École Supérieure de Technologie de Casablanca

Département Génie Informatique

Framework Python-Django

Construction d'Applications Web Scalables

Support de cours

Pr. Moad Hicham Safhi

Département Génie Informatique
Email : h.m.safhi@gmail.com
Website : https://safhi.me

Dernière mise à jour : 9 janvier 2026

Framework Python-Django

Table des matières

1 CHAPITRE 1 : INTRODUCTION AU FRAMEWORK DJANGO

1.1 CONTEXTE ET DÉFINITIONS FONDAMENTALES

1.1.1 Qu'est-ce qu'un framework web ?

Imaginez que vous devez construire une maison. Vous pourriez fabriquer chaque brique, chaque fenêtre, chaque porte vous-même. Ou vous pourriez utiliser des éléments préfabriqués et des outils spécialisés qui accélèrent la construction. Un framework web, c'est comme cette boîte à outils complète pour construire des applications web.

Sans framework, vous devriez réécrire les mêmes fonctionnalités à chaque projet : gérer les connexions à la base de données, traiter les formulaires, assurer la sécurité. Avec un framework, ces fonctionnalités sont déjà construites et testées.

1.1.2 Pourquoi Django spécifiquement ?

Django est un framework "batteries included" - toutes les piles sont incluses. Quand vous installez Django, vous obtenez immédiatement :

— Un système de gestion de base de données (ORM).
— Une interface d'administration automatique.
— Un système d'authentification des utilisateurs.
— Une protection contre les attaques courantes.
— Un système de templates pour générer du HTML.

Des entreprises comme Instagram, Pinterest et Spotify utilisent Django pour gérer des millions d'utilisateurs. Si c'est assez bon pour eux, c'est probablement assez bon pour vos projets !

Django est :

— **Demandé** : Beaucoup d'entreprises recherchent des développeurs Django. -**Stable** : Existe depuis 2005, avec un support à long terme.
— **Bien documenté** : Une des meilleures documentations dans le monde open source.
— **Communautaire** : Une communauté active et accueillante.

1.2 ARCHITECTURE MVT - LE CŒUR DE DJANGO

1.2.1 Comprendre MVT (Modèle-Vue-Template)

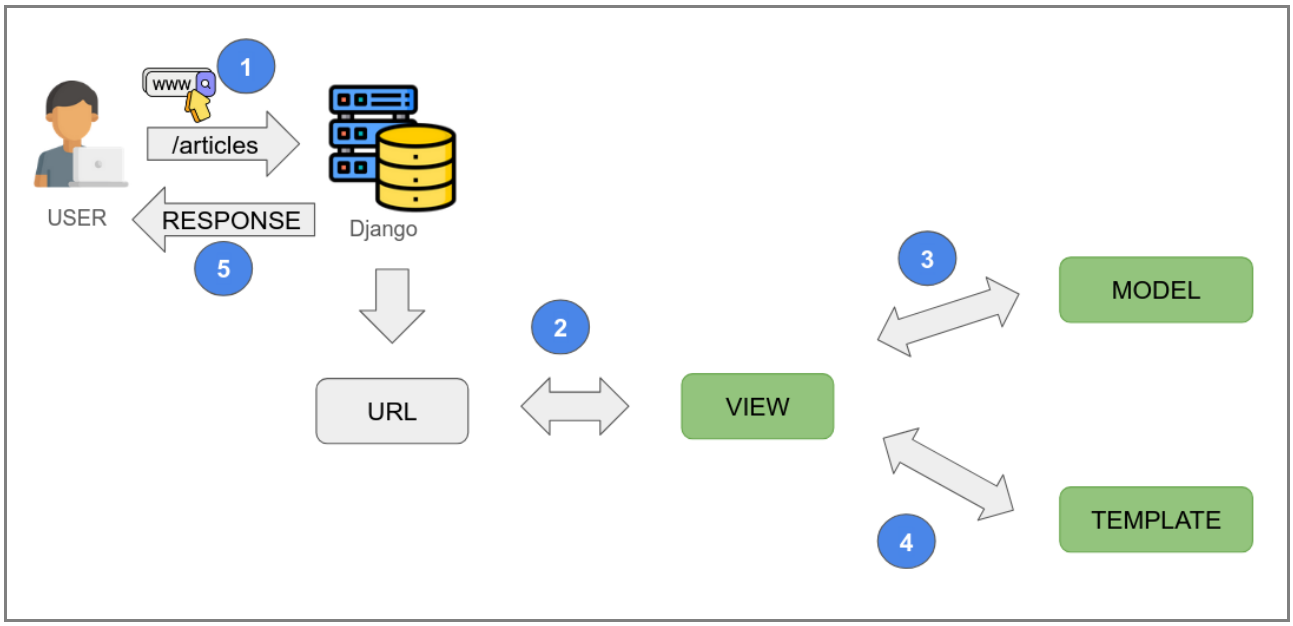
Django utilise une architecture appelée MVT, qui ressemble beaucoup au MVC mais avec des noms différents :

Modèle : C'est la représentation de vos données. Si vous construisez un blog, vos modèles seraient "Article", "Auteur", "Commentaire". Le modèle dit à Django comment stocker et récupérer les données dans la base de données.

Vue : C'est le cerveau de votre application. La vue reçoit une requête web, décide quoi faire avec, et renvoie une réponse. Par exemple, quand un utilisateur demande la page d'accueil, la vue récupère les derniers articles et les prépare pour l'affichage.

Template : C'est l'apparence de votre site. Les templates sont des fichiers HTML avec des espaces réservés pour les données dynamiques. Ils se chargent de l'affichage, pas de la logique.

