

CADEAU OFFERT

Aide-mémoire Django 6

La fiche de référence à garder sous la main

Tout ce qu'un développeur Django utilise au quotidien, réuni en quelques pages : modèles, vues, gabarits, formulaires, ORM, routes, fichiers statiques, relations, `on_delete`, et le partage des rôles entre service, helper, vue et modèle.

Ce que vous y trouverez

- Les commandes essentielles, du démarrage au déploiement
- Les champs de modèles et leurs options les plus utilisées
- Les trois types de relations, et comment choisir
- Les comportements de `on_delete`, expliqués simplement
- Les requêtes ORM courantes, prêtes à réutiliser
- Service, helper, vue ou modèle : qui fait quoi

Extrait du livre « Django 6, de débutant à professionnel »

398 pages pour construire une application complète, jusqu'à la mise en ligne.

Aide-mémoire Django 6

La fiche de référence de tout ce que le livre utilise, à garder sous la main

Ce mémo rassemble, en une fiche, l'essentiel de Django 6 utilisé tout au long du BugTracker. Il ne remplace pas les chapitres : il te sert de rappel rapide quand tu cherches une syntaxe ou un réflexe. Les exemples reprennent les conventions du livre : projet nommé **bugtracker**, utilisateur **CustomUser**, cloisonnement par entreprise.

1. Structure du projet

```
bugtracker/          # dossier racine
├── manage.py        # le couteau suisse des commandes
├── bugtracker/      # package de configuration
│   ├── settings.py  # configuration
│   ├── urls.py      # routes du projet
│   └── wsgi.py       # point d'entrée serveur
├── accounts/        # utilisateur, entreprise, rôles
├── projects/        # projets
├── tickets/         # tickets, commentaires, pièces jointes
├── templates/       # gabarits HTML
├── static/ assets/  # CSS compilé, JS, images
└── .env             # secrets, jamais versionné
```

2. Les commandes essentielles

Commande	Rôle
<code>django-admin startproject bugtracker .</code>	Créer le projet
<code>python manage.py startapp <nom></code>	Créer une application
<code>python manage.py makemigrations</code>	Générer les migrations
<code>python manage.py migrate</code>	Appliquer les migrations
<code>python manage.py runserver</code>	Lancer le serveur de développement
<code>python manage.py tailwind runserver</code>	Serveur + surveillance Tailwind
<code>python manage.py createsuperuser</code>	Créer un compte administrateur
<code>python manage.py shell</code>	Ouvrir la console Python
<code>python manage.py test</code>	Lancer les tests
<code>python manage.py collectstatic</code>	Rassembler les fichiers statiques

3. Les modèles

Un modèle décrit une table. On hérite d'un **BaseModel** commun pour l'**UUID** et les **dates**. Voir aussi l'aide-mémoire dédié aux champs et relations à la fin de cette partie.

```
import uuid
from django.db import models

class BaseModel(models.Model):
    id = models.UUIDField(primary_key=True,
        default=uuid.uuid4, editable=False)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    class Meta:
        abstract = True

class Project(BaseModel):
    company = models.ForeignKey('accounts.Company',
        on_delete=models.CASCADE, related_name='projects')
    name = models.CharField(max_length=150)
    def __str__(self):
        return self.name
```

Astuce

Réflexes modèles : toujours une méthode `__str__` pour un affichage lisible ; **`on_delete`** traduit une règle métier (**`CASCADE`**, **`SET_NULL`**, **`PROTECT`**) ; les valeurs fixes se déclarent en **`TextChoices`** ; et on archive au lieu de supprimer, avec un champ booléen **`archived`**.

4. Les vues basées sur les classes

Vue	Usage
ListView	Afficher une liste d'objets
DetailView	Afficher le détail d'un objet
CreateView	Créer un objet via un formulaire
UpdateView	Modifier un objet existant
TemplateView	Afficher une page avec un contexte
View	Vue générique : on écrit <code>get</code> et <code>post</code> à la main

Les mixins de protection :

