tous les `Contact` dans `index.html` via la clé `formation`.

  views

Inscription (avec rôle)

`index_inscription_view` :

- récupère `nom/email/password/role`
- valide : nom obligatoire, email format (regex), email unique, mdp min 6
- récupère le rôle via `Role.objects.get_or_create(nom_role=...)`
- crée le `User` avec `make_password(password)` puis stocke en session : `{id, nom, role}`

  views

- redirige vers `/#accueil` en cas de succès.

  views

Connexion

`login_view` :

- cherche `User` par email (en minuscule)
- vérifie le hash via `check_password`
- stocke en session : `{id, nom, email, role}` puis redirige `/#accueil`.

views

Déconnexion

`deconexion_view` :

- `request.session.flush()` (vide toute la session)
- redirige vers `name_acceuil`.

views

---

3) `urls.py` — routes

Tu exposes 4 routes :

urls

- `root_gemini_accueil` → `index_inscription_view`
- `connexion/` → `login_view`
- `deconnexion/` → `deconexion_view`
- `page_inscription` → `page_inscription`

---

4) `admin.py`, `apps.py`, `tests.py`

- `admin.py` vide (rien enregistré).

admin

- `apps.py` standard `AppContactConfig`.

apps

- `tests.py` vide.

  tests

Petits points à surveiller

- **Sécurité / cohérence** : comme tu n'utilises pas `django.contrib.auth`, tout repose sur ta session `request.session['user']` (ce que tu fais déjà dans `app_acceuil` côté dashboard).

  views

- **Redirect logout** : `return redirect('name_acceuil')` suppose qu'il existe une URL nommée `name_acceuil` (probablement dans `app_acceuil/urls.py`). Si le nom n'existe pas, ça plantera.

  views

- **Role CASCADE** : si tu veux éviter qu'une suppression de rôle supprime des comptes, passe plutôt à `PROTECT` (ou `SET_NULL` si tu acceptes un rôle nul).

Si tu m'envoies aussi le dossier **`app_contact/migrations`**, je peux te dire si la base est parfaitement alignée avec tes modèles (et repérer les champs qui auraient été ajoutés dans le code sans migration).

Sources
0001_initial.py
Python
0002_user.py
Python
0003_alter_user_email.py
Python
0004_user_role.py
Python
0005_role_alter_user_nom_alter_user_password_and_more.py
Python
0006_table_role_de_user.py
Python
0007_administrateur_role.py

Python
Voici le contenu du dossier app_contact/migrations
Réflexion durant une seconde

Super — tes migrations **`app_contact/migrations`** racontent bien l'évolution de ton module contact + auth "maison". Voilà la lecture claire, dans l'ordre.

Chronologie des migrations

`0001_initial.py`

Crée le modèle **`Contact`** avec `nom`, `email`, `message`, `date_envoi` (auto_now_add).

0001_initial

`0002_user.py`

Ajoute un modèle **`User`** (sans rôle à ce moment-là) :

- `nom` (max 250)
- `email` (pas unique encore)
- `password` (TextField)

0002_user

`0003_alter_user_email.py`

Rend `User.email` **unique** (`unique=True`).

0003_alter_user_email

`0004_user_role.py`

Ajoute un champ `role` **en CharField à choix** (`Administrateur/Moderateur/Utilisateur`, default `Utilisateur`).

0004_user_role

`0005_role_alter_user_nom_alter_user_password_and_more.py`

Gros changement : passage d'un "role texte" à une vraie table de rôles.

- Crée le modèle **`Role`** (`nom_role` unique).

  0005_role_alter_user_nom_alter_…

- Modifie `User.nom` → max 100
- Modifie `User.password` → CharField max 255
- Remplace `User.role` → **ForeignKey vers Role** (`CASCADE`, related_name=`utilisateurs`)

  0005_role_alter_user_nom_alter_…

✅ À partir d'ici, ton `models.py` actuel (Role + FK) est cohérent avec la base.

`0006_table_role_de_user.py`

Migration **RunPython** qui insère des rôles par défaut :

- Administrateur, Moderateur, Utilisateur, Responsable de Saisie

  0006_table_role_de_user

⚠️ Petit détail : le `get_or_create(nom_role=code, defaults={'nom_role': nom})` est un peu "bizarre" :

- comme `nom_role` sert à la fois de "code" et de "nom", le `defaults` ne sert presque jamais (si l'objet n'existe pas, il sera créé avec `nom_role=code`).
- résultat probable : tu auras `nom_role="Administrateur"` etc. (ce qui est OK), mais la paire `(nom, code)` est inutile telle quelle.

`0007_administrateur_role.py`

Migration **RunPython** qui crée :

- le rôle `Administrateur` (si absent)
- un user admin par défaut `admin@exemple.com` avec mot de passe hashé `admin123`

  0007_administrateur_role

⚠️ Point important : ce compte est **en dur** dans l'historique de migration → en prod, c'est un risque si ce mdp n'est pas changé.

Ce que je retiens (cohérence & risques)

- ✅ Le schéma final correspond bien à tes modèles actuels : `User.role` = FK vers `Role`.

  0005_role_alter_user_nom_alter_…

- ⚠️ `CASCADE` sur `User.role` : supprimer un rôle supprimera tous les users associés. (Souvent on préfère `PROTECT`).
- ⚠️ Migration `0007` : identifiants admin “fixes” (`admin@exemple.com` / `admin123`).

  0007_administrateur_role

Si tu veux, je peux te proposer une version “propre prod” :

- garder la migration sans casser l’historique,
- mais déplacer la création admin vers une **commande Django** (ou un script de seed),
- et sécuriser `CASCADE` → `PROTECT`.

Sources
admin.py
Python
apps.py
Python
middleware.py
Python
models.py
Python
services.py
Python
signals.py
Python
tests.py
Python
urls.py
Python

views.py
Python
Voici le contenu du dossier app_historique
Réflexion durant une seconde

Parfait — **`app_historique`** est une app de **traçabilité (audit log)** qui enregistre :

- les actions **CREATE / UPDATE / DELETE** sur certains modèles via **signals**,
- et les **READ** (consultation de pages) via un **middleware**,
- avec une **rétention configurable** + exports **PDF/Excel/CSV**.

Voici le détail fichier par fichier.

1) `models.py` — tables historique + rétention

Tu as 2 modèles :

models

`HistoriqueRetentionSetting` (singleton)

- Une seule ligne forcée avec `pk=1`, champ `retention_years` (par défaut 2) + `updated_at`.
- Méthode `get_value()` qui crée la ligne si absente.

  models

`HistoriqueUser`

- FK vers `app_contact.User` (donc ton user “maison”).

  models

- Champs principaux : `action` (CREATE/READ/UPDATE/DELETE), cible (`app_label`, `model_name`, `object_id`, `object_repr`), date/heure auto, + trace technique (`url`, `method`, `ip`, `details`).
- Index sur `date_action`, `action`, `(app_label, model_name)`.

  models

2) `signals.py` — log automatique CREATE/UPDATE/DELETE

- Écoute `post_save` → écrit **CREATE** ou **UPDATE**.

  signals

- Écoute `post_delete` → écrit **DELETE**.

  signals

- Filtre : ignore l'app `app_historique` elle-même et ne trace que `{"app_acceuil", "app_contact", "app_user"}`.

  signals

- Récupère l'utilisateur courant via `get_current_user()` (thread-local défini dans le middleware).

3) `middleware.py` — récupération user + purge + log READ

Récupération de l'utilisateur courant

- Lit `request.session["user"]["id"]` (structure de session que tu utilises déjà dans `app_contact`) et charge `app_contact.models.User`.

  middleware

- Stocke l'objet user dans un `threading.local()` pour que les signals puissent le retrouver.

  middleware

Purge automatique (1 fois/jour)

- À chaque requête, déclenche `purge_old_history()` au max une fois par jour via clé session `hist_purge_YYYY-MM-DD`.

