Concepts fondamentaux : Admin auto-généré et authentification

Particularité majeure : Django Admin n'a pas d'équivalent de cette qualité ailleurs.

L'interface d'administration de Django (souvent appelée *Django admin*) est un **outil intégré** destiné à **gérer les données des applications** directement depuis un navigateur web.
Elle repose sur une idée simple : *si un modèle existe, Django peut en déduire une interface CRUD complète sans qu'on ait à écrire de code HTML, de formulaires ou de routes*.

L'admin est **auto-générée à partir des modèles déclarés** dans le code Python, et non depuis la base de données directement.
Ce choix architectural reflète la philosophie Django : **le modèle est la source de vérité**, et tout le reste (migrations, formulaires, admin) s'en déduit.

L'admin Django repose sur un système de registre appelé admin.site.
Chaque modèle doit être enregistré pour apparaître dans l'interface, via le décorateur @admin.register(Model) ou la méthode admin.site.register().

Ce registre relie :

- Un modèle Python (ex. Hero)
- À une classe d'administration (HeroAdmin) héritée de ModelAdmin

Ce mécanisme rend possible :

- La génération automatique des vues CRUD
- La configuration fine des affichages
- L'extension du comportement (actions, filtres, permissions, etc.)

Personnalisation

Champs affichés

On peut définir les champs visibles dans la liste via list_display, et ceux disponibles dans les filtres via list_filter.
L'objectif est de ne pas tout montrer, mais de faciliter la navigation.

Actions personnalisées

On peut ajouter des actions de masse (par ex. désactiver plusieurs héros à la fois).
Ces actions utilisent l'API de ModelAdmin et peuvent être déclenchées depuis la liste des objets.

Point critique :
Une action mal conçue peut avoir des effets irréversibles.
Il faut toujours prévoir une confirmation et vérifier les permissions.

Relations et modularité

Les trois types de relations principales sont :

Type de relation	Déclaration Django	Exemple concret
1 → N (ForeignKey)	models.ForeignKey()	Un héros appartient à une équipe
N → N (ManyToMany)	models.ManyToManyFiel d()	Un héros a plusieurs pouvoirs
1 → 1 (OneToOne)	models.OneToOneField()	Un profil d'utilisateur pour un compte

Pour garder une admin claire et performante, Django propose plusieurs **mécanismes de modularité** :

a) autocomplete_fields

Au lieu d'afficher une grande liste déroulante, Django propose un **champ de recherche asynchrone**.
L'utilisateur tape quelques lettres, Django interroge la base via AJAX, et renvoie les correspondances.
→ C'est idéal pour les ForeignKey ou ManyToMany volumineux.

b) raw_id_fields

Alternative plus basique : remplace le champ de sélection par un simple champ texte contenant l'ID de la relation.
→ Plus performant, mais moins convivial.
→ Souvent utilisé pour les tables très grandes.

c) filter_horizontal et filter_vertical

Widgets visuels permettant de sélectionner plusieurs éléments plus facilement dans les relations ManyToMany.
→ Adapté pour des relations de taille moyenne.

d) Inlines (TabularInline / StackedInline)

Permet d'**éditer des objets liés directement dans la page parent**.
Exemple : éditer les pouvoirs d'un héros sur la même page.
→ Utile pour les relations 1-N.
→ Attention cependant à la surcharge visuelle si les objets enfants sont nombreux.

Modularité applicative :

Chaque application Django peut avoir son propre fichier admin.py.
Cela permet :

- Une organisation claire du code (chaque domaine fonctionnel gère ses modèles).
- Une modularité naturelle : on peut désactiver ou remplacer l'admin d'une app sans affecter les autres.
- De créer plusieurs interfaces d'admin si besoin, en instanciant plusieurs AdminSite

Atouts de l'admin

- **Gain de temps** considérable pour la gestion interne ou les prototypes
- **Intégration native** avec le système d'authentification et de permissions
- **Cohérence** entre le modèle et l'interface (zéro duplication de logique)
- **Extensibilité** : possibilité d'ajouter des vues custom, des templates et des actions complexes

Limites de l'admin

- **Pas pensé pour les utilisateurs finaux** : interface sobre, peu flexible sur le design
- **Complexité croissante** quand les relations et les actions s'empilent
- **Performance** : certaines pages peuvent devenir lentes avec beaucoup d'objets liés
- **Couplage fort** avec le modèle et la logique ORM → difficile à séparer pour un usage "front"
- **Maintenance délicate** si trop de logique métier est intégrée directement dans les classes admin

Authentification et gestion des permissions

L'interface d'administration de Django repose sur le **système d'authentification intégré** du framework, fourni par le module django.contrib.auth(que tu peux retrouver dans INSTALLED_APPS).