- **`LoginRequiredMixin`** : exige la connexion.
- **`UserPassesTestMixin`** : autorise selon **`test_func`** (le rôle, une permission).

```
class ProjectListView(LoginRequiredMixin, ListView):
    model = Project
```

```

template_name = 'projects/project_list.html'
context_object_name = 'projects'
paginate_by = 10

def get_queryset(self):          # filtrer les données
    return Project.objects.for_company(
        self.request.user.company_id).active()

def form_valid(self, form):      # forcer une donnée au serveur
    form.instance.company = self.request.user.company
    return super().form_valid(form)

def get_context_data(self, **kwargs): # ajouter au contexte
    context = super().get_context_data(**kwargs)
    context['x'] = ...
    return context

def test_func(self):             # avec UserPassesTestMixin
    return self.request.user.role == Roles.ADMIN

```

5. Les URLs

```

# projects/urls.py
from django.urls import path
from . import views

app_name = 'projects'          # espace de noms

urlpatterns = [
    path('', views.ProjectListView.as_view(), name='list'),
    path('<uuid:pk>/', views.ProjectDetailView.as_view(), name='detail'),
]

# bugtracker/urls.py
urlpatterns = [
    path('admin/', admin.site.urls),
    path('projects/', include('projects.urls')),
]

```

Note

On référence une route avec `{% url 'projects:detail' project.pk %}` dans un gabarit, ou `reverse('projects:detail', args=[pk])` en Python. L'application **accounts** garde des noms simples (**login**, **logout**, **company**), car le système d'authentification l'attend ; les autres apps ont un **app_name**.

6. Les gabarits (*templates*)

Balise	Rôle
{% extends 'base.html' %}	Hériter d'un gabarit de base
{% block content %}...{% endblock %}	Zone remplaçable
{% include 'partials/_x.html' %}	Inclure un fragment
{% for x in xs %} ... {% empty %} ...	Boucle avec cas vide
{% if cond %} ... {% endif %}	Condition
{% url 'app:name' arg %}	Lien vers une route
{% csrf_token %}	Jeton anti-falsification (formulaires POST)
{{ valeur date:"d/m/Y" }}	Filtre de formatage
{{ valeur default:"—" }}	Valeur de repli si vide
{{ liste length }}	Nombre d'éléments

7. Les formulaires

```
# forms.py – style habituel avec le dictionnaire widgets
class ProjectForm(forms.ModelForm):
    class Meta:
        model = Project
        fields = ['name', 'description']
        widgets = {
            'name': forms.TextInput(attrs={'class': 'form-input'}),
            'description': forms.Textarea(
                attrs={'class': 'form-textarea', 'rows': 4}),
        }
```

Astuce

Pré-remplir : `TicketForm(initial={'title': ...})`. Limiter un choix : `form.fields['project'].queryset = ....`
Afficher un champ stylé dans le gabarit : boucler sur `{% for field in form %}` et afficher `{{ field }}` ; les classes viennent des **widgets** ou du **mixin** de style.

8. Authentification, rôles, permissions

```
# Toujours référencer l'utilisateur ainsi :
from django.contrib.auth import get_user_model
User = get_user_model()           # dans le code
settings.AUTH_USER_MODEL         # dans un ForeignKey

# Rôle : logique avec role, affichage avec role_label
if user.role == Roles.ADMIN: ...
{{ member.role_label }}
```

```
# Connexion / déconnexion : vues intégrées de Django
# Groupes = rôles : user.groups.add(Group.objects.get(name=...))
```

9. L'ORM, requêtes courantes

Besoin	Code
Filtrer	<code>.filter(champ=valeur)</code>
Exclure	<code>.exclude(champ=valeur)</code>
OU logique	<code>from django.db.models import Q ; .filter(Q(a=1) Q(b=2))</code>
Traverser une relation	<code>.filter(project__company=company)</code>
Un objet ou erreur 404	<code>get_object_or_404(Model, pk=pk, company=...)</code>
Compter	<code>.count()</code> ou <code>Count('id')</code> dans <code>annotate</code>
Regrouper et compter	<code>.values('status').annotate(total=Count('id'))</code>
Premier / existe	<code>.first()</code> / <code>.exists()</code>
Trier	<code>.order_by('-created')</code> ou <code>Meta.ordering</code>

⚠ Attention

Réflexe de sécurité numéro un : toute requête qui renvoie des données doit d'abord être filtrée par l'entreprise de l'utilisateur, avec `.filter(company=...)` ou `.filter(project__company=...)`. C'est ce qui garantit qu'une entreprise ne voit jamais les données d'une autre.