1.2.2 Comment ces trois parties collaborent ?



1. Un utilisateur demande une URL (par exemple : /articles/)
2. Django trouve la vue associée à cette URL
3. La vue interroge la base de données via les modèles
4. La vue prend les données et les envoie à un template
5. Le template génère du HTML qui est renvoyé au navigateur

Cette séparation rend votre code plus organisé et plus facile à maintenir.

1.3 INSTALLATION ET PREMIERS PAS

1.3.1 Préparer son environnement

Avant d'installer Django, créez un environnement virtuel. C'est comme une bulle isolée où vous installez Django sans affecter les autres projets Python sur votre ordinateur.

Pour créer et activer un environnement virtuel :

```
safhi@django-course:~$
safhi@django-course:~$ mkdir TP_DJANGO
safhi@django-course:~$ cd TP_DJANGO/
safhi@django-course:~/TP_DJANGO$
safhi@django-course:~/TP_DJANGO$ python3 -m venv django_env
safhi@django-course:~/TP_DJANGO$ source django_env/bin/activate
(django_env) safhi@django-course:~/TP_DJANGO$
(django_env) safhi@django-course:~/TP_DJANGO$
```

1.3.2 Installer Django

Une fois l'environnement activé :

```
pip install django
```

Pour vérifier que Django est bien installé :

```
django-admin --version
```

```
(django_env) safhi@django-course:~/TP_DJANGO$
(django_env) safhi@django-course:~/TP_DJANGO$ pip install django
Collecting django
  Downloading django-6.0.1-py3-none-any.whl.metadata (3.9 kB)
Collecting asgiref>=3.9.1 (from django)
  Downloading asgiref-3.11.0-py3-none-any.whl.metadata (9.3 kB)
Collecting sqlparse>=0.5.0 (from django)
  Using cached sqlparse-0.5.5-py3-none-any.whl.metadata (4.7 kB)
Downloading django-6.0.1-py3-none-any.whl (8.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8.3/8.3 MB 4.7 MB/s eta 0:00:00
Downloading asgiref-3.11.0-py3-none-any.whl (24 kB)
Using cached sqlparse-0.5.5-py3-none-any.whl (46 kB)
Installing collected packages: sqlparse, asgiref, django
Successfully installed asgiref-3.11.0 django-6.0.1 sqlparse-0.5.5
(django_env) safhi@django-course:~/TP_DJANGO$
(django_env) safhi@django-course:~/TP_DJANGO$ django-admin --version
6.0.1
(django_env) safhi@django-course:~/TP_DJANGO$
```

1.3.3 Créer votre premier projet

Un projet Django est comme un conteneur pour votre application web complète :

```
(django_env) safhi@django-course:~/TP_DJANGO$
(django_env) safhi@django-course:~/TP_DJANGO$
(django_env) safhi@django-course:~/TP_DJANGO$ django-admin startproject monprojet
(django_env) safhi@django-course:~/TP_DJANGO$
(django_env) safhi@django-course:~/TP_DJANGO$ ls
django_env  monprojet
(django_env) safhi@django-course:~/TP_DJANGO$
```

Cela crée une structure de dossiers :

```
(django_env) safhi@django-course:~/TP_DJANGO$ tree monprojet/
monprojet/
├── manage.py
└── monprojet
    ├── asgi.py
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py

2 directories, 6 files
(django_env) safhi@django-course:~/TP_DJANGO$
```

```
monprojet/
   manage.py          # Outil de commande
   monprojet/
```

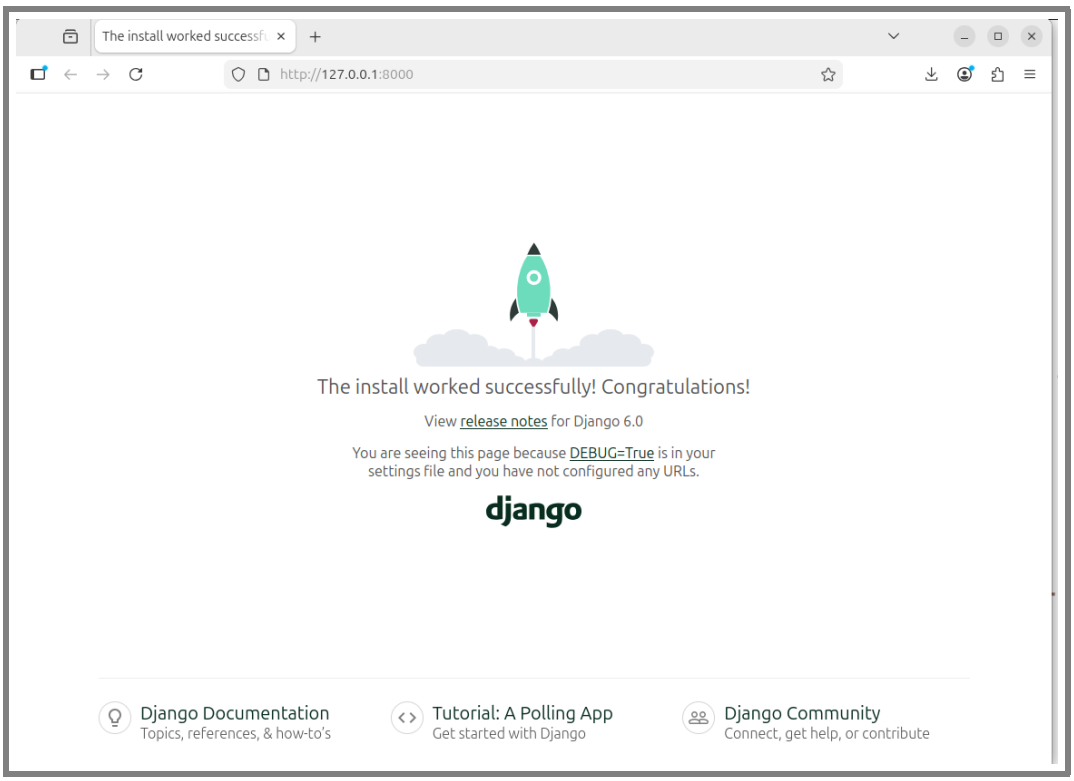
```
__init__.py
settings.py     # Configuration
urls.py         # Routes URLs
asgi.py         # Pour serveurs modernes
wsgi.py         # Pour serveurs web
```