Log READ

Dans `process_response`, si :

- user connecté,
- requête GET,
- response < 400,
- pas `/static/`, `/media/`, `/favicon.ico`,
- pas `/fragment/` ni `/settings/` (pour éviter auto-log),
- et `Accept` contient `text/html`,
  alors crée une entrée `HistoriqueUser` avec action **READ**.

4) `services.py` — règle de rétention

`purge_old_history()` :

- conserve **N années** = année courante + (N-1) précédentes,
- supprime tout ce qui est **avant le 1er janvier** de l'année seuil.

services

Ex : si on est en 2026 et `N=2` → garde 2026 & 2025, supprime <= 2024.

5) `views.py` — fragment UI + settings + exports

- `_is_admin()` : autorise uniquement si `request.session["user"]["role"] == "Administrateur"`.

views

Affichage

- `historique_fragment` : rend un fragment template avec les **300 dernières lignes**, + infos sur les années gardées.

views

Settings (AJAX/POST)

- `historique_settings` :
  - `btn_save_retention` : clamp 1..10, sauvegarde le singleton, purge, retourne JSON {ok, deleted}.

views

- `btn_purge_history` : purge immédiate.

  views

Exports

- CSV : flux direct.

  views

- Excel : utilise `openpyxl` si installé, sinon 501.

  views

- PDF : utilise `reportlab` si installé, sinon 501.

  views

6) `urls.py`

Routes :

- `fragment/`, `settings/`
- `export/pdf/`, `export/excel/`, `export/csv/`

  urls

7) `admin.py` + `apps.py` + `tests.py`

- Admin : `HistoriqueRetentionSetting` et `HistoriqueUser` avec list_display/filter/search/order.

  admin

- `apps.py` active les signals dans `ready()`.

apps

- tests vides.

  tests

Points à surveiller (pratiques)

- **READ logging** : ça peut générer beaucoup de lignes (chaque page HTML vue) → tu as déjà des filtres anti-spam (assets + endpoints historiques) : bien.

  middleware

- **Sécurité admin** : `_is_admin()` dépend du contenu session `role` (string). Si un jour tu passes à un système de permissions plus strict, faudra basculer sur une vérif DB (`User.role.nom_role`).
- **`TRACK_APP_LABELS` contient `app_user`** mais je n'ai pas encore vu ses fichiers ici : si l'app n'existe pas dans ton projet, aucune conséquence (juste jamais match).

  signals

Si tu m'envoies aussi **`app_historique/migrations`**, je te confirme si tout est bien migré (tables, indexes, singleton retention) et si c'est aligné avec les modèles.

Sources
purge_historique.py
Python
Voici le contenu du dossier app_historique/management/commands
Réflexion durant quelques secondes

Parfait — dans **`app_historique/management/commands`**, tu as une **commande Django** pour lancer la purge manuellement.

`purge_historique.py`

- Déclare une commande `manage.py purge_historique`
- Elle appelle le service `purge_old_history()` (la logique de rétention)
- Puis affiche combien de lignes ont été supprimées

purge_historique

Concrètement, ça te permet de purger l'historique **sans passer par le middleware** (utile pour cron, maintenance, ou exécution ponctuelle).

Comment l'utiliser

Depuis la racine du projet Django :

```
python manage.py purge_historique
```

Ce que ça purge exactement

La purge supprime toutes les entrées `HistoriqueUser` dont `date_action` est **avant le 1er janvier** de l'année seuil calculée via `HistoriqueRetentionSetting.retention_years` (ex : si on est en 2026 et retention=2 → garde 2026 & 2025, supprime <= 2024).

Si tu veux, je peux aussi te proposer une version "production" :

- ajout d'un argument `--dry-run`,
- ajout d'un `--retention-years` temporaire,
- et logs plus détaillés (date seuil, nombre total avant/après).

Sources
0001_initial.py
Python
Voici le contenu du dossier app_historique/migrations
Réflexion durant une seconde

Parfait — dans **`app_historique/migrations`**, tu as (pour l'instant) une seule migration :

`0001_initial.py`

Cette migration initialise complètement l'app **historique** et dépend de `app_contact` (jusqu'à la migration `0007_administrateur_role`).

0001_initial

1) Création de `HistoriqueRetentionSetting`

- Table : `historique_retention_setting`
- Champs :
  - `id` : `PositiveSmallIntegerField` **pk=1** (editable=False) → logique de **singleton**
  - `retention_years` : entier positif, défaut **2**
  - `updated_at` : `auto_now=True`

    0001_initial

✅ C'est cohérent avec ton modèle "une seule config de rétention".

2) Création de `HistoriqueUser`

- Table : `historique_user`
- Champs :
  - `action` : choix CREATE/READ/UPDATE/DELETE
  - infos cible : `app_label`, `model_name`, `object_id`, `object_repr`
  - date/heure : `date_action` (auto_now_add), `heure_action` (auto_now_add)
  - trace requête : `url`, `method`, `ip`, `details`
  - `user` : FK vers `app_contact.User` (`CASCADE`, related_name=`historiques_actions`)

    0001_initial

Options & perf :

- `ordering = ['-date_action', '-heure_action']`
- Index :
  - `date_action`
  - `action`
  - `(app_label, model_name)`

    0001_initial

✅ C'est aligné avec ce que tu utilises dans le middleware + signals (tri récent, filtres par action/app/modèle, purge par date).

Petit point à noter (fonctionnel)

Dans la migration, `date_action` est un **DateField** (pas DateTime), et `heure_action` est séparé.

0001_initial

Ça marche très bien, mais si un jour tu veux filtrer “entre 14:03 et 14:10” sur une même date, ce sera un peu moins pratique qu’un `DateTimeField`. (Pas bloquant, juste un choix de design.)

Si tu veux, je peux te dire aussi si ton `services.purge_old_history()` est parfaitement cohérent avec ce choix `DateField` (spoiler : oui, car tu compares par “année”/seuil de date).

Sources
admin.py
Python
apps.py
Python
models.py
Python
tests.py
Python
urls.py
Python
views.py
Python
Voici le contenu du dossier app_user
Réflexion durant une seconde

Parfait — j’ai bien reçu **`app_user`**. Cette app sert surtout à gérer une **navbar dynamique / permissions par rôle** + une page de **gestion des utilisateurs** (en grande partie via un module externe `PackageUser.modul_new_user`).

1) `models.py` — Menus + Matrice d’accès par rôle

Tu as 2 modèles :

models

`Menu`

- `nom`
- `code_menu` unique
- `__str__` retourne le nom

  models

`RoleMenuAcces`

- table de liaison **Rôle ↔ Menu**
- champs :
  - `role` (FK vers `app_contact.Role`, `CASCADE`)
  - `menu` (FK vers `Menu`, `CASCADE`)
  - `est_visible` (bool, défaut False)
- contrainte `unique_together = ('role', 'menu')`

  models

Initialisation automatique via signals (post_save)

- Quand tu crées un **Role**, tu crées automatiquement toutes les lignes `RoleMenuAcces` correspondantes à tous les menus (et inversement quand tu crées un **Menu**)

  models

- Règle spéciale : pour **Administrateur × "Gestion des Utilisateurs"**, `est_visible=True` par défaut.

  models

✅ Ça évite les "trous" (pas besoin de remplir manuellement les permissions pour les nouveaux rôles/menus).