10. Réglages et environnement

```
# bugtracker/settings.py
import environ
env = environ.Env(DEBUG=(bool, False))
environ.Env.read_env(BASE_DIR / '.env')

SECRET_KEY = env('SECRET_KEY', default='django-insecure-')
DEBUG = env('DEBUG')
ALLOWED_HOSTS = env.list('ALLOWED_HOSTS',
    default=['127.0.0.1', 'localhost'])
DATABASES = {'default': env.db()} # lit DATABASE_URL
AUTH_USER_MODEL = 'accounts.CustomUser'
```

📖 Note

Règle d'or : le code va sur GitHub, les secrets vont dans le `.env`, jamais l'inverse. En production, `DEBUG` vaut `False` et `ALLOWED_HOSTS` contient le domaine.

11. Fichiers statiques et média

```
STATICFILES_DIRS = [BASE_DIR / 'assets'] # source (Tailwind)
STATIC_ROOT = BASE_DIR / 'staticfiles' # production (collectstatic)
MEDIA_URL = 'media/'
MEDIA_ROOT = BASE_DIR / 'media' # fichiers uploadés
```

```
# En dev, servir les médias (bugtracker/urls.py) :
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
        document_root=settings.MEDIA_ROOT)
```

12. Messages et session

```
from django.contrib import messages
messages.success(request, 'Enregistré.') # message flash
messages.error(request, 'Erreur.')

request.session['clef'] = valeur          # stocker
request.session.pop('clef', '')           # lire et retirer
```

13. L'API REST (DRF), en bref

```
# serializer
class TicketSerializer(serializers.ModelSerializer):
    class Meta:
        model = Ticket
        fields = [...]
        read_only_fields = ['submitter', 'status']

# viewset (filtré par entreprise + données forcées)
class TicketViewSet(viewsets.ModelViewSet):
    serializer_class = TicketSerializer
    permission_classes = [permissions.IsAuthenticated]
    def get_queryset(self):
        return Ticket.objects.filter(
            project__company=self.request.user.company)

# routeur + JWT dans les URLs
router.register('tickets', TicketViewSet, basename='ticket')
```

14. Les tests

```
from django.test import TestCase

class MonTest(TestCase):
    def setUp(self):
        # créer les données de test
        ...
    def test_isolation(self):
        self.client.force_login(self.utilisateur)
        r = self.client.get(url)
        self.assertEqual(r.status_code, 404)
```

Note

Django crée une base de test séparée (droit CREATEDB requis sur l'utilisateur). Chaque test est isolé dans une transaction annulée. `self.client` simule un navigateur, `self.client.force_login` connecte un utilisateur.

15. Déploiement, l'essentiel

La checklist de mise en production :

- `DEBUG = False` et `ALLOWED_HOSTS` renseigné.
- Secrets dans les variables d'environnement de l'hébergeur.
- Fichiers statiques servis par WhiteNoise, après collectstatic.
- Médias sur un stockage objet (le disque du conteneur est éphémère).
- Serveur d'application Unicorn, pas runserver.
- Migrations appliquées au démarrage.

16. Les réflexes de sécurité du livre

- Filtrer toute requête par l'entreprise de l'utilisateur.
- Forcer les données sensibles au serveur (entreprise, rapporteur), jamais depuis le formulaire ou l'IA.
- Protéger les vues d'écriture au serveur (mixins, permissions), pas seulement en cachant les boutons.
- Ne jamais mettre de secret dans le code ; tout dans le `.env`.
- Avec l'IA : elle propose, ton code exécute avec tes règles.

Garde cette fiche ouverte pendant que tu codes. Au fil du projet, ces réflexes deviendront naturels, et tu n'auras plus besoin d'y revenir. Le prochain bloc du bonus complétera les notifications de l'application, partout où elles ont du sens.

Les champs et les relations des modèles

Ce document est une fiche de référence à garder sous la main. Il rassemble les champs de modèle les plus utilisés en Django, les trois types de relations, les options les plus courantes, et surtout la méthode pour choisir entre elles. Les exemples s'appuient sur les modèles du BugTracker, pour te parler.

Les champs de données les plus utilisés

Un champ décrit une donnée simple : du texte, un nombre, une date, un fichier. Voici ceux que tu rencontreras le plus souvent.