1.3.4 Lancer le serveur de développement

```
python manage.py runserver
```

Ouvrez votre navigateur à l'adresse http://127.0.0.1:8000/

Vous devriez voir la page de bienvenue de Django ! Ce serveur est uniquement pour le développement, pas pour la production.



1.4 LES COMPOSANTS CLÉS EXPLIQUÉS

1.4.1 Le fichier `settings.py` : Le centre de contrôle

Ce fichier contient toute la configuration de votre projet :

— La base de données à utiliser (SQLite pour le développement, PostgreSQL pour la production).

```
 GNU nano 7.2                                    monprojet/settings.py

# Database
# https://docs.djangoproject.com/en/6.0/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

— La liste des applications installées.

```
 GNU nano 7.2                                    monprojet/settings.py


# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

— Les paramètres de sécurité.

```
 GNU nano 7.2                                    monprojet/settings.py

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'django-insecure-5js5abfnm7fq&8m                    '

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = []
```

— La langue et le fuseau horaire.

```
 GNU nano 7.2                                    monprojet/settings.py

# Internationalization
# https://docs.djangoproject.com/en/6.0/topics/i18n/

LANGUAGE_CODE = 'en-us'

TIME_ZONE = 'UTC'

USE_I18N = True

USE_TZ = True
```

— Les chemins vers les templates et fichiers statiques.

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/6.0/howto/static-files/

STATIC_URL = 'static/'
```

1.4.2 Les URLs : Le système de routage

Le fichier `urls.py` fait le lien entre les adresses web et votre code. C'est comme un réceptionniste qui dirige les visiteurs vers le bon bureau.

Exemple simple :

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.accueil),  # La page d'accueil
    path('articles/', views.liste_articles),  # Liste des articles
    path('article/<int:id>/', views.detail_article),  # Détail d'un article
]
```

1.4.3 Les vues : La logique métier

Une vue est une fonction Python qui reçoit une requête web et retourne une réponse. C'est là que vous écrivez la logique de votre application.

Exemple d'une vue simple :

```
from django.http import HttpResponse
from django.shortcuts import render
from .models import Article

def accueil(request):
    """Affiche la page d'accueil"""
    return HttpResponse("Bienvenue sur mon site!")

def liste_articles(request):
    """Affiche la liste des articles"""
    articles = Article.objects.all()  # Récupère tous les articles
    return render(request, 'articles/liste.html', {'articles': articles})
```

1.4.4 Les templates : L'apparence du site

Les templates séparent la présentation de la logique. Au lieu de mélanger HTML et Python, vous gardez le HTML dans des fichiers templates.

Un template simple :

```
<!DOCTYPE html>
<html>
<head>
    <title>Mon Blog</title>
</head>
<body>
    <h1>Articles récents</h1>
```

```
    {% for article in articles %}
    <article>
        <h2>{{ article.titre }}</h2>
        <p>Publié le {{ article.date_publication|date:"d/m/Y" }}</p>
        <p>{{ article.contenu|truncatechars:200 }}</p>
    </article>
    {% empty %}
    <p>Aucun article pour le moment.</p>
    {% endfor %}
</body>
</html>
```

1.4.5 Les modèles : La structure des données

Les modèles définissent la structure de votre base de données en Python pur. Django traduit ensuite ces modèles en instructions SQL.

Exemple de modèle :

```python
from django.db import models

class Article(models.Model):
    """Modèle représentant un article de blog"""
    titre = models.CharField(max_length=200)
    contenu = models.TextField()
    date_publication = models.DateTimeField(auto_now_add=True)
    auteur = models.ForeignKey('Auteur', on_delete=models.CASCADE)
    publie = models.BooleanField(default=False)

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

Django crée automatiquement une table en base de données pour ce modèle, avec toutes les colonnes nécessaires.

1.5 LE LANGAGE DE TEMPLATES DJANGO (DTL)

1.5.1 Pourquoi un langage de templates spécial ?

Le Django Template Language (DTL) est conçu pour être suffisamment puissant pour créer des pages dynamiques, mais suffisamment limité pour éviter de mettre trop de logique dans les templates. C'est une question de séparation des responsabilités.

1.5.2 Les fonctionnalités principales du DTL

1.5.2.1 Les variables

Afficher une variable : `{ variable }`

```
<h1>{{ titre_article }}</h1>
<p>Par {{ auteur.nom }}</p>
```

1.5.2.2 Les filtres

Transformer les variables :

```
<!-- Mettre en majuscules -->
<p>{{ texte|upper }}</p>

<!-- Formater une date -->
<p>{{ date|date:"d F Y" }}</p>

<!-- Tronquer un texte -->
<p>{{ long_texte|truncatechars:100 }}</p>
```

1.5.2.3 Les balises

Les balises ajoutent de la logique aux templates :

```
<!-- Conditions -->
{% if user.is_authenticated %}
    <p>Bienvenue, {{ user.username }}!</p>
{% else %}
    <p>Veuillez vous connecter.</p>
{% endif %}

