DJANGO - COURS

Coffre Maureen, Ziane-Chaouche Louiza,
Logier Elsa

Table des matières

Partie 1 : Histoire et évolution de Django

1. Un peu d'histoire…

1.1. Origine et contexte:

Développé en 2003 pour le journal local Lawrence, Django à pour **but** de rendre le développement d'applications web simple et basé sur la réutilisation de code.
Son nom est inspiré du guitariste Django Reinhardt.

Le **but** de Django à l'origine est de répondre aux contraintes des journalistes : l'administration générée par le framework permet ainsi un développement aisé de fonctionnalités axé « contenu ». Le framework se voulait également accessible, afin d'éviter le recrutement d'experts en développement pour les journaux[1]
Au début des années 2000, les sites d'actualité (presse, médias) devaient publier rapidement beaucoup de contenu dynamique.
Les frameworks web étaient rares, souvent limités à du routage basique.

Source :

Coding Web Apps Back in the Early 2000s

Back then it was pretty much ASP Classic or PHP. There were other options (ColdFusion anyone?!) but those were the 2 heavy hitters if you wanted to do server side programming.

I started with ASP Classic and moved onto PHP, which I've written about in the past.

Back then there weren't a ton of frameworks to use. The concept of package managers didn't even exist, or if they did, I had no idea what one was at the time.

I was a PHP cowboy. Over the years I wrote over 300,000 lines of PHP and all of my applications required nothing more than the PHP 4.x standard library. I am literally laughing out loud to myself as I write this sentence because that sounds insane, but it's the truth.

I have these monstrous 7,500+ line PHP files that have a mixture of PHP, HTML, SQL and JS. Each page of the site had its own PHP file because there was no concept of a router. It was actually really simple to reason about.

→ au début des années 2000, la plupart des sites étaient faits "à la main", sans framework complet : juste du PHP ou ASP avec du routage basique.

En juillet 2005 Django à été publié sous licence BSD (permet de réutiliser tout ou une partie du logiciel sans restriction, une licence BSD s'approche de la notion de domaine public avec certaines contraintes)

1.2. Analyse critique : ADN de Django

- Django est historiquement un **framework "éditorial"** : pensé pour la structuration de données relationnelles et la gestion de contenu (pas pour le temps réel ou le microservice).
- Cela explique son ADN : **ORM robuste, admin auto-généré, sécurité intégrée, cohérence de l'écosystème**

1.3. Évolution et maturité du framework

Évolution

Version	Date de sortie	Dernier patch	Remarques
1.0	3 sept. 2008	9 oct. 2009	Début de la maturité du framework Première version stable
2.0	2 déc. 2017	12 fév. 2019	Passage au support Python 3
3.0	2 déc. 2019	6 avr. 2021	Introduction d'ASGI
4.0	7 déc. 2021	14 fév. 2023	Typage renforcé
5.0	4 déc. 2023	2 avr. 2025	API moderne, sécurité accrue
6.0	Décembre 2025	(en dev.)	CSP policies renforcées et monitorées Background tasks email api moderne

Quelques définitions :

CSP policies renforcées et monitorées

« CSP » veut dire Content Security Policy.
C'est une mesure de sécurité intégrée au navigateur pour éviter que des scripts ou du contenu malveillant s'exécutent sur votre site

Exemple :

- une attaque XSS (injection de code JavaScript).
- Si votre site charge une image depuis un autre domaine, Django vérifiera automatiquement si c'est autorisé par la politique de sécurité.

Background tasks

C’est une nouvelle gestion native des tâches en arrière-plan.
Actuellement, quand on veut exécuter une tâche longue (comme envoyer 100 e-mails ou recalculer des statistiques), il faut passer par un outil externe.

Avec Django 6.0, il sera possible de déclarer directement dans le framework des tâches asynchrones qui tournent en arrière-plan, sans bloquer la réponse à l’utilisateur.

Email API moderne

L’envoi d’e-mails dans Django existe depuis longtemps, mais l’API était un peu ancienne et rigide.
La nouvelle Email API moderne apportera :

- une syntaxe plus simple et plus cohérente ;
- un meilleur support pour les emails HTML, les pièces jointes

2. Notoriété et concurrents

2.1. Concurrents principaux :

- Flask
- Tornado
- Web2py
- CherryPy

Positionnement de Django face à ses concurrents

Django se positionne comme un **framework complet et structuré**, ce qu’on appelle *“batteries incluses”*.

Contrairement à Flask ou Tornado, qui sont plus légers et demandent d’ajouter beaucoup d’extensions, Django fournit tout dès le départ : ORM, authentification, admin, sécurité, gestion des templates, etc.