Champ	À quoi il sert	Exemple
CharField	Texte court, avec une longueur maximale obligatoire.	<code>name = models.CharField(max_length=120)</code>
TextField	Texte long, sans limite pratique.	<code>description = models.TextField(blank=True)</code>
BooleanField	Une valeur vraie ou fausse.	<code>archived = models.BooleanField(default=False)</code>
IntegerField	Un nombre entier.	<code>order = models.IntegerField(default=0)</code>
DecimalField	Un nombre décimal précis, idéal pour l'argent.	<code>price = models.DecimalField(max_digits=8, decimal_places=2)</code>
DateTimeField	Une date avec l'heure.	<code>created = models.DateTimeField(auto_now_add=True)</code>
DateField	Une date seule, sans heure.	<code>due_date = models.DateField(null=True, blank=True)</code>
EmailField	Un email, avec validation du format.	<code>email = models.EmailField()</code>
SlugField	Un texte propre pour les URLs.	<code>slug = models.SlugField()</code>
UUIDField	Un identifiant unique universel.	<code>id = models.UUIDField(primary_key=True, default=uuid.uuid4)</code>
FileField	Un fichier envoyé par l'utilisateur.	<code>file = models.FileField(upload_to='attachments/')</code>
ImageField	Une image (nécessite Pillow).	<code>logo = models.ImageField(upload_to='companies/')</code>

Comment fonctionne Django

Pour un champ limité à une liste de valeurs fixes, comme le statut d'un ticket, on n'utilise pas un champ spécial : on prend un **CharField** auquel on ajoute l'option **choices**. Par exemple, **status = models.CharField(max_length=20, choices=TicketStatus.choices)**. Le champ reste un texte, mais Django n'accepte que les valeurs prévues et affiche un menu déroulant.

Les trois relations entre modèles

Une relation relie un modèle à un autre. Il n'en existe que trois, et c'est le nombre d'objets de chaque côté qui les distingue.

Relation	Quand l'utiliser	Exemple
ForeignKey	Un-à-plusieurs. Un objet appartient à un seul autre, qui peut en avoir plusieurs.	<code>project = models.ForeignKey(Project, on_delete=models.CASCADE, related_name='tickets')</code>
ManyToManyField	Plusieurs-à-plusieurs. Chaque côté peut en avoir plusieurs.	<code>members = models.ManyToManyField(User, related_name='projects')</code>
OneToOneField	Un-à-un. Un objet correspond à exactement un autre.	<code>profile = models.OneToOneField(User, on_delete=models.CASCADE)</code>

Dans le BugTracker :

- Un ticket a un seul projet, un projet a plusieurs tickets : c'est une ForeignKey sur le ticket.
- Un projet a plusieurs membres, un membre travaille sur plusieurs projets : c'est un ManyToManyField.
- Si chaque utilisateur avait un unique profil détaillé, ce serait un OneToOneField.

Comment choisir entre ForeignKey, ManyToMany et OneToOne

La bonne méthode tient en une question, posée dans les deux sens. Prends tes deux modèles, appelons-les A et B, et demande-toi combien d'objets de chaque côté.

Combien de A pour un B ? / Combien de B pour un A ?	Relation	Exemple BugTracker
Plusieurs A pour un B, un seul B pour un A	ForeignKey	Plusieurs tickets pour un projet
Plusieurs des deux côtés	ManyToManyField	Membres et projets
Un seul de chaque côté	OneToOneField	Un profil par utilisateur

Astuce

Le côté où poser la **ForeignKey** est toujours le côté plusieurs. Le ticket porte la clé vers le projet, parce qu'il y a plusieurs tickets pour un projet. On dit que le ticket appartient au projet. Si tu hésites, formule la phrase à voix haute : celui qui appartient à l'autre porte la **ForeignKey**.

Ce qu'un recruteur regarde

Savoir justifier ton choix de relation est un signe de maîtrise. En entretien, on pourra te demander pourquoi tu as utilisé un **ManyToMany** ici plutôt qu'une **ForeignKey**. Répondre en raisonnant sur le nombre d'objets de chaque côté, plutôt que par habitude, montre que tu comprends ta modélisation.

Les options les plus courantes

Les options affinent le comportement d'un champ. Voici celles que tu utiliseras le plus.