2) `views.py` — page centrale + update permissions

Le fichier contient 2 choses principales :

views

`gestion_globale`

- Affiche :
  - liste des users (`select_related("role")`)
  - tous les rôles
  - tous les menus
  - tous les contacts (triés du plus récent au plus ancien)
- Construit `acces_existants` = liste de clés `"roleId_menuId"` pour les cases cochées.
- Calcule `mes_menus_visibles` pour l'utilisateur connecté via `request.session["user"]["id"]` puis en lisant `RoleMenuAcces` (est_visible=True).

views

`update_menus`

- Parcourt tous les couples (role, menu) et lit les checkboxes `access_<roleid>_<menuid>` depuis POST.
- Force **Administrateur × Gestion des Utilisateurs** à `True` quoi qu'il arrive.
- Fait `update_or_create` pour enregistrer `est_visible`.

views

`add_user`

- Délègue la création user au module externe `GestionnaireUtilisateur` (dans `PackageUser.modul_new_user`).

views

3) `urls.py` — routes (avec appels lambda vers un gestionnaire externe)

Tu instancies `GestionnaireUtilisateur()` puis tu routes vers des méthodes de l'instance via `lambda`.

urls

- `root_gestion_utilisateur/` → `_Fonction_Page_gestion_des_Utilisateurs`
- `user/add/` → `creer_un_nouvel_utilisateur`
- `user/edit/<id>/`, `user/delete/<id>/` → edit/delete
- `user/update-menus/` → `update_menus`

⚠ Note : les lambdas marchent, mais c'est atypique en Django et ça rend parfois le debug/stacktrace moins clair. Une alternative propre serait d'exposer des wrappers de vues normales qui appellent `gestionnaire.method(...)`.

4) `admin.py`, `apps.py`, `tests.py`

- `admin.py` vide (rien enregistré).

  admin

- `apps.py` standard.

  apps

- tests vides.

  tests

Points importants à surveiller

- **Redondance `User` import** : dans `views.py`, tu importes `User` depuis `.models`, mais ton `models.py` n'a pas de modèle `User` (tu relies les users depuis `app_contact`). Ça peut être une incohérence (ou un ancien reste) selon ton projet complet.
- **Dépendance externe** : une grosse partie du CRUD user est dans `PackageUser/modul_new_user.py` (non fourni ici). Donc l'app `app_user` dépend fortement de ce package.

Si tu m'envoies aussi **`app_user/migrations`** (ou si tu me confirmes qu'il n'y en a pas), je peux te dire si tes tables `Menu` / `RoleMenuAcces` existent bien en DB et si tout est cohérent avec les imports (`User`, `Role`, etc.).

Sources
0001_initial.py
Python
0002_setup_menus.py
Python
0003_alter_menuacces_user_delete_user.py
Python
0004_alter_menu_code_menu_rolemenuacces_delete_menuacces.py
Python
0005_seed_rolemenuacces.py

Python
Voici le contenu du dossier app_user/migrations
Réflexion durant quelques secondes

Bien reçu — tes migrations **`app_user/migrations`** montrent une refonte progressive : d'abord un système "accès par user", puis migration vers "accès par rôle" (ce que tu as maintenant dans `models.py`).

Chronologie des migrations

`0001_initial.py` — (ancien design) Menus + Users + accès par user

Crée 3 modèles :

0001_initial

- `Menu(nom, code_menu unique)`
- `User(nom, email unique, password)`
- `MenuAcces(user FK -> app_user.User, menu FK -> Menu, est_visible)` avec `unique_together(user, menu)`

✅ Donc au départ, les permissions étaient **par utilisateur**.

`0002_setup_menus.py` — seed des menus par défaut

Ajoute une `RunPython` qui crée 4 menus :

0002_setup_menus

- "Gestion des Utilisateurs" (`menu_ventes`)
- "Gestion d'Affichage du Site" (`menu_stocks`)
- "Factures" (`menu_factures`)
- "Rapports" (`menu_rapports`)

⚠️ Les `code_menu` ("menu_ventes", "menu_stocks"…) ne correspondent pas aux noms (ça marche, mais c'est un peu trompeur).

`0003_alter_menuacces_user_delete_user.py` — raccord avec `app_contact.User`

Dépend de `app_contact 0003_alter_user_email` et :

0003_alter_menuacces_user_delet…

- remplace `MenuAcces.user` → FK vers **`app_contact.User`**
- supprime le modèle `app_user.User`

✅ Donc l'app commence à utiliser ton **User "maison" central** (dans `app_contact`) au lieu d'avoir un User dupliqué.

`0004_alter_menu_code_menu_rolemenuacces_delete_menuacces.py` — passage au design par rôle

Dépend de `app_contact 0007_administrateur_role` et :

0004_alter_menu_code_menu_rolem…

- agrandit `Menu.code_menu` de 50 à 100 caractères
- crée `RoleMenuAcces(role FK -> app_contact.Role, menu FK -> Menu, est_visible)` avec `unique_together(role, menu)`
- supprime `MenuAcces`

✅ C'est exactement le modèle actuel dans ton `models.py` (`RoleMenuAcces`).

models

`0005_seed_rolemenuacces.py` — seed complet des couples rôle×menu

RunPython :

0005_seed_rolemenuacces

1. crée toutes les combinaisons (role, menu) avec `est_visible=False`
2. force **Administrateur** × **“Gestion des Utilisateurs”** à `True`

✅ C’est cohérent avec tes signaux dans `models.py` qui font la même chose pour les nouveaux rôles/menus.

models

Ce que ça implique (cohérence & petit point à corriger)

- ✅ Ta DB est bien passée de “permissions par user” → “permissions par rôle”.
- ✅ `RoleMenuAcces` est bien seeded (migration 0005) **et** maintenu ensuite automatiquement (signals).

⚠️ **Incohérence probable dans `views.py`** : tu importes `User` depuis `.models`, alors que ton `models.py` actuel **n’a plus de modèle `User`** (il a `Menu` + `RoleMenuAcces` seulement). Ça peut casser au runtime selon ton code exécuté.
👉 Fix simple : remplacer `from .models import User, Menu, RoleMenuAcces` par `from app_contact.models import User` + `from .models import Menu, RoleMenuAcces`.

Si tu veux, je te propose un patch exact (diff) pour nettoyer `views.py` + sécuriser les imports (et éventuellement enlever les vieux restes “Gemini links” dans les fichiers).

Sources
admin_extras.py
Python
Voici le contenu du dossier app_user/templatetags
Réflexion durant une seconde

Parfait — dans **`app_user/templatetags`**, tu as un seul fichier : **`admin_extras.py`**. Il ajoute des helpers pour tes templates Django (typiquement la page de gestion des menus/permissions).

`admin_extras.py`

1) Tag `check_access`

Un **`simple_tag`** qui renvoie `True/False` selon que la clé `"roleId_menuId"` existe dans `acces_existants`.

admin_extras

- `row_id` = id du rôle (ligne)
- `menu_id` = id du menu (colonne)
- `acces_existants` = une liste/tuple/set de chaînes comme `"3_7"` que tu construis dans `views.py` (`acces_existants.append(f"{role_id}_{menu_id}")`)

Donc dans ton template, tu peux faire un truc du genre :

```
{% check_access role.id menu.id acces_existants as is_checked %}
<input type="checkbox" {% if is_checked %}checked{% endif %}>
```

2) Filtre `get_item`

Un filter utilitaire : `{{ mon_dict|get_item:key }}`.

- Si l'objet n'est pas un dict → renvoie `[]`
- Sinon renvoie `dictionary.get(key, [])`

admin_extras

Remarque pratique

Le tag `check_access` est robuste (vérifie que `acces_existants` est bien une collection).

admin_extras

Si tu veux optimiser, tu peux t'assurer côté view que `acces_existants` soit un **set** (au lieu d'une liste) pour rendre le `in` toujours O(1) (mais c'est déjà OK vu la taille probable).
Sources
modul_new_user.py
Python
Voici le contenu du dossier PackageUser
Réflexion durant une seconde

Parfait — **`PackageUser`** contient (au moins) le fichier **`modul_new_user.py`** qui centralise le **CRUD des utilisateurs** + l'affichage de la page "Gestion des Utilisateurs", et sert de "backend" à ton app `app_user`.