Flask, lui, est parfait pour les petits projets ou les microservices car il est minimaliste et très flexible.

Tornado est plus orienté applications temps réel ou asynchrones (par exemple du chat ou du streaming).

Et **Web2py ou CherryPy** sont des frameworks plus anciens : ils ont inspiré certains concepts, mais leur communauté est aujourd’hui beaucoup moins active.

Utilisation en entreprise

Entreprise / Organisation	Ce qu'elles utilisent Django / pour quel(s) usage(s)	Raisons souvent citées pour avoir choisi Django / ce que ça montre / limites probables
Instagram	Django est la base de leur backend, historiquement très fortement utilisé.	Ils ont besoin de traiter énormément de contenu (photos, vidéos, interactions). Ils ont choisi Django pour gérer leurs données et leur charge d'interaction avec les utilisateurs. La simplicité de Django, combinée à la lisibilité de Python, permet à Instagram de gérer efficacement des milliards de publications et d'interactions générées par les utilisateurs, tout en gardant une architecture cohérente.
Spotify	Django n'est pas nécessairement le framework "core" pour tous les services, mais il est utilisé pour plusieurs backend services, notamment des tâches de gestion, de prototypage, analyses de données.	La compatibilité de Django avec les puissantes bibliothèques de traitement de données de Python permet à Spotify d'offrir un service de streaming musical transparent. Moins utilisé pour les services ultra latents ou critiques, probablement parce que certains de leurs besoins nécessitent des performances ou une architecture plus spécifique.
Pinterest	Certaines parties de Pinterest utilisent Django pour des fonctionnalités de backend, gestion des contenus / "boards", etc.	Ce qui est illustratif : Django peut soutenir des plateformes visuelles riches, avec beaucoup de données (images, métadonnées)
National Geographic	Utilisation de Django pour leur site web et CMS de contenu éditorial.	Dans ce type de site web riche en contenu médiatique, l'édition, les médias, les images, vidéos — tout cela impose des besoins de gestion de fichiers, d'organisation de templates, etc. Django s'y prête bien.

NASA	Pour certains de leurs portails de science (images, contenus scientifiques) etc.	Exigences de fiabilité, sécurité, capacité à servir du contenu volumineux. Bon exemple de Django utilisé pour des usages publics, documentaires. Mais probablement pas pour tous leurs services (les plus critiques ou temps réel).

En résumé :

Domaine	Django excelle	Django est moins adapté
Applications web structurées (blogs, intranets, e-commerce, plateformes éditoriales, diffusion de contenu)	Rapidité, cohérence, sécurité	
Applications temps réel ou asynchrones (chat, streaming, API massives)		Plus adapté : FastAPI, Node.js
Back-office interne	Admin auto-généré, auth intégrée	
Microservices ou API-first		Framework trop monolithique

Concepts fondamentaux : MVT, ORM

1.1. Modèle MVT

Django gère lui même la partie contrôleur (gestion des requêtes du client, des droits sur les actions) du MVC
On parle donc plutôt de “Modèle Vue Template”
Un template est un fichier html récupéré par la vue, analysé par le framework comme s’il s’agissait d’un fichier avec du code avant d’être envoyé au client.
Django fournit un moteur de templates html utile permettant, dans le code HTML d’afficher des variables, utiliser des conditionnelles ainsi que des boucles for

Séparation claire :

- **Modèle** → données (ORM)
  - Il correspond à la structure de la base de données.
  - Chaque modèle est une classe Python qui hérite de django.db.models.Model.
  - Django fournit un ORM (Object-Relational Mapping), ce qui permet de manipuler les données sans écrire de SQL.

```python
#models.py

from django.db import models

class Article(models.Model):
    titre = models.CharField(max_length=100)
    contenu = models.TextField()
    date_pub = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.titre
```

- **Vue** → logique métier
  - La vue reçoit une requête du client (HTTP Request) et renvoie une réponse (HTTP Response).
  - C’est ici qu’on décide quelles données afficher, quels modèles utiliser, et quel template charger.
  - Une vue est souvent une fonction ou une classe en Python.

```python
#views.py

from django.shortcuts import render
from .models import Article

def index(request):
    articles = Article.objects.all()
    return render(request, 'blog/index.html', {'articles': articles})
```

- **Template** → affichage HTML
    - C'est un fichier HTML qui contient du code Django pour afficher dynamiquement les données.
    - Django fournit un moteur de templates : il permet d'insérer des variables, des conditions, et des boucles directement dans le HTM

```
2
3   //index.html
4
5   <h1>Liste des articles</h1>
6   <ul>
7     {% for article in articles %}
8       <li>{{ article.titre }} — publié le {{ article.date_pub }}</li>
9     {% endfor %}
10  </ul>
11
```

1.2. ORM déclaratif

ORM veut dire *Object Relational Mapper*.
→ C'est un système qui permet de manipuler la base de données directement en Python, sans écrire de requêtes SQL.