Option	Ce qu'elle fait	Exemple
<code>primary_key=True</code>	Fait de ce champ la clé primaire du modèle.	<code>id = models.UUIDField(primary_key=True)</code>
<code>default</code>	Valeur prise si rien n'est fourni.	<code>archived = ...(default=False)</code>
<code>null=True</code>	Autorise l'absence de valeur en base de données.	<code>due_date = ...(null=True)</code>
<code>blank=True</code>	Autorise le champ vide dans les formulaires.	<code>description = ...(blank=True)</code>
<code>choices</code>	Limite les valeurs possibles à une liste.	<code>choices=TicketStatus.choices</code>
<code>unique=True</code>	Interdit deux fois la même valeur dans la table.	<code>email = ...(unique=True)</code>
<code>on_delete</code>	Que faire quand l'objet lié est supprimé.	<code>on_delete=models.CASCADE</code>
<code>related_name</code>	Nom pour remonter la relation en sens inverse.	<code>related_name='tickets'</code>
<code>auto_now_add</code>	Fixe la date à la création, une seule fois.	<code>created = ...(auto_now_add=True)</code>
<code>auto_now</code>	Met la date à jour à chaque enregistrement.	<code>updated = ...(auto_now=True)</code>
<code>upload_to</code>	Sous-dossier de destination d'un fichier.	<code>upload_to='avatars/'</code>

Le piège classique : *null* contre *blank*

Ces deux options se ressemblent mais ne parlent pas de la même chose, et on les confond souvent. **null** concerne la base de données : le champ peut être vide en base. **blank** concerne les formulaires : le champ peut être laissé vide à la saisie.

La règle pratique :

- Pour un champ texte, utilise seulement **blank=True**, sans **null=True**. Un texte vide se représente par une chaîne vide, pas par un vide en base. Mettre les deux créerait deux façons différentes d'être vide, source de confusion.
- Pour les autres **champs**, **dates**, **nombres**, **ForeignKey**, utilise **null=True** et **blank=True** ensemble si le champ est facultatif, car il n'existe pas de chaîne vide pour eux.

Attention

C'est l'une des confusions les plus fréquentes chez les débutants. Retiens la formule : **blank** pilote la **saisie**, **null** pilote le **stockage**. Pour du texte facultatif, `blank=True` suffit. Pour une date ou une relation facultative, il faut `null=True` en plus.

Les comportements de *on_delete*

Quand tu poses une ForeignKey, Django exige que tu précises ce qui se passe si l'objet pointé est supprimé. Voici les choix les plus utiles.

Valeur	Effet quand l'objet lié est supprimé	Exemple d'usage
CASCADE	Supprime aussi les objets qui pointent vers lui.	Un projet supprimé emporte ses tickets.
SET_NULL	Met le lien à vide (exige <i>null=True</i>).	Un chef de projet parti, le projet reste sans chef.
PROTECT	Empêche la suppression tant qu'il reste des liens.	Interdire d'effacer une donnée encore utilisée.
SET_DEFAULT	Remet la valeur par défaut du champ.	Rare, si une valeur de repli existe.

Comment fonctionne Django

Il n'y a pas de bon choix universel : cela dépend du sens métier. Se demander a-t-il encore un sens de garder cet objet si son parent disparaît guide la décision. Un ticket sans projet n'a pas de sens, donc *CASCADE*. Un projet sans chef de projet garde tout son sens, donc *SET_NULL*. Choisir *on_delete*, c'est décrire une règle métier.

La différence entre un "service" et "helper"

La distinction est subtile car les deux vivent souvent dans un fichier *services.py*. La différence tient à leur **responsabilité**, pas à leur emplacement.

Un **service** porte une règle métier. C'est une action qui a du sens dans le domaine de ton application, avec ses conditions, ses validations et parfois plusieurs étapes enchaînées. Il prend des décisions : qui a le droit, dans quel ordre, sous quelles conditions. Dans ton BugTracker, *assign_project_manager*, *change_user_role*, *assign_developer* sont des services : ils vérifient les rôles avec des *guard clauses*, respectent l'entreprise, et modifient les données selon les règles. Un service répond à la question « **que fait l'application, métier compris** »

Un **helper** est un petit outil technique, générique et réutilisable, qui ne prend aucune décision métier. Il répond à la question « **comment accomplir une petite tâche technique** ». Il rend un service au sens courant du mot, mais il est bête et neutre : il fait une tâche précise, toujours de la même façon, sans savoir dans quel contexte on l'appelle. *notify(recipient, message, url)*, *notify_many*, ou *_cosine(a, b)* sont des helpers : ils créent une notification, calculent une similarité, sans se soucier de savoir si c'est permis ou pertinent. Ces deux helpers de notification ne sont pas inventés pour l'exemple : tu les écriras au bonus B2, qui complète les notifications de l'application. Garde-les en tête en lisant ce qui suit, tu les reconnaîtras.