`PackageUser/modul_new_user.py` — rôle & fonctionnement

1) `GestionnaireUtilisateur.creer_un_nouvel_utilisateur(...)`

C'est un "routeur" interne : selon `nom_du_template`, il appelle :

- `_add_user()` si `page_utilisateur.html`
- `_index_inscription_view()` si `index.html`
- sinon renvoie `404.html`

  modul_new_user

⚠ Dans le fichier fourni, **`_index_inscription_view()` n'apparaît pas** (peut-être plus bas dans ton vrai fichier ou dans une autre partie du package). Si elle n'existe pas réellement, l'appel plantera quand tu passes `"index.html"`.

modul_new_user

---

2) `_Fonction_Page_gestion_des_Utilisateurs(...)` — rendu page admin users/permissions/contacts

Cette méthode construit toutes les données nécessaires au template `page_utilisateur.html` :

modul_new_user

- `users` : liste des utilisateurs (`select_related("role")`)
- `menus` : tous les menus
- `contacts` : tous les messages contact triés par date
- `roles` : tous les rôles
- `acces_existants` : toutes les permissions actives (`RoleMenuAcces.est_visible=True`) sous forme `"roleId_menuId"`

Elle calcule aussi `mes_menus_visibles` pour l'utilisateur connecté via la session (`request.session["user"]["id"]`) puis filtre sur son `role_id`.

modul_new_user

✅ **Contrôle d'accès** : l'ouverture de la page est autorisée seulement si l'utilisateur a un accès visible au menu `"Gestion des Utilisateurs"`. Sinon redirection vers `name_acceuil`.

modul_new_user

👉 Ça correspond bien au `templatetag check_access` côté template (pour cocher les cases).

admin_extras

---

3) `_add_user(...)` — création user avec validations + role FK

Logique propre :

- valide nom/email/password/role
- vérifie unicité email
- transforme le `role` (texte) → **objet `Role`** via `Role.objects.get(nom_role=...)`
- crée le user avec `make_password(password)`
- message succès puis redirect `gestion_globale`

  modul_new_user

---

4) `edit_user(...)` — mise à jour user existant

- récupère le user via `get_object_or_404`
- valide email format + unicité (en excluant l'ID courant)
- **exige un mot de passe** (actuellement il est obligatoire)

  modul_new_user

- si le nouveau mdp ne commence pas par `pbkdf2_`, alors il est re-hashé

En cas d'erreurs, renvoie vers `_Fonction_Page_gestion_des_UtilisateursEDIT(...)` qui ré-affiche `page_utilisateur.html` avec les erreurs pour rouvrir la modale côté JS.

modul_new_user

⚠ Bug/fragilité probable : tu passes `roles` au template sous la clé `'roles': role_obj` (qui est une queryset) mais ailleurs tu passes les rôles via `roles = Role.objects.all()` (même nom de clé). Là tu as deux styles (`role_obj` vs `roles`), il faut que ton template soit cohérent.

modul_new_user

---

5) **`delete_user(...)`**

Supprime un user sur POST, et affiche un message.

modul_new_user

Points importants à améliorer (pratiques)

1. **Mot de passe obligatoire en édition**
   Actuellement, si tu edits un user sans retaper de mdp, ça déclenche une erreur "obligatoire". Souvent on préfère : "si vide → ne pas changer". Là tu as déjà une logique de "ne changer que si fourni", mais tu bloques avant avec une validation stricte.

   modul_new_user

2. **`_index_inscription_view` manquant (si réellement absent)**
   Si tu veux que `creer_un_nouvel_utilisateur(..., "index.html")` marche, il faut que cette méthode existe.
3. **Nettoyage**
   Il y a des commentaires Gemini + `print()` (ok en dev, mais à remplacer par `logging` en prod).

   modul_new_user

Si tu veux, je peux te proposer un patch concret pour :

- rendre le mdp **optionnel** en édition,
- ajouter (ou supprimer proprement) `_index_inscription_view`,
- remplacer les `print()` par un logger,
- et rendre le rendu `roles`/`role_obj` cohérent côté template.

Sources
asgi.py
Python
settings.py
Python
urls.py
Python
wsgi.py
Python
Voici le contenu du dossier portfolio
Réflexion durant une seconde

Bien reçu — le dossier **`portfolio/`** correspond au **cœur du projet Django** (settings + urls + asgi/wsgi). Voilà ce qu'il contient et comment ça s'assemble.

`asgi.py`

Config standard ASGI : définit `DJANGO_SETTINGS_MODULE='portfolio.settings'` et expose `application = get_asgi_application()`.

asgi

`wsgi.py`

Config standard WSGI (même principe que ASGI).

wsgi

`settings.py`

Points clés :

settings

Apps installées

Tu as bien enregistré tes 4 apps :

- `app_acceuil`
- `app_contact`
- `app_user`
- `app_historique`

  settings

Middleware

Tu as ajouté ton middleware d'audit :

- `app_historique.middleware.HistoriqueCurrentUserMiddleware` placé juste après `SessionMiddleware` (bon endroit : la session est déjà disponible).

  settings

Base de données

- SQLite3 (`db.sqlite3`).

  settings

Static & Media

- `STATIC_URL = 'static/'`
- `STATICFILES_DIRS = [BASE_DIR / 'static/']` (via `os.path.join(BASE_DIR,'static/')`)
- `MEDIA_URL = '/media/'`
- `MEDIA_ROOT = BASE_DIR / 'media'` (via `os.path.join(BASE_DIR,'media')`)

  settings

⚠ Détail : tu importes `os` deux fois dans ce fichier (pas grave, juste à nettoyer).

settings

Sécurité / prod

- `DEBUG = True`
- `ALLOWED_HOSTS = []`
- `SECRET_KEY` est **en clair** dans le fichier.

  settings

  ➡️ OK en dev, mais en prod il faut le sortir en variable d'environnement.

`urls.py`

Tu inclus toutes les apps **sur la racine** :

urls

- `path('', include("app_acceuil.urls"))`
- `path('', include("app_contact.urls"))`
- `path('', include("app_user.urls"))`
- `path('', include("app_historique.urls"))`

Et tu sers `MEDIA_URL` en debug.

urls

⚠️ Point à surveiller : collisions d'URLs

Comme tu inclus **tout sur** `''`, si deux apps définissent la même route (ex: `path('', ...)` ou `path('dashboard/', ...)`), Django prendra **la première qui matche dans l'ordre** (ici `app_acceuil` avant `app_contact` avant `app_user` avant `app_historique`).

urls

Ce n'est pas forcément un bug, mais ça peut expliquer des comportements "bizarres" si tu ajoutes de nouvelles routes.

Si tu veux, je peux te proposer une version plus propre de `portfolio/urls.py` avec des **préfixes** (ex: `/contact/`, `/users/`, `/historique/`) pour éviter toute collision, tout en gardant la home publique sur `/`.

Sources
diapo_Javascript_Page1.html
Fichier
email_Javascript.htm
Fichier
index.html
Fichier
page_administration.html
Fichier
page_inscription.html
Fichier
page_utilisateur.html
Fichier
TransportRoutier_Main.html
Fichier
Voici le contenu du dossier templates
Réflexion durant quelques secondes

Bien reçu — le dossier **`templates/`** contient **7 fichiers HTML** (templates Django + quelques pages “démo” transport). Voici à quoi ils servent, et comment ils s’intègrent au projet.