<!-- Boucles -->
<ul>
{% for article in articles %}
    <li>{{ article.titre }}</li>
{% empty %}
    <li>Aucun article disponible.</li>
{% endfor %}
</ul>
```

1.5.2.4 L'héritage de templates

C'est l'une des fonctionnalités les plus puissantes. Vous créez un template de base avec une structure commune, et les autres templates héritent de cette base.

base.html (template de base) :

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block titre %}Mon Site{% endblock %}</title>
</head>
<body>
    <header>Mon en-tête commun</header>
    <main>
        {% block contenu %}
        {% endblock %}
    </main>
    <footer>Mon pied de page commun</footer>
</body>
</html>
```

page_accueil.html (template enfant) :

```
{% extends "base.html" %}

{% block titre %}Accueil - Mon Site{% endblock %}

{% block contenu %}
<h1>Bienvenue sur notre site</h1>
<p>Contenu spécifique à la page d'accueil...</p>
{% endblock %}
```

1.5.2.5 L'inclusion de templates

Pour réutiliser des morceaux de HTML à plusieurs endroits :

```
{% include "includes/menu.html" %}
```

1.5.3 Sécurité dans les templates

Django échappe automatiquement les caractères dangereux dans les variables pour éviter les attaques XSS (Cross-Site Scripting). Si vous avez besoin d'afficher du HTML sûr, vous pouvez utiliser le filtre `|safe` :

```
<p>{{ html_securise|safe }}</p>
```

Mais utilisez-le avec prudence, seulement quand vous êtes sûr que le contenu est sûr !

2 CHAPITRE 2 : LES APPLICATIONS DJANGO, MODULARITÉ ET RÉUTILISATION

2.1 POURQUOI DÉCOUPER EN APPLICATIONS ?

Imaginez que vous construisez une maison. Vous ne mettez pas toute la plomberie, l'électricité, les murs et le toit dans un seul grand tas. Vous organisez par pièces : cuisine, salle de bain, chambre. Chaque pièce a une fonction spécifique.

Dans Django, c'est pareil. Une application est une **pièce fonctionnelle** de votre projet. Par exemple :

— Une application `blog` pour les articles et commentaires.
— Une application `utilisateurs` pour l'inscription et la connexion.
— Une application `boutique` pour les produits et commandes.

Cette organisation permet de :

1. **Réutiliser** des applications dans d'autres projets.
2. **Maintenir** le code plus facilement.
3. **Travailler en équipe** (chaque développeur sur une app différente).
4. **Tester** chaque fonctionnalité séparément.

2.2 PROJET vs APPLICATION : LA DIFFÉRENCE

2.2.1 Le Projet : La maison complète

Le projet est le conteneur global. Il contient :

— La configuration (settings.py).
— Les URLs principales (urls.py).
— Le serveur (wsgi.py, asgi.py).

Quand vous créez un projet avec `startproject`, Django crée ce conteneur.

2.2.2 L'Application : Une pièce de la maison

Une application est un module autonome qui gère une fonctionnalité spécifique.

Analogie concrète :

```
MonSite/                    Le projet (la maison)
  blog/                   Application blog (la cuisine)
  utilisateurs/           Application utilisateurs (la salle de bain)
  boutique/               Application boutique (le salon)
  config/                 Configuration (les fondations)
```

2.3 CRÉER UNE APPLICATION : LA COMMANDE STARAPP

2.3.1 Où et comment lancer la commande ?

1. Assurez-vous d'être dans le bon dossier :

```
cd monprojet  # Le dossier qui contient manage.py
```

2. Créez l'application :

```
python manage.py startapp blog
```

```
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$ ls
db.sqlite3  manage.py  monprojet
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$ python manage.py startapp blog
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$ ls
blog  db.sqlite3  manage.py  monprojet
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$
```

2.3.2 Ce que la commande crée

Après `startapp blog`, vous obtenez cette structure :

```
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$ tree blog/
blog/
├── admin.py
├── apps.py
├── __init__.py
├── migrations
│   └── __init__.py
├── models.py
├── tests.py
└── views.py

2 directories, 7 files
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$
```

2.4 ENREGISTRER L'APPLICATION DANS LE PROJET

2.4.1 Étape cruciale souvent oubliée !

Après avoir créé une application, vous devez l'annoncer à Django. C'est comme dire à la mairie que vous avez ajouté une nouvelle pièce à votre maison.

Ouvrez `settings.py` dans votre projet et ajoutez le nom de l'application à `INSTALLED_APPS` :

```
  GNU nano 7.2                                    monprojet/settings.py
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',
]
```

Pourquoi c'est important ? Sans cette étape :

— Django ne connaît pas votre application.
— Les modèles ne sont pas créés en base de données.
— L'admin ne fonctionne pas.
— Les templates ne sont pas trouvés.

2.5 EXEMPLE COMPLET : CRÉATION D'UNE APPLICATION BLOG

2.5.1 Étape par étape

1. **Créer le projet** (si pas encore fait) :

```
django-admin startproject monprojet
cd monprojet
```

2. **Créer l'application blog** :

```
python manage.py startapp blog
```

3. **Enregistrer l'application** dans `settings.py` :

```
# monprojet/settings.py
INSTALLED_APPS = [
    # ... applications par défaut ...
    'blog',  #  AJOUT
]
```