Note

Un **dernier repère utile** : le nom. Un **service** porte souvent un verbe métier, *assign*, *create*, *archive*, *change_role*. Un **helper** porte un verbe technique et générique, *notify*, *format*, *slugify*, *compute*. Et la frontière reste parfois floue : certaines équipes rangent les services dans *services.py* et les helpers dans

utils.py. L'important n'est pas l'étiquette, mais de savoir si le code porte une règle métier ou une simple mécanique.

Service, helper, vue ou modèle : qui fait quoi

Un doute fréquent : où ranger un bout de code ? Voici les quatre couches et leur responsabilité.

Couche	Ce qu'elle fait — exemple du livre
Modèle	Décrit les données et leur comportement propre. Ex : <code>project.archive()</code>
Service	Porte les règles métier : vérifie les permissions, valide, orchestre plusieurs étapes. Ex : <code>assign_developer</code> , <code>change_user_role</code>
Helper	Tâche technique isolée, réutilisable, sans décision métier. Ex : <code>notify</code> , <code>notify_many</code>
Vue	Reçoit la requête HTTP, appelle le service, renvoie la réponse. Reste mince. Ex : <code>ProjectCreateView</code>

Astuce

Comment trancher en cas de doute : y a-t-il une règle métier, une permission, plusieurs étapes, ou un partage entre le site et l'API ? **C'est un service**. Est-ce une petite tâche technique répétée, sans décision ? **C'est un helper**. Le nom du fichier importe peu, `services.py` ou `utils.py` ; ce qui compte, c'est la responsabilité du code.

Pourquoi la documentation est votre meilleur professeur

Beaucoup de développeurs débutants pensent que la meilleure façon d'apprendre consiste à regarder des tutoriels vidéo, suivre des formations en ligne ou copier des solutions trouvées sur Internet. Ces ressources sont utiles, mais elles ont une limite importante : elles répondent généralement à un problème précis sans toujours expliquer le fonctionnement réel de la technologie.

À l'inverse, la documentation officielle est la source de référence sur laquelle s'appuient les créateurs d'un framework ou d'une bibliothèque. Elle contient les informations les plus récentes, les comportements exacts des fonctionnalités et les explications qui permettent de comprendre les choix de conception du projet.

Apprendre à lire une documentation n'est donc pas une compétence secondaire. C'est l'une des qualités qui distinguent un développeur capable de résoudre des problèmes complexes, d'apprendre rapidement de nouvelles technologies et de rester autonome tout au long de sa carrière.

Dans cette section, nous allons découvrir pourquoi la documentation officielle devrait devenir votre premier réflexe lorsque vous apprenez Django, *Django REST Framework* ou toute autre technologie. Vous verrez qu'au-delà de la simple recherche d'informations, elle constitue un véritable accélérateur de progression et un atout majeur pour votre évolution professionnelle.

L'autonomie, d'abord. C'est le bénéfice qui change tout. Tant qu'on dépend d'un tutoriel, d'un forum ou d'une IA, on ne peut construire que ce que quelqu'un a déjà expliqué. Dès qu'on sait lire une documentation, on peut aborder n'importe quelle bibliothèque, y compris obscure ou toute récente, sans attendre que quelqu'un écrive l'article. C'est précisément la différence entre un développeur qu'il faut accompagner et un développeur à qui l'on confie un sujet.

L'exactitude et la fraîcheur. Une réponse trouvée sur un forum date de trois ans et concerne peut-être une version incompatible. La documentation officielle décrit la version que tu utilises réellement. L'erreur de **collectstatic** en est un bon exemple : le comportement du stockage manifeste de **WhiteNoise** et la nature du fichier source **Tailwind** sont documentés, mais aucun tutoriel généraliste ne t'aurait prévenu de leur interaction dans ce cas précis.

La découverte de ce que tu ne cherchais pas. En lisant la documentation d'une classe, tu croises des options dont tu ignorais l'existence, et qui t'évitent parfois d'écrire cinquante lignes. Beaucoup de code inutile s'écrit simplement parce que son auteur ignorait que l'outil savait déjà le faire. Django est particulièrement concerné : il regorge de fonctionnalités qui répondent à des besoins que les gens réimplémentent à la main.