1) `index.html` (page publique principale)

C’est ta **page d’accueil publique** du portfolio : elle **hérite** de `layouts/main.html` et remplit les blocs `title`, `desc`, `contenu`.

index

On voit aussi que tu affiches des infos utilisateur si `utilisateur.nom` et `utilisateur.email` sont présents.

index

2) `page_administration.html` (dashboard admin “Portfolio”)

C’est une **interface d’administration** (onglets Bootstrap) pour gérer :

- Profil & bio (photo + “à propos” + visibilité),
- Spécialité,
- Réalisations,
- Témoignages,
- Compétences,
- Contact,
- Stats vues projets,
- Graphique,
- Exporter Données,
- Historique utilisateur.

page_administration

Tu as aussi une logique d’affichage des boutons selon `request.session.user.role` (Administrateur / Modérateur / autre).

page_administration

3) `page_inscription.html` (popup d’inscription)

Une page qui affiche un bouton “S’inscrire maintenant” et ouvre une **popup responsive** avec formulaire (`csrf_token`, champs, gestion des erreurs).

page_inscription

Le `form` envoie vers `{% url 'root_gemini_accueil' %}`.

page_inscription

4) `page_utilisateur.html` (admin centrale utilisateurs / permissions / contacts)

Page “Administration Centrale” avec onglets :

- Utilisateurs (CRUD via modales),
- Permissions menus,
- Messages.

page_utilisateur

Elle charge `admin_extras` (`check_access`, etc.) pour gérer l'affichage des accès.

⚠ Point sécurité : dans le tableau, tu affiches `u.password` dans un `<input type="password">`. Même masqué, c'est le hash stocké en DB : ce n'est pas utile côté UI et ça peut être récupéré via inspecteur.

page_utilisateur

5) `email_Javascript.htm` (formulaire "Contact / Devis")

Page "Contact / Devis" avec un formulaire et un `onsubmit="submitContact(event)"` (logique JS attendue).

email_Javascript

Les liens de navbar pointent vers des `url` Django avec ancres (ex: `{% url 'name_Page_accueil_section_contact' %}#contact`).

email_Javascript

6) `diapo_Javascript_Page1.html` (diaporama + menus multi-niveaux)

Une page démo avec :

- carrousel W3.CSS (3 images),
- navbar Bootstrap avec **menus à plusieurs niveaux** (dropdown-submenu),
- CSS popup overlay, cards flexbox responsive, etc.

diapo_Javascript_Page1

Je vois aussi une version "django-ifiée" où des liens ont été remplacés par `{% url ... %}` et `/#accueil` etc. (même structure mais plus intégré Django).

TransportRoutier_Main

7) `TransportRoutier_Main.html` (page “camions transport”)

Page démo “Transport routier” (liens depuis le menu “Types de camions” dans les diapos), vraisemblablement une page longue avec ancres (ex: `#Le_camion_tautliner`).

TransportRoutier_Main

Comment Django les trouve

Dans `portfolio/settings.py`, tu as `TEMPLATES['DIRS'] = ['templates/']`, donc Django charge directement ce dossier.

settings

Si tu me dis **ce que tu veux faire ensuite** (ex: “brancher `email_Javascript.htm` sur `app_contact`”, “nettoyer les routes”, “corriger l’affichage mot de passe”, “uniformiser layout/main.html”), je te propose un plan + les modifications exactes (views + urls + templates).

Sources
Voici le contenu du dossier templates/app_historique/fragment_historique.html : <h4 class="mb-3">Historique Utilisateur</h4>   <label class="form-label fw-semibold">Durée de conservation (années)</label> <input id="retention_years" type="number" class="form-control" min="1" max="10" value="{{ retention_years }}" required> <small class="text-muted"> Exemple : 2 ans en {{ current_year }} ⇒ conserver {{ years_kept }} ; supprimer avant. </small>   <button class="btn btn-primary" onclick="saveRetention()">Enregistrer</button> <button class="btn btn-outline-danger ms-2" onclick="purgeHistory()">Purger maintenant</button>   <!-- Résumé visible de la conservation --> Conservation actuelle : {{ retention_years }} an(s) — Année courante : {{ current_year }} — Années conservées : {{ years_kept }}   <table class="table table-striped align-middle"> <thead> <tr> <th>DATE</th> <th>HEURE</th> <th>Utilisateur</th> <th>Action</th> <th>App</th> <th>Modèle</th> <th>Objet</th> <th>URL</th> </tr> </thead> <tbody> {% for h in history_rows %} <tr> <td>{{ h.date_action }}</td> <td>{{ h.heure_action }}</td> <td>{{ h.user.email }}</td> <td>{{ h.action }}</td> <td>{{ h.app_label }}</td> <td>{{ h.model_name }}</td> <td>{{ h.object_repr }}</td> <td>{{ h.url }}</td> </tr> {% empty %} <tr><td colspan="8" class="text-center text-muted">Aucun historique.</td></tr> {% endfor %} </tbody> </table>

Bien reçu ✅ Ton template `templates/app_historique/fragment_historique.html` est un **fragment Bootstrap** qui affiche :

Ce qu'il fait

1) Bloc "Paramètres de conservation"

- Champ numérique `retention_years` (min 1 / max 10) alimenté par `{{ retention_years }}`.
- Deux boutons JS :
  - `saveRetention()` → enregistrer la durée de conservation
  - `purgeHistory()` → purger tout de suite
- Zone de message `#hist_msg` pour afficher un retour utilisateur.

Le texte d'aide explique la logique : garder `years_kept` (ex: 2 ans en année courante → garder 2 années). ✅

2) Résumé visible (alert info)

Affiche :

- Conservation actuelle
- Année courante
- Liste/texte des années conservées via `{{ years_kept }}`

3) Table historique

Affiche une table avec colonnes :

- date, heure
- utilisateur (`h.user.email`)
- action (badge)
- app_label / model_name
- object_repr
- url

Et un fallback `{% empty %}`.

Petites améliorations simples (si tu veux)

- **Badge par action** : mettre une couleur différente selon CREATE/UPDATE/DELETE/READ (plus lisible).
- **URL cliquable** : transformer `{{ h.url }}` en lien `<a href="{{ h.url }}" ...>` (souvent utile).
- **Heure** : si `h.heure_action` est un `TimeField`, tu peux formater : `{{ h.heure_action|time:"H:i:s" }}`.
- **Protection email** : si tu veux éviter d'afficher l'email complet, tu peux tronquer (ex: filtre custom).

Important : où sont `saveRetention()` et `purgeHistory()` ?

Ce fragment suppose que ces fonctions JS existent déjà (dans un `<script>` du `page_administration.html` ou un fichier `.js`).
Si tu me montres le JS correspondant (ou `page_administration.html` autour du bloc historique), je peux vérifier que les URLs appelées correspondent bien à :

- `/settings/` (POST)
- `/fragment/` (GET)
  et que le `csrf_token` est bien géré côté fetch/AJAX.

footer.html
Fichier
header.html
Fichier
main.html
Fichier
Voici le contenu du dossier templates/layouts
Réflexion durant quelques secondes

Bien reçu ✅ Le dossier **`templates/layouts/`** contient tes **3 templates "socle"** (base du site) :

1) `main.html` (layout principal / base template)

C'est ton **template maître** : il charge `{% load static %}`, définit le `<head>` (meta, Bootstrap, FontAwesome, flag-icons…), puis inclut :

- `layouts/header.html`

- un bloc `{% block "contenu" %}{% endblock %}`
- `layouts/footer.html`

  main

Tu as aussi :

- une meta `csrf-token` (utile pour fetch/AJAX)

  main

- beaucoup de CSS “global” directement dans la page (navbar, hero, projets, etc.)

  main

- un script JS qui gère les **dropdown-submenus** en mobile avec `matchMedia` (ouvre/ferme les sous-menus sans casser le comportement desktop)

  main

👉 Conclusion : `main.html` est la **base** que tes pages comme `index.html` étendent.