4. **Créer un modèle** dans `blog/models.py` :

```
 GNU nano 7.2                    blog/models.py
from django.db import models

# Create your models here.



class Article(models.Model):
    titre = models.CharField(max_length=200)
    contenu = models.TextField()
    date_publication = models.DateTimeField(auto_now_add=True)

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

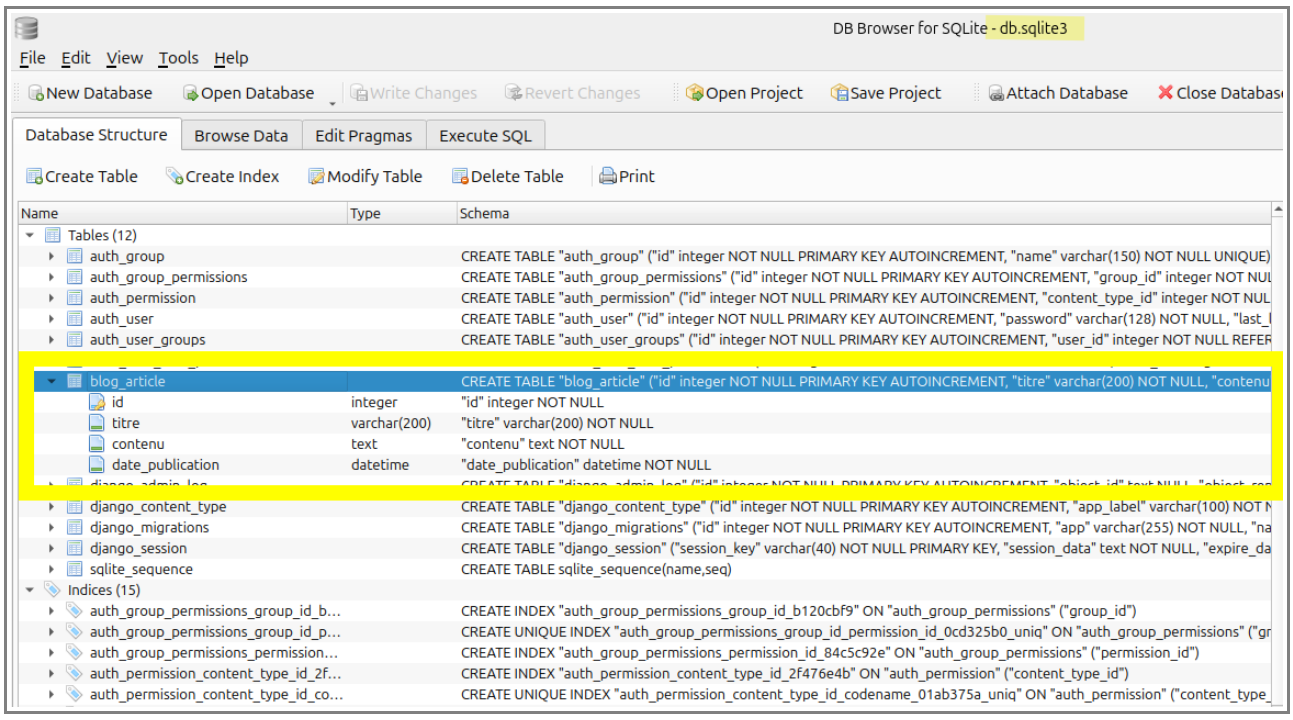
5. **Créer une migration** (traduire le modèle en SQL) :

```
python manage.py makemigrations blog
```

6. **Appliquer la migration** (créer la table en base de données) :

```
python manage.py migrate
```

```
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$ python manage.py makemigrations blog
Migrations for 'blog':
  blog/migrations/0001_initial.py
    + Create model Article
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying blog.0001_initial... OK
  Applying sessions.0001_initial... OK
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$
(django_env) safhi@django-course:~/TP_DJANGO/monprojet$
```



5. **Créer une vue** dans `blog/views.py` :

```
 GNU nano 7.2                    blog/views.py
from django.shortcuts import render

from .models import Article

# Create your views here.


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

5. **Créer un template** dans `blog/templates/blog/liste.html` :

```
 GNU nano 7.2            blog/templates/blog/liste.html

<!DOCTYPE html>
<html>
<head>
    <title>Mon Blog</title>
</head>
<body>
    <h1>Articles du blog</h1>
    {% for article in articles %}
    <h2>{{ article.titre }}</h2>
    <p>{{ article.contenu|truncatewords:50 }}</p>
    {% endfor %}
</body>
</html>
```

5. **Créer un fichier d'URLs** pour l'application (`blog/urls.py`) :

```
 GNU nano 7.2                          blog/urls.py


from django.urls import path
from . import views

urlpatterns = [
    path('', views.liste_articles, name='liste_articles'),
]
```

5. **Inclure les URLs de l'application** dans le projet (`monprojet/urls.py`) :

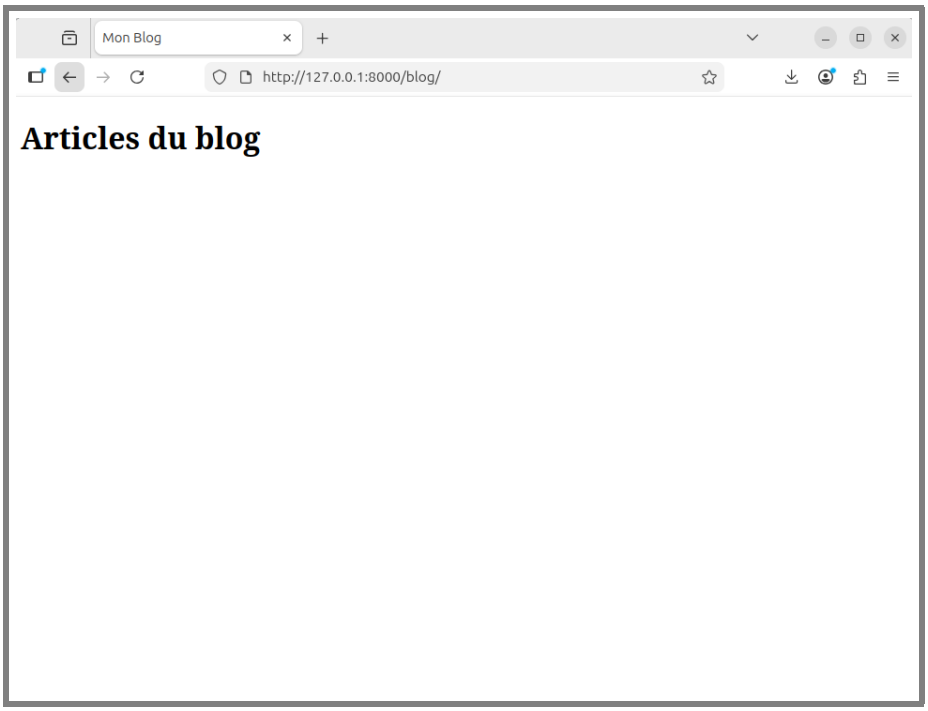
```
 GNU nano 7.2                        monprojet/urls.py
"""
URL configuration for monprojet project.

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/6.0/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls')),

]
```