Au lieu d'écrire du SQL à la main, on déclare des classes Python qui représentent les tables de la base.
Chaque attribut de la classe correspond à une colonne, et Django se charge de traduire automatiquement les actions Python en requêtes SQL.

Par exemple :

```
class Article(models.Model):
  titre = models.CharField(max_length=100)
  contenu = models.TextField()
```

Ici, Django crée automatiquement une table article avec deux colonnes : titre et contenu.

Tu peux ensuite faire :

Article.objects.create(titre="Bonjour", contenu="Ceci est un article")

au lieu d'écrire une requête SQL manuelle.

→ L'ORM de Django permet de gérer la base de données comme des objets Python, sans jamais écrire de SQL. Il traduit le code Python en requêtes SQL pour nous.

Requêtes utiles :

- **Création** : NomModele.objects.create()
- **Lecture** : NomModele.objects.all() / NomModele.objects.get() / NomModele.objects.filter()
- **Modification** : modifier les attributs → save()
- **Suppression** : NomModele.objects.delete()

1.3. Les migrations dans Django

Les migrations sont le mécanisme qui permet à Django de synchroniser les modèles Python avec la base de données.
→ Elles traduisent les changements faits dans les classes de modèles (a**jout, suppression ou modification de champs, de tables, etc**.) en instructions SQL que Django exécute automatiquement.

1. Tu crées ou modifies un modèle dans models.py. Tu exécutes la commande :

   python manage.py makemigrations

→ Django analyse ton code et crée un fichier de migration (dans le dossier migrations/) qui décrit les changements à appliquer à la base de données.

2. Tu appliques ces migrations avec :

   python manage.py migrate

→ Django exécute alors les requêtes SQL nécessaires pour mettre à jour la base de données selon tes modèles.

Commandes principales:

Commande	Rôle
python manage.py makemigrations	Génère les fichiers de migration à partir des

	modifications dans les modèles
python manage.py migrate	Applique les migrations à la base de données
python manage.py showmigrations	Affiche la liste des migrations existantes et leur état (appliquées ou non)
python manage.py sqlmigrate <app_name> <numéro_migration>	Montre le SQL qui sera exécuté pour une migration donnée

Concepts fondamentaux : Routage explicite

Le routage explicite dans Django se fait via via le fichier urls.py.
Django adopte un routage explicite, c'est-à-dire que chaque route doit être déclarée manuellement dans un fichier Python, sans détection automatique des endpoints.
(Pour rappel : Un endpoint (ou point de terminaison) est une adresse d'accès à une ressource ou une fonctionnalité d'une application, autrement dit un endpoint = une URL + une méthode HTTP")

Le routage explicite favorise la lisibilité et la maintenance à grande échelle.

Il permet aussi de **découper le routage par application**, ce qui améliore la modularité.

Le routage explicite renforce la **séparation des responsabilités** :

- le contrôleur logique (vue) est dans views.py
- la déclaration du chemin d'accès est dans urls.py

Exemple :

```python
# projet/urls.py
from django.urls import path
from app.views import liste_articles, detail_article

urlpatterns = [
    path('articles/', liste_articles, name='liste_articles'),
    path('articles/<int:id>/', detail_article, name='detail_article'),
]
```

Les formulaires (Forms)

Les formulaires Django permettent de récupérer, valider et traiter des données utilisateur de manière sécurisée et cohérente.

Ils constituent un pont entre :

- les templates (affichage HTML du formulaire),
- les vues (logique de traitement),
- et les modèles (enregistrement des données).

Deux types de formulaires principaux

forms.Form

➜ Pour créer un formulaire manuel, indépendant d'un modèle.

Exemple :

```python
from django import forms

class ContactForm(forms.Form):
    nom = forms.CharField(max_length=50)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)
```

forms.ModelForm

➜ Pour créer un formulaire lié directement à un modèle de la base de données.

Exemple :

```python
from django import forms
from .models import Article

class ArticleForm(forms.ModelForm):
    class Meta:
        model = Article
        fields = ['titre', 'contenu']

```

Cycle de vie d’un formulaire Django

1. **Affichage du formulaire dans le template :**

```
<form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Envoyer</button>
</form>

```

2. **Soumission :** Django reçoit les données via une requête POST.