---

2) `header.html` (navbar + menus dynamiques)

Contient la **barre de navigation Bootstrap** fixe en haut, avec :

Menu “Portfolio”

Liens vers des ancres sur la page d’accueil (`#accueil`, `#projets`, `#temoignages`, etc.) via `{% url 'name_acceuil' %}#...`

header

Menu “Menus dynamiques” (prototype)

Boucle sur `mes_menus_visibles` et affiche certains items selon `acces.menu.nom` (ex: “Gestion des Utilisateurs”, “Gestion d’Affichage du Site”, etc.)

header

Menu “Administration” (lié aux droits)

Même logique : boucle sur `mes_menus_visibles` mais avec des liens concrets :

- “Gestion des Utilisateurs” → `{% url 'gestion_globale' %}`
- “Gestion d’Affichage du Site” → `{% url 'dashboard_admin' %}`

  header

Bloc connexion / déconnexion

Si `request.session.user` existe :

- affiche le nom tronqué via `.truncate-text` et le rôle
- lien déconnexion `{% url 'deconexion' %}`

  header

  Sinon :

- liens “S’inscrire” (ancre `#sinscrire`) et “Connexion” (`loginpage`)

  header

---

3) `footer.html` (pied de page)

Très simple : copyright “2025 | Portfolio de Halley Mario …”

footer

Petit point d’architecture (utile pour la suite)

Actuellement, `main.html` contient **beaucoup** de CSS/JS inline. Ça marche, mais si tu veux rendre le projet plus “propre Django”, on pourra déplacer ça dans `static/css/...` et `static/js/...` (et ne garder dans `main.html` que les `<link>` / `<script src>`).

main

Si tu veux, dis-moi : tu veux que je t’aide à **rendre les menus dynamiques vraiment automatiques** (liens + filtrage par rôle), ou tu veux d’abord **brancher l’historique** (boutons `saveRetention()` / `purgeHistory()`) dans l’admin ?

Sources
db.sqlite3
Fichier
Voici la base de données
Réflexion durant 37s

Parfait — j’ai pu ouvrir ta base **`db.sqlite3`** et vérifier son contenu. Voilà un état des lieux clair.

Tables présentes (25)

On retrouve bien tes apps + les tables Django standard :

- **app_acceuil** : `projetscards`, `projetsfirstspeciality`, `projetmesrealisations`, `projetphotodeprofil`, `projetaproposdemoi`, `mescompetencescles`, `reseausocial`
- **app_contact** : `contact`, `role`, `user`
- **app_user** : `menu`, `rolemenuacces`
- **app_historique** : `historique_retention_setting`, `historique_user`
- **django/auth** : `auth_*`, `django_*`, `django_session`, etc.

Volume des données (lignes)

- `app_acceuil_projetscards` : **3**
- `app_acceuil_projetsfirstspeciality` : **1**
- `app_acceuil_projetmesrealisations` : **4**
- `app_acceuil_projetphotodeprofil` : **1**
- `app_acceuil_projetaproposdemoi` : **1**

- `app_acceuil_mescompetencescles` : **4**
- `app_acceuil_reseausocial` : **2**
- `app_contact_role` : **4**
- `app_contact_user` : **4**
- `app_contact_contact` : **0**
- `app_user_menu` : **4**
- `app_user_rolemenuacces` : **16** (4 rôles × 4 menus)
- `historique_retention_setting` : **1**
- `historique_user` : **301**
- Tables Django `auth_user`, `auth_group` : **0** (tu n'utilises pas l'auth Django par défaut)

Rôles et utilisateurs (app_contact)

Rôles

1. Administrateur
2. Moderateur
3. Utilisateur
4. Responsable de Saisie

Utilisateurs

- `admin@exemple.com` → rôle **Administrateur**
- `gregoire@yahoo.fr` → rôle **Moderateur**
- `john@hotmail.com` → rôle **Moderateur**
- `jerry@gmail.fr` → rôle **Utilisateur**

Menus & permissions (app_user)

Menus (4)

- Gestion des Utilisateurs (`menu_ventes`)
- Gestion d'Affichage du Site (`menu_stocks`)
- Factures (`menu_factures`)
- Rapports (`menu_rapports`)

Exemple de permissions

Pour le rôle **Administrateur**, les accès sont majoritairement activés (je vois au moins les 3 premiers menus en `est_visible=1`, “Rapports” est à `0` dans les premières lignes affichées).
Comme tu as 16 lignes au total, c’est bien une matrice complète rôle×menu.

Historique (app_historique)

- **Rétention** : `retention_years = 3` (mise à jour le **2026-02-12**)
- **Dernières traces** : beaucoup de `READ` sur `/dashboard/`, `/` et `/deconnexion/`, et aussi des `UPDATE` sur des objets de `app_acceuil` (ex : `Projetphotodeprofil`).

Si tu me dis ce que tu veux faire maintenant avec cette base (ex : “voir toutes les permissions par rôle”, “exporter les projets”, “comprendre pourquoi l’historique a 301 lignes”, “trouver quel user a fait tel update”), je te sors les requêtes exactes + un diagnostic.

Pouvez vous générer toute l'ANALYSE MERISE COMPLETE correspondant (Modèle Conceptuel des Données, Schéma de Circulation des Documents, ...) avec toutes les REGLES DE GESTION au très GRAND COMPLET , si cela vous est possible ? s'il vous plaît

Oui — je peux te générer une **analyse MERISE complète** à partir de ce que tu as partagé (structure Django : `app_acceuil`, `app_contact`, `app_user`, `app_historique`, templates, migrations).
Je vais donc te donner :

1. **Règles de gestion (RG) ultra complètes**
2. **Dictionnaire des données**
3. **MCD (Modèle Conceptuel des Données)** (entités, attributs, associations, cardinalités)
4. **MLD (Modèle Logique)** (relations) + contraintes
5. **MPD (Modèle Physique)** (tables, index, clés)
6. **Schéma de circulation des documents / flux** (utilisateur ↔ système ↔ BD)
7. **MCT (Modèle Conceptuel des Traitements)** (événements, opérations, résultats)
8. Matrices utiles (CRUD, droits/menus, traçabilité)

Remarque : je “déduis” les flux (documents / écrans) depuis tes vues + templates + migrations. Si tu as des écrans non fournis, ils ne seront pas dans le flux.

---

1) Périmètre fonctionnel (ce que couvre le SI)

Le système est un **portfolio** avec :

- **Contenu public** (page d'accueil) : profil, à-propos, spécialité, réalisations, témoignages, compétences, réseaux sociaux.
- **Gestion du contenu** via un **dashboard** (Administrateur/Modérateur).
- **Gestion des utilisateurs** (création/édition/suppression) + **droits d'accès par rôle** (menus visibles).
- **Authentification "maison"** (sessions, mot de passe hashé).
- **Historisation / audit** :
  - CREATE/UPDATE/DELETE (signals)
  - READ (middleware)
  - purge selon une durée de conservation
  - export CSV/Excel/PDF

---

2) Règles de gestion (RG) – très détaillées

2.1 Gestion des utilisateurs & rôles

RG-U1 : Un utilisateur est défini par (nom, email, mot de passe, rôle).
RG-U2 : L'email utilisateur est **unique** et normalisé en **minuscule** à l'enregistrement.
RG-U3 : Le mot de passe est stocké **haché** (pas en clair).
RG-U4 : Un utilisateur appartient à **un seul rôle**.
RG-U5 : Un rôle peut être attribué à **plusieurs utilisateurs**.
RG-U6 : Les rôles de base existent : Administrateur, Moderateur, Utilisateur, Responsable de Saisie.
RG-U7 : La session contient au minimum l'identifiant utilisateur et le rôle pour gérer les accès.