5. **Tester** : python manage.py runserver puis allez sur http://127.0.0.1:8000/blog/



2.6 BONNES PRATIQUES POUR LES APPLICATIONS

2.6.1 Nommage des applications

— Singulier : `blog` pas `blogs`.
— Court et descriptif : `utilisateurs` pas `gestion_des_utilisateurs_connectes`.
— Pas de caractères spéciaux : `blog` pas `blog-app`.

2.6.2 Quand créer une nouvelle application ?

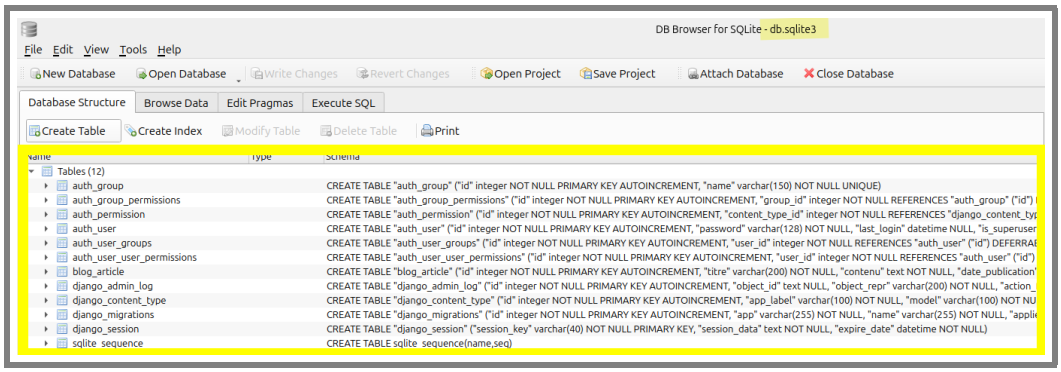
Créez une nouvelle application quand : - La fonctionnalité peut être nommée clairement (blog, utilisateurs, boutique). - Elle a ses **propres modèles** (tables en base de données). - Elle pourrait être **réutilisée** dans un autre projet . - L'application actuelle dépasse 500 lignes de code.

2.7 LES APPLICATIONS FOURNIES PAR DJANGO

Django inclut déjà plusieurs applications prêtes à l'emploi. Dans `INSTALLED_APPS` par défaut :

```
INSTALLED_APPS = [
    'django.contrib.admin',       # Interface d'administration
    'django.contrib.auth',        # Système d'authentification
    'django.contrib.contenttypes', # Gestion des types de contenu
    'django.contrib.sessions',    # Gestion des sessions
    'django.contrib.messages',    # Système de messages
    'django.contrib.staticfiles', # Gestion des fichiers statiques
]
```

Ces applications sont déjà migrées (tables créées en base de données) quand vous lancez `python manage.py migrate`.



2.8 L'ORM DJANGO : LA BASE DE DONNÉES SANS SQL

2.8.1 Qu'est-ce qu'un ORM ?

ORM signifie Object-Relational Mapping. C'est une couche qui vous permet d'utiliser des objets Python pour interagir avec votre base de données, au lieu d'écrire du SQL manuellement.

2.8.2 Avantages de l'ORM

1. **Écriture en Python pur** : Pas besoin d'apprendre un langage différent pour chaque base de données.
2. **Sécurité** : Protection intégrée contre les injections SQL.
3. **Portabilité** : Le même code fonctionne avec SQLite, PostgreSQL, MySQL, etc.
4. **Productivité** : Moins de code à écrire et à maintenir.

2.8.3 Exemples d'opérations avec l'ORM

2.8.3.1 Créer un objet

```
# Au lieu de: INSERT INTO article (titre, contenu)  VALUES ('Django', 'cours')
article = Article(titre="Django", contenu="cours")
article.save()
```

Ou en une ligne :

```
Article.objects.create(titre="Mon titre", contenu="Mon contenu")
```

2.8.3.2 Lire des données

```
# Récupérer tous les articles
tous_les_articles = Article.objects.all()

# Récupérer un article spécifique
article = Article.objects.get(id=1)

# Filtrer les articles
articles_publies = Article.objects.filter(publie=True)
articles_recentes = Article.objects.filter(date_publication_year=2023)

# Articles triés par date
articles_ordonnes = Article.objects.all().order_by('-date_publication')
```

2.8.3.3 Mettre à jour

```
article = Article.objects.get(id=1)
article.titre = "Nouveau titre"
article.save()
```

2.8.3.4 Supprimer

```
article = Article.objects.get(id=1)
article.delete()
```

2.8.4 Relations entre modèles

Django gère trois types de relations :

1. **ForeignKey** : Une relation plusieurs-à-un (plusieurs articles pour un auteur).
2. **ManyToManyField** : Une relation plusieurs-à-plusieurs (un article peut avoir plusieurs tags, un tag peut être sur plusieurs articles).
3. **OneToOneField** : Une relation un-à-un (un utilisateur a un profil).