3. **Validation automatique :**

    - Les champs obligatoires, formats d’e-mail, tailles, etc. sont vérifiés.
    - Si le formulaire est valide → on peut enregistrer les données.
    - Sinon → les erreurs sont renvoyées au template.

4. **Traitement dans la vue :**

```python
from django.shortcuts import render, redirect
from .forms import ArticleForm

def ajouter_article(request):
    if request.method == "POST":
        form = ArticleForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('liste_articles')
    else:
        form = ArticleForm()
    return render(request, 'ajouter_article.html', {'form': form})

```

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**

Concepts fondamentaux : Système d’applications

Chaque app est une **unité logique et fonctionnelle**, regroupant ses propres :

- modèles (base de données),
- vues (logique métier),
- templates (interface),
- tests,
- fichiers de configuration (urls.py, admin.py, etc.).

Autrement dit, une *app* Django, c’est **un mini-module web complet**, que tu peux développer, tester et réutiliser ailleurs.

Chaque application peut être **autonome** :

Tu peux la copier-coller dans un autre projet Django sans tout casser, tant qu’elle est référencée dans INSTALLED_APPS.

Aspect	Apport
Modularité	Travail en équipe : chaque groupe peut travailler sur une app.
Réutilisabilité	Une app peut être extraite et réutilisée ailleurs (ex. “blog”, “auth”).
Maintenance	On peut tester et déployer une app indépendamment.
Séparation des responsabilités	Logique claire entre le projet global et les modules locaux.

Les Tests (tests unitaires et d'intégration)

Chaque app Django inclut généralement un fichier tests.py (ou un dossier tests/) où tu peux écrire tes tests automatisés.

Ces tests permettent de vérifier :

- que les modèles enregistrent et récupèrent bien les données,
- que les vues renvoient les bonnes réponses HTTP,
- que les templates s'affichent correctement,
- que les formulaires et les URLs fonctionnent comme prévu.

Exemple :

tests.py

```python
from django.test import TestCase
from .models import Article

class ArticleModelTest(TestCase):
    def test_creation_article(self):
        article = Article.objects.create(titre="Test", contenu="Ceci est un test")
        self.assertEqual(article.titre, "Test")

```

Exécution des tests :

```
>>> python manage.py test
```

→ **Django détecte automatiquement tous les fichiers de test dans les apps et exécute les scénarios.**

Limites à connaître

- Trop d'apps = risque de dépendances croisées (importations circulaires, logique dupliquée).
- Certaines configurations globales (authentification, templates, middleware) peuvent lier fortement les apps entre elles.
- Les tests mal structurés ou redondants peuvent ralentir le développement s'ils ne sont pas bien ciblés.

Maturité et communauté

1. Documentation

La documentation de Django est l'une des plus complètes et pédagogiques du monde open source. Elle est structurée et maintenue à jour à chaque version.
On y trouve :

- des guides pour débuter (installation, premiers projets),
- des tutoriels avancés (authentification, ORM, déploiement),
- et une référence technique détaillée pour chaque module du framework.

Cette qualité de documentation a beaucoup contribué à la popularité de Django : même sans expérience, un développeur peut progresser vite.

2. Ecosystème de packages et sa fiabilité :

L'écosystème Django est l'un des plus riches de l'univers Python.
Il existe des milliers de **packages officiels et communautaires** permettant d'étendre les fonctionnalités sans réinventer la roue.
Ces extensions sont **fiables** car :

- Souvent maintenues par des **équipes expérimentées ou des entreprises**.
- Suivent le rythme des mises à jour officielles de Django.
- Sont validées par la **Django Software Foundation (DSF)** ou référencées sur djangopackages.org.

Package	Rôle principal	Fiabilité
Django REST Framework	Création d'API RESTful modernes	Maintenu activement, utilisé par Netflix, Mozilla
Celery	Exécution de tâches asynchrones	Très populaire, compatible Django 5
django-allauth	Authentification sociale (Google, GitHub, etc.)	Mises à jour régulières
django-crispy-forms	Formulaires HTML élégants	Fortement utilisé
django-debug-toolbar	Débogage et analyse de requêtes SQL	Outil standard des devs Django

3. Communauté internationale :

Django est soutenu par une **communauté mondiale très active** :

- Plusieurs milliers de **contributeurs sur GitHub** ;
- Une **conférence annuelle DjangoCon** organisée sur plusieurs continents (Europe, US, Afrique) ;
- Des **meetups locaux** et un forum officiel d'entraide.