La compréhension du modèle mental. Une bonne documentation ne se contente pas de lister des fonctions, elle explique la logique de conception. Comprendre pourquoi Django sépare la vue du gabarit, ou pourquoi DRF distingue authentification et permission, te permet de deviner le comportement dans des cas que tu n'as jamais rencontrés. Tu passes de la mémorisation à la déduction.

Le débogage. Un message d'erreur devient exploitable quand on sait où chercher ce qu'il désigne. **MissingFileError** ne veut rien dire tant qu'on n'a pas compris ce que fait le stockage manifeste, et cette information est dans la documentation, pas ailleurs.

La crédibilité professionnelle. En entretien comme en équipe, la question « comment tu fais quand tu ne sais pas ? » a une bonne réponse : la documentation d'abord, les forums ensuite. Un développeur qui cite la documentation officielle plutôt qu'un article de blog inspire immédiatement confiance. C'est d'ailleurs un critère implicite de recrutement.

La durabilité de la compétence. Les frameworks changent, les modes passent, mais la capacité à assimiler une documentation se transfère intégralement. C'est un investissement qui ne se déprécie jamais, contrairement à la connaissance par cœur d'une API donnée.

Un dernier point, souvent négligé. Savoir lire une documentation apprend aussi à en écrire une bonne. Tu repères ce qui t'a aidé et ce qui t'a fait perdre du temps, et tes propres **README**, tes **docstrings** et ta documentation d'API en profitent. C'est directement visible dans un portfolio.

Un conseil pratique pour progresser : quand tu utilises une nouvelle fonctionnalité, prends l'habitude de lire sa page de référence en entier, pas seulement l'exemple qui règle ton problème immédiat. Cinq minutes de plus, et tu en ressors avec trois choses que tu ne cherchais pas.

Et quand la documentation ne suffit pas, il reste une marche à franchir : lire les sources. Django est écrit en Python, dans ton propre environnement virtuel, le code de **ListView**, de **ModelForm** ou de **get_object_or_404** est à quelques clics dans VS Code. Il suffit de maintenir **Ctrl** et de cliquer sur un nom de classe pour y arriver. C'est intimidant la première fois, et beaucoup moins la deuxième : le code du framework ressemble à celui qu'on écrit soi-même, avec des noms de variables ordinaires et des commentaires. C'est souvent la façon la plus rapide de comprendre l'ordre exact dans lequel une vue générique appelle ses méthodes, ou ce qu'un argument fait vraiment. Les développeurs expérimentés y vont plus souvent qu'on ne l'imagine, non pas parce qu'ils sont plus forts, mais parce qu'ils ont cessé de croire que le framework est une boîte noire.

En résumé

Pour un champ de donnée, choisis selon le type d'information : texte, nombre, date, fichier. Pour une relation, compte les objets de chaque côté : un-à-plusieurs donne une **ForeignKey**, plusieurs-à-plusieurs

un **ManyToManyField**, un-à-un un **OneToOneField**. Pour les options, retiens surtout la distinction **blank** contre **null**, et le fait que **on_delete** traduit une règle métier. Garde cette fiche près de toi : au fil du projet, ces réflexes deviendront naturels.



CET AIDE-MÉMOIRE EST UN EXTRAIT

Le livre complet

Vous venez de lire l'un des sept bonus du livre. Le livre lui-même compte 398 pages et vous mène jusqu'à la mise en ligne de votre application.

Ce que vous allez construire

- Un traqueur de bugs multi-entreprises, complet
- Sécurité par rôles, vérifiée au serveur
- Une API REST avec DRF, JWT et documentation
- Tests automatisés, CI et déploiement Docker
- 7 bonus, dont la préparation à l'entretien
- La Partie 9 sur l'IA, offerte par email

Essayez l'application avant d'acheter

<https://bugtrackerapp.up.railway.app/> · Comptes de démonstration en un clic

Obtenir le livre complet

Téléchargement immédiat en PDF
Mises à jour gratuites

<https://tekprototype.gumroad.com/l/ffvwe>

10 % de réduction avec le code : CODELANCEMENT10

à saisir au moment du paiement

Garantie satisfait ou remboursé de 7 jours

Le livre ne vous convient pas ? Écrivez-moi dans les 7 jours, je vous rembourse intégralement.