2.9 L'INTERFACE D'ADMINISTRATION

2.9.1 Une fonctionnalité révolutionnaire

L'interface d'admin de Django est générée automatiquement à partir de vos modèles. En quelques lignes de code, vous obtenez une interface complète pour gérer vos données.

2.9.2 Activer l'admin pour un modèle

Dans `admin.py` de votre application :

```
 GNU nano 7.2                    blog/admin.py
from django.contrib import admin

# Register your models here.


from .models import Article

admin.site.register(Article)
```

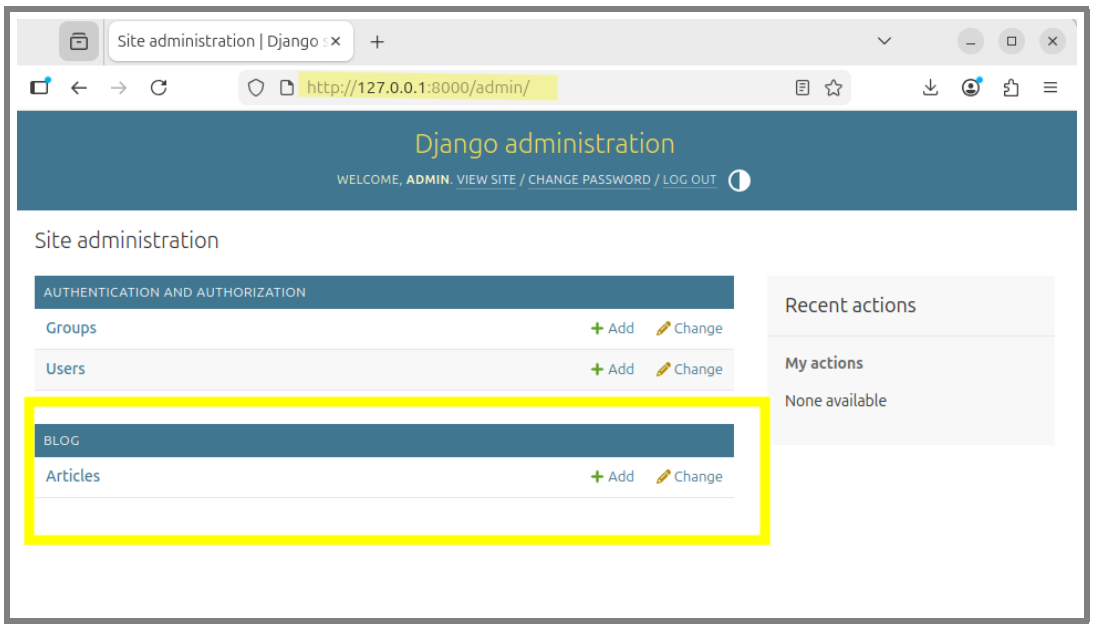
Et voilà ! Vous pouvez maintenant créer, lire, mettre à jour et supprimer des articles via l'interface d'admin à l'adresse `/admin`.

2.9.3 Créer un superutilisateur

Pour accéder à l'admin, vous avez besoin d'un compte administrateur :

```
python manage.py createsuperuser
```

Suivez les instructions pour créer votre compte, puis allez sur `/admin` pour vous connecter.



2.10 AUTHENTIFICATION ET SÉCURITÉ

2.10.1 Le système d'authentification intégré

Django inclut un système d'authentification complet :

- Inscription et connexion des utilisateurs.
- Gestion des mots de passe (hachés de manière sécurisée).
- Permissions et groupes.
- Sessions utilisateur.

2.10.2 Utiliser l'authentification dans les vues

Vérifier si un utilisateur est connecté :

```
def ma_vue(request):
    if request.user.is_authenticated:
        # Utilisateur connecté
        return HttpResponse(f"Bonjour {request.user.username}!")
    else:
        # Utilisateur non connecté
        return HttpResponse("Veuillez vous connecter.")
```

2.10.3 Les permissions

Django crée automatiquement des permissions pour chaque modèle :

- `app.add_modele` : permission d'ajouter.
- `app.change_modele` : permission de modifier.
- `app.delete_modele` : permission de supprimer.
- `app.view_modele` : permission de voir.

Vérifier une permission :

```
if request.user.has_perm('blog.add_article'):
    # L'utilisateur peut ajouter des articles
```

2.10.4 La protection CSRF

CSRF (Cross-Site Request Forgery) est une attaque où un site malveillant essaie de faire exécuter des actions à un utilisateur connecté sans son consentement.

Django protège contre cela avec un jeton CSRF. Dans vos templates, incluez toujours `{% csrf_token %}` dans les formulaires POST :

```
<form method="post">
    {% csrf_token %}
    <!-- Vos champs de formulaire ici -->
    <input type="submit" value="Envoyer">
</form>
```

2.11 PAGINATION - GÉRER LES LONGUES LISTES

2.11.1 Pourquoi paginer ?

Imaginez un site avec 10000 articles. Si vous affichez tous les articles sur une même page :

- La page mettra très longtemps à charger.
- L'expérience utilisateur sera mauvaise.
- Le référencement (SEO) en souffrira.

La solution : diviser les résultats en plusieurs pages.