2.2 Menus & droits (contrôle d'accès)

RG-M1 : Un menu est défini par (nom, code_menu unique).
RG-M2 : Les droits ne sont pas par utilisateur mais par **rôle**.
RG-M3 : L'accès à un menu pour un rôle est matérialisé par `RoleMenuAcces(est_visible)`.
RG-M4 : Il existe **une seule ligne** par couple (rôle, menu). (unicité)

RG-M5 : Lors de la création d'un rôle ou d'un menu, le système crée automatiquement toutes les lignes d'association manquantes (matrice complète).
RG-M6 : Le menu "Gestion des Utilisateurs" doit être **toujours visible** pour le rôle Administrateur (non désactivable).

2.3 Gestion du contenu du portfolio (app_acceuil)

Visibilité

RG-A1 : Les éléments du portfolio possèdent un indicateur `est_visible`.
RG-A2 : La page publique n'affiche que les éléments `est_visible=True`.
RG-A3 : Les éléments masqués restent en base mais ne sont pas visibles sur le site public.

Témoignages (Projetscards)

RG-A4 : Un témoignage contient (nom, description, image optionnelle).
RG-A5 : Un témoignage est identifié fonctionnellement par un **fingerprint** unique (anti-doublon).
RG-A6 : Le fingerprint est calculé à partir du nom + description normalisés (ex : trim, lower, etc.) puis hash (SHA-256).
RG-A7 : Deux témoignages identiques au sens (nom+description) ne peuvent pas coexister.

Spécialité (Projetsfirstspeciality)

RG-A8 : Spécialité = (image optionnelle, description optionnelle).
RG-A9 : Le système impose un comportement "singleton" côté administration : **une seule spécialité** active en base (au moins via la restriction d'ajout).

Réalisations (Projetmesrealisations)

RG-A10 : Une réalisation contient (nom, description, image optionnelle).
RG-A11 : Une réalisation possède un compteur `compteur_demo_live` (suivi des clics/visites de démo).
RG-A12 : Le compteur est incrémenté via un endpoint "demo-live".
RG-A13 : Le compteur peut être réinitialisé depuis l'administration.

Photo de profil (Projetphotodeprofil)

RG-A14 : La photo de profil est gérée comme singleton : **une seule** photo de profil.

À propos (ProjetAproposDeMoi)

RG-A15 : “À propos” est un texte géré comme singleton : **un seul** bloc.

Compétences (MesCompetencesCles)

RG-A16 : Une compétence est définie par un nom unique.
RG-A17 : Les compétences peuvent être rendues visibles / invisibles.
RG-A18 : L’ordre d’affichage est par défaut du plus récent au plus ancien (ordering).

Réseaux sociaux (ReseauSocial)

RG-A19 : Un réseau social = (nom unique, url, visible).
RG-A20 : Seuls les réseaux `visible=True` s’affichent sur la page publique.

2.4 Contact

RG-C1 : Un message de contact contient (nom, email, message, date_envoi).
RG-C2 : L’email du contact est normalisé en minuscule à l’enregistrement.

2.5 Historique / Audit

Traçabilité CRUD (signals)

RG-H1 : Toute création d’un objet suivi génère un log CREATE.
RG-H2 : Toute modification génère un log UPDATE.
RG-H3 : Toute suppression génère un log DELETE.
RG-H4 : Les apps suivies incluent notamment : `app_acceuil`, `app_contact`, `app_user` (et potentiellement d’autres si configurées).
RG-H5 : Le log stocke : utilisateur (si connu), action, app, modèle, id, représentation, date/heure, détails.

Traçabilité des consultations (middleware)

RG-H6 : Une consultation de page HTML (GET, code < 400, non static/media…) génère un log READ.
RG-H7 : Les endpoints techniques (static/media, fragment/settings) sont exclus.
RG-H8 : Le log READ contient l’URL, la méthode, l’IP et le user si connecté.

Conservation et purge

RG-H9 : La durée de conservation est configurable (1 à 10 ans).
RG-H10 : Une purge supprime toutes les traces plus anciennes que le seuil calculé (selon année courante – N).
RG-H11 : La purge peut être déclenchée :

- automatiquement (au plus 1 fois/jour par session)
- manuellement via page settings
- via commande Django `purge_historique`.

Export

RG-H12 : L'historique peut être exporté en CSV / Excel / PDF.
RG-H13 : Si la librairie nécessaire n'est pas installée (openpyxl/reportlab), l'export correspondant est indisponible.

2.6 Import/Export JSON du portfolio (bundle)

RG-J1 : Le système exporte/import un bundle JSON "portfolio" et un bundle "users".
RG-J2 : Les images/fichiers sont encodés en base64 à l'export, restaurés à l'import.
RG-J3 : L'import peut être en mode "replace" (purge puis insertion) ou "upsert" (mise à jour / création).
RG-J4 : Les unicités fonctionnelles (ex : fingerprint, nom unique…) sont respectées lors de l'import.

---

3) Dictionnaire des données (extrait structuré)

Je te mets la structure par entité (attribut : type, contraintes, rôle).

ROLE

- id : identifiant
- nom_role : texte, unique

USER

- id
- nom : texte

- email : texte, unique, normalisé
- password : texte (hash)
- role_id : FK → ROLE

MENU

- id
- nom : texte
- code_menu : texte, unique

ROLE_MENU_ACCES

- id
- role_id : FK → ROLE
- menu_id : FK → MENU
- est_visible : bool
- contrainte : unique(role_id, menu_id)

CONTACT

- id
- nom : texte
- email : texte, normalisé
- message : texte
- date_envoi : date/heure auto

HISTORIQUE_RETENTION_SETTING (singleton)

- id=1
- retention_years : int
- updated_at : date/heure

HISTORIQUE_USER

- id

- user_id : FK → USER (nullable possible selon design)
- action : enum {CREATE, READ, UPDATE, DELETE}
- app_label : texte
- model_name : texte
- object_id : texte
- object_repr : texte
- date_action : date
- heure_action : heure
- url, method, ip, details

PORTFOLIO (contenus)

- Projetscards (témoignages) : nom, images?, description?, fingerprint unique, created_at, est_visible
- Projetsfirstspeciality : images?, description_speciality?, created_at, est_visible
- Projetmesrealisations : nom?, images?, description?, created_at, est_visible, compteur_demo_live
- Projetphotodeprofil : images?, created_at, est_visible
- ProjetAproposDeMoi : description?, created_at, est_visible
- MesCompetencesCles : nom unique, created_at, est_visible
- ReseauSocial : nom unique, url, visible

---

4) MCD — Modèle Conceptuel des Données (MERISE)

4.1 Entités

- **ROLE**
- **UTILISATEUR**
- **MENU**
- **ACCES_ROLE_MENU**
- **CONTACT**
- **HISTORIQUE_ACTION**
- **PARAMETRE_RETENTION**
- **TEMOIGNAGE**
- **SPECIALITE**

- **REALISATION**
- **PHOTO_PROFIL**
- **A_PROPOS**
- **COMPETENCE**
- **RESEAU_SOCIAL**

4.2 Associations + cardinalités

A1 : “Appartenir” (UTILISATEUR — ROLE)

- Un UTILISATEUR appartient à **1,1** ROLE
- Un ROLE possède **0,n** UTILISATEURS

A2 : “Autoriser” (ROLE — MENU) via ACCES_ROLE_MENU

- Un ROLE est lié à **0,n** MENUS
- Un MENU est lié à **0,n** ROLES
- Association porteuse d’attribut : `est_visible`
- Unicité : (ROLE, MENU) unique

A3 : “Produire” (UTILISATEUR — HISTORIQUE_ACTION)

- Une action d’historique est réalisée par **0,1** UTILISATEUR (peut être null si pas connecté)
- Un UTILISATEUR produit **0,n** actions

A4 : “Paramétrer” (PARAMETRE_RETENTION — HISTORIQUE_ACTION)

- Un PARAMETRE_RETENTION (singleton) “règle” la purge sur **0,n** actions
  (association conceptuelle : seuil appliqué)

Entités portfolio

Elles ne sont pas reliées entre elles (dans ton code), donc ce sont des entités indépendantes :