Ce système fournit :

- Un modèle **User** représentant les comptes utilisateurs
- Un modèle **Group** permettant de gérer les rôles collectifs
- Un modèle **Permission** pour définir des droits précis sur chaque modèle
- Des outils de gestion : connexion, déconnexion, changement de mot de passe, sessions, etc.
- Une **intégration directe** dans le site d'administration.

L'objectif de ce système est de garantir une **gestion sécurisée des accès**, tout en s'intégrant avec l'ORM et l'admin.

Le modèle User

Le modèle utilisateur de base est défini dans django.contrib.auth.models.User. Il représente un compte dans l'application, qu'il soit administrateur ou simple utilisateur.

Champs principaux :

- username : identifiant unique
- email, first_name, last_name : informations personnelles
- is_staff : indique si l'utilisateur peut accéder à l'interface admin
- is_superuser : accorde tous les droits sans restriction
- is_active : active ou désactive un compte
- groups : relation ManyToMany vers le modèle Group
- user_permissions : relation ManyToMany vers le modèle Permission

Points clés :

- Seuls les utilisateurs ayant is_staff=True peuvent se connecter à /admin
- Les superutilisateurs (is_superuser=True) contournent les vérifications de permissions
- Les mots de passe sont toujours **hachés** avec un algorithme sécurisé (PBKDF2 par défaut)

Le modèle Group

Les groupes sont un mécanisme de **regroupement logique d'utilisateurs**. Chaque groupe peut se voir attribuer un ensemble de permissions, automatiquement héritées par ses membres. Cela évite de gérer les permissions utilisateur par utilisateur.

Le modèle Permission

Les permissions sont des objets représentant une **action autorisée** sur un modèle. Django crée automatiquement trois permissions pour chaque modèle :

- add_<modelname>
- change_<modelname>
- delete_<modelname>

Ces permissions sont enregistrées dans la table auth_permission et accessibles depuis l'admin. On peut ensuite les attribuer à un utilisateur ou un groupe via l'interface admin ou les vérifier dans le code Python avec user.has_perm("app_name.add_modelname")

Intégration avec l'admin

L'accès à l'interface d'administration repose sur ces mêmes modèles.

- Chaque connexion à /admin/ vérifie si l'utilisateur :
    - est **authentifié**
    - et possède **is_staff=True**

- Les actions affichées dans l'admin dépendent des **permissions de l'utilisateur connecté**. Par exemple, un utilisateur sans la permission "delete" ne verra pas le bouton "Supprimer".

Rôle	**Accès admin**	**Permissions principales**
Superuser	Oui	Tous les droits
Éditeur	Oui	add et change sur certains modèles
Utilisateur standard	Non	Aucune (is_staff false)

Gestion personnalisée

Django permet aussi :

- de **remplacer le modèle User par un modèle personnalisé** (via AUTH_USER_MODEL) pour gérer des champs supplémentaires comme une photo, un rôle spécifique, etc.
- d'utiliser l'authentification dans d'autres contextes : formulaires, API, permissions sur vues, etc.

Atouts :

- Granularité fine via les permissions
- Compatible avec l'admin, les vues et le middleware
- Intégration automatique dans l'admin

Limites :

- Personnalisation complexe et modèle User souvent insuffisant pour ses besoins (à customiser)
- Permission peu flexible pour les gros projet

En résumé :

- L'admin Django **ne gère que les utilisateurs ayant is_staff=True**
- L'authentification et les permissions sont des **modèles Django standards**, donc extensibles
- La séparation User / Group / Permission reflète une **logique de contrôle d'accès à plusieurs niveaux**