Le projet est géré par la **Django Software Foundation (DSF)**, qui veille à sa gouvernance, sa documentation, et la sécurité des nouvelles versions.

On peut aussi mentionner **DjangoGirls**, une initiative mondiale pour former gratuitement des femmes au développement web avec Django : plus de 1000 événements dans 90 pays et un site Internet regroupant tutoriels et informations sur les événements.

4. Maturité du code :

Django suit un **cycle de version stable** avec des **releases LTS (Long-Term Support)** garantissant 3 ans de maintenance.
Le code source est :

- intégralement **open source et audité** ;
- accompagné de plus de **15 000 tests unitaires** automatisés ;
- respectueux des conventions Python (PEP8) et des bonnes pratiques de sécurité.

Cette rigueur assure une **stabilité industrielle** : Django est adopté aussi bien par des startups que par des géants comme Instagram ou la NASA.

Points forts et limites

Points forts :

- Système d'authentification intégré
  - pas besoin d'ajouter de librairies comparé à d'autres frameworks
  - paramétrage dans un fichier de config -> simple à faire
  - permet de faire des choses assez poussées (limitation des tentatives de connexion, système de rôle sur les utilisateurs…)
- Bonne documentation en anglais
  - contrairement à d'autres frameworks, documentation fournie et maintenue
  - beaucoup d'exemples de code
  - Mais seule une partie traduite en français (nuance du propos)
- Bonne gestion des exceptions et backtraces
  - Pages d'erreurs web très fournies en informations
  - Détaillé
- Django inclut la prévention des attaques courantes
  - comme la contrefaçon de requêtes Cross-site (CSRF)
  - SQL Injections.

Django fournit aux développeurs une boîte à outils complète comme vu précédemment (ORM,Sécurité, templating, Authentification intégré, Admin intégré), ce qui en fait un choix attrayant pour la création d'applications Web.

Points faibles:

- Ne permet pas seul l'intégration d'AJAX (architecture informatique pour système dynamique en js et xml) côté client web
    - pas de modifications poussées du dom comme en react
- Manière standard de définir et exécuter les tâches
    - on ne peut pas sortir du cadre donc obligé de respecter les normes du framework.
- Convient difficilement aux petits projets (contrairement au framework Flask)
    - Django est créé dans le but de faire de grandes applications
    - Il gaspille beaucoup de bande passante pour des petits projets.
- L'ORM de Django peut causer des problèmes de performance
    - Il est parfois trop simple d'utilisation et peut cacher des sous-requêtes si mal utilisé

Conclusion

- Django est un framework “batteries incluses” : il intègre ORM, formulaires, authentification, sécurité, et interface d’administration.
- Conçu à l’origine pour des besoins éditoriaux (publication rapide de contenu), il privilégie la productivité et la cohérence du code.
- Son architecture MVT (Model–View–Template) garantit une séparation claire des responsabilités.
- Un projet Django est composé d’applications autonomes : chaque app regroupe modèles, vues, templates et fichiers de configuration.
- L’ORM déclaratif évite d’écrire du SQL
- L’interface d’administration est générée automatiquement à partir des modèles Python — un outil puissant pour la gestion interne et le prototypage rapide.
- L’admin est personnalisable et intégrée au système d’authentification.
- Le trio User / Group / Permission offre un contrôle d’accès fin et extensible.
- Chaque application peut définir son propre `admin.py`, renforçant la clarté et la maintenabilité du projet.
- Django bénéficie d’une documentation exemplaire, d’une grande communauté.
- Cycle de release stable avec versions LTS et un code hautement testé.
- Adopté par de grands acteurs (Instagram, NASA, Pinterest) pour sa stabilité et sa sécurité.

Sources :

- https://www.django-cms.org/en/blog/2021/09/23/what-can-you-build-with-python/
- https://medium.com/hackerdawn/9-companies-that-are-using-django-to-scale-up-e8e50e9bebd1
- https://nglogic.com/django-apps/
- https://djangogirls.org/en/
- https://en.wikipedia.org/wiki/Django_(web_framework)
- https://www.netguru.com/blog/django-pros-and-cons
- https://medium.com/%40ITservices_expert/django-a-comprehensive-examination-of-advantages-limitations-and-competitive-superiority-cb5860b02887
- https://callmerohit.medium.com/is-django-dying-an-honest-look-at-the-future-94cb617c1f88
- https://www.youtube.com/watch?v=l0QEGvAX8rU
- https://docs.djangoproject.com/fr/5.2/intro/tutorial01/