- TEMOIGNAGE (Projetscards)
- SPECIALITE (Projetsfirstspeciality) — singleton conceptuel

- REALISATION (Projetmesrealisations)
- PHOTO_PROFIL — singleton
- A_PROPOS — singleton
- COMPETENCE
- RESEAU_SOCIAL

---

5) MLD — Modèle Logique des Données (relations)

Notation : TABLE(PK, attributs…, FK→…)

- ROLE(id_role PK, nom_role U)
- UTILISATEUR(id_user PK, nom, email U, password, id_role FK→ROLE)
- MENU(id_menu PK, nom, code_menu U)
- ROLE_MENU_ACCES(id PK, id_role FK→ROLE, id_menu FK→MENU, est_visible, U(id_role,id_menu))
- CONTACT(id_contact PK, nom, email, message, date_envoi)
- HISTO_RETENTION(id PK=1, retention_years, updated_at)
- HISTORIQUE(id_hist PK, id_user FK→UTILISATEUR NULL, action, app_label, model_name, object_id, object_repr, date_action, heure_action, url, method, ip, details)
- TEMOIGNAGE(id PK, nom, images?, description?, fingerprint U, created_at, est_visible)
- SPECIALITE(id PK, images?, description_speciality?, created_at, est_visible)
- REALISATION(id PK, nom?, images?, description?, created_at, est_visible, compteur_demo_live)
- PHOTO_PROFIL(id PK, images?, created_at, est_visible)
- A_PROPOS(id PK, description?, created_at, est_visible)
- COMPETENCE(id PK, nom U, created_at, est_visible)
- RESEAU_SOCIAL(id PK, nom U, url, visible)

---

6) MPD — Modèle Physique (tables, index, contraintes)

Index importants

- HISTORIQUE : index sur date_action, action, (app_label, model_name)

- ROLE_MENU_ACCES : contrainte unique(role_id, menu_id)
- USER : unique(email)
- ROLE : unique(nom_role)
- MENU : unique(code_menu)
- TEMOIGNAGE : unique(fingerprint)
- COMPETENCE : unique(nom)
- RESEAU_SOCIAL : unique(nom)

Particularités physiques

- Champs image : stockés côté MEDIA (chemin) ; import/export gère base64.
- Singleton : SPECIALITE/PHOTO_PROFIL/A_PROPOS/PARAM_RETENTION assurés par règle de gestion + restrictions d'ajout (pas par contrainte SQL stricte sauf param retention pk=1).

---

7) Schéma de circulation des documents (flux MERISE)

Je le décris en "documents/écrans" + acteurs + flux.

Acteurs

- **Visiteur** (non connecté)
- **Utilisateur connecté**
- **Modérateur**
- **Administrateur**
- **Système** (middleware/signals/exports/imports)

Documents / Écrans

- D1 : Page Portfolio publique (`index.html`)
- D2 : Formulaire inscription (`page_inscription.html`)
- D3 : Formulaire connexion
- D4 : Dashboard contenu (`page_administration.html`)
- D5 : Admin centrale utilisateurs/permissions/messages (`page_utilisateur.html`)

- D6 : Fragment historique (`templates/app_historique/fragment_historique.html`)
- D7 : Exports (CSV/Excel/PDF)
- D8 : Bundle JSON export/import

Flux principaux

F1 — Consultation publique

Visiteur → (GET /) → Système → BD (contenus visibles) → Visiteur (page)

- Historique : (si connecté) log READ.

F2 — Inscription

Utilisateur → D2 → POST inscription → BD (User + Role) → Session ouverte → Redirection accueil

- log CREATE user (si tracking) + log READ/UPDATE selon navigation.

F3 — Connexion / Déconnexion

Utilisateur → POST login → BD (check hash) → Session
Déconnexion → flush session → redirection

F4 — Gestion contenu portfolio (dashboard)

Admin/Modo → D4 → actions CRUD sur contenus → BD

- signals : log CREATE/UPDATE/DELETE
- middleware : log READ sur pages HTML

F5 — Gestion utilisateurs & permissions

Admin → D5 → CRUD users → BD
Admin → D5 → update menus (checkbox) → BD (RoleMenuAcces)

- log UPDATE/CREATE/DELETE

F6 — Historique + réglage rétention

Admin → D6 → GET fragment (liste)
Admin → POST settings retention → BD (retention) + purge → BD delete lignes
Admin → export → génération fichier (CSV/Excel/PDF)

F7 — Import/Export JSON

Admin → export JSON → fichier bundle
Admin → import JSON → BD (replace/upsert) + restauration médias

---

8) MCT — Modèle Conceptuel des Traitements

Je te donne une version “événement → opération → résultat”.

E1 : “Visiteur demande la page d’accueil”

- Opération : Charger contenus visibles + menus visibles si session
- Résultat : Page `index.html` rendue

E2 : “Utilisateur s’inscrit”

- Opération : valider données → créer role si besoin → créer user (password hash) → créer session
- Résultat : user créé + redirection accueil

E3 : “Utilisateur se connecte”

- Opération : recherche email → check_password → session
- Résultat : session active ou message d’erreur

E4 : “Admin modifie contenu portfolio”

- Opération : CRUD (témoignages, réalisations, compétences…) + toggles visibilité
- Résultat : contenus mis à jour + logs C/U/D

E5 : “Admin gère utilisateurs”

- Opération : create/edit/delete user + affectation rôle
- Résultat : base users à jour

E6 : “Admin modifie droits”

- Opération : pour chaque (role, menu) mettre est_visible selon checkbox + forcer admin×gestion_users
- Résultat : matrice des accès à jour

E7 : “Consultation page (READ)”

- Opération (middleware) : si conditions OK → créer log READ
- Résultat : ligne historique ajoutée

E8 : “Paramétrage rétention / purge”

- Opération : sauvegarder retention_years → calcul seuil → supprimer anciennes lignes
- Résultat : historique purgé + message retour

E9 : “Export historique”

- Opération : charger lignes → générer CSV/Excel/PDF
- Résultat : fichier téléchargeable

E10 : “Import/Export bundle”

- Opération : sérialiser/désérialiser + base64 médias + upsert/replace
- Résultat : données synchronisées

---

9) Matrice CRUD (résumé)

Entité	Visiteur	Utilisateur	Modérateur	Administrateur
Portfolio (lecture)	R	R	R	R
Témoignages	-	-	CRUD	CRUD
Réalisations	-	-	CRUD	CRUD
Compétences	-	-	CRUD	CRUD
Réseaux sociaux	-	-	CRUD	CRUD
Users	-	-	-	CRUD
Menus/Droits	-	-	-	U
Historique	-	-	-	R + Export + Purge + Param
Contact messages	C (si form branché)	C	R	R

Si tu veux une livraison “document MERISE”

Je peux aussi te le fournir sous forme :

- **Word/PDF** (chapitré, diagrammes ASCII/Mermaid)
- ou **fichier Markdown** prêt à imprimer

Dis-moi juste le format (PDF / DOCX / Markdown) et je te le génère dans la foulée.

Pourriez vous mettre tout le contenu au très grand complet comme le contenu de l'ANALYSE MERISE que vous venez d'afficher ci-dessus de préférable dans un fichier 1) Word 2026 et également dans un fichier 2) PDF (chapitré, diagrammes ASCII/Mermaid) puis dans un 3) fichier Markdown prêt à imprimer
Réflexion durant 7m 39s

Voici les 3 fichiers générés (même contenu, chapitré, avec diagrammes ASCII + blocs Mermaid) :

1. **Word (DOCX – compatible Word 2026)**
   Download Analyse_MERISE_Portfolio.docx
2. **PDF**
   Download Analyse_MERISE_Portfolio.pdf

3. **Markdown prêt à imprimer**
   Download Analyse_MERISE_Portfolio.md