2.11.2 Mise en œuvre de la pagination

Dans votre vue :

```
from django.core.paginator import Paginator

def liste_articles(request):
    articles = Article.objects.filter(publie=True)
    paginator = Paginator(articles, 10)  # 10 articles par page

    page_number = request.GET.get('page')
    page_obj = paginator.get_page(page_number)

    return render(request, 'articles/liste.html', {'page_obj': page_obj})
```

Dans votre template :

```
{% for article in page_obj %}
    <!-- Afficher chaque article -->
{% endfor %}

<!-- Liens de pagination -->
<div class="pagination">
    {% if page_obj.has_previous %}
        <a href="?page=1">Première</a>
        <a href="?page={{ page_obj.previous_page_number }}">Précédente</a>
    {% endif %}

  <span>Page {{ page_obj.number }} sur {{ page_obj.paginator.num_pages }}</span>

    {% if page_obj.has_next %}
        <a href="?page={{ page_obj.next_page_number }}">Suivante</a>
        <a href="?page={{ page_obj.paginator.num_pages }}">Dernière</a>
    {% endif %}
</div>
```

2.12 MIDDLEWARE ; LE SYSTÈME DE FILTRES

2.12.1 Qu'est-ce qu'un middleware ?

Le middleware est une couche intermédiaire qui traite les requêtes et les réponses. C'est comme une série de filtres que traverse chaque requête.

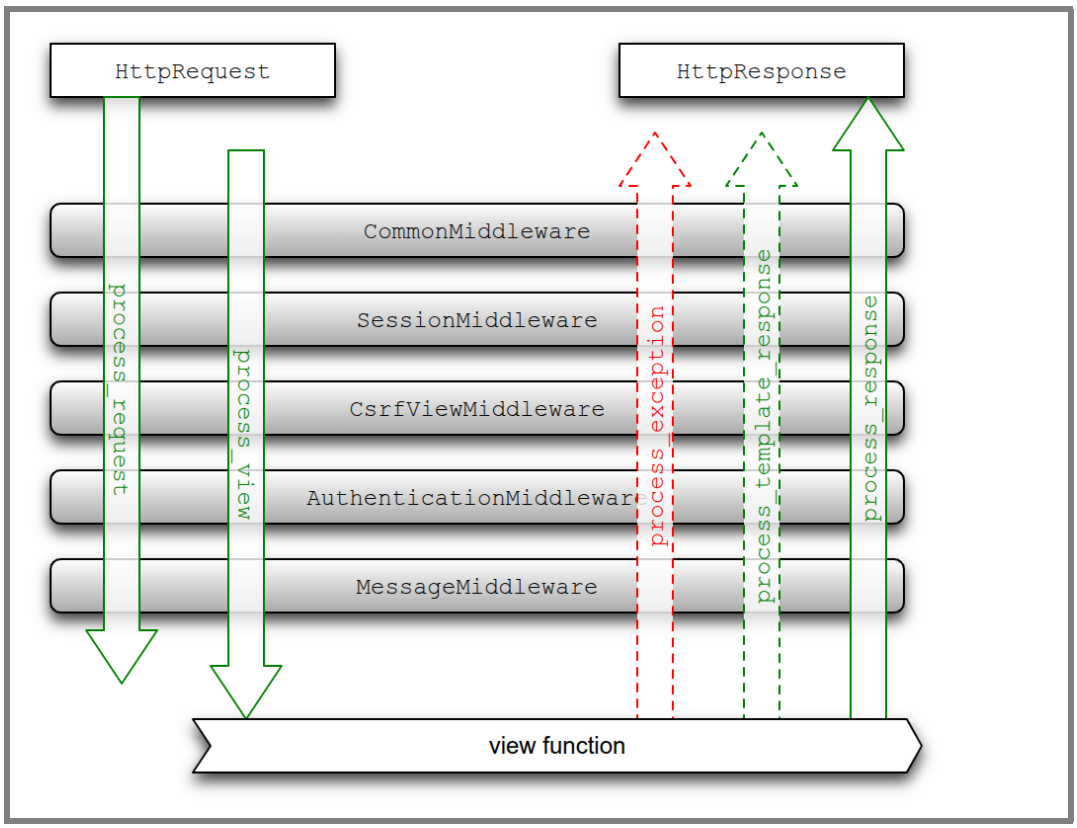
2.12.2 Exemples de middleware intégrés

— AuthenticationMiddleware : Ajoute l'objet `user` à chaque requête.
— SecurityMiddleware : Ajoute des en-têtes de sécurité.
— CsrfViewMiddleware : Vérifie les tokens CSRF.
— SessionMiddleware : Gère les sessions utilisateur.

2.12.3 Comment ça fonctionne ?

Quand une requête arrive :

1. Elle passe par tous les middleware (dans l'ordre).
2. Elle arrive à la vue.
3. La réponse repasse par tous les middleware (dans l'ordre inverse).



2.12.4 Créer son propre middleware

Un middleware simple pour logger les requêtes :

```
class LoggingMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Code exécuté avant la vue
        print(f"Requête reçue: {request.path}")

        response = self.get_response(request)

        # Code exécuté après la vue
        print(f"Réponse envoyée avec statut: {response.status_code}")

        return response
```

Ajoutez-le ensuite dans `settings.py` :

```
MIDDLEWARE = [
    # ...
    'monapp.middleware.LoggingMiddleware',
```

```
    # ...
]
```

3 CHAPITRE 3 : DJANGO REST FRAMEWORK, CRÉER DES APIS RESTFUL

3.1 POURQUOI CRÉER UNE API ?

Imaginez que vous développez une application mobile pour votre site web. L'application mobile ne peut pas exécuter le code Python de Django directement. Elle a besoin d'un moyen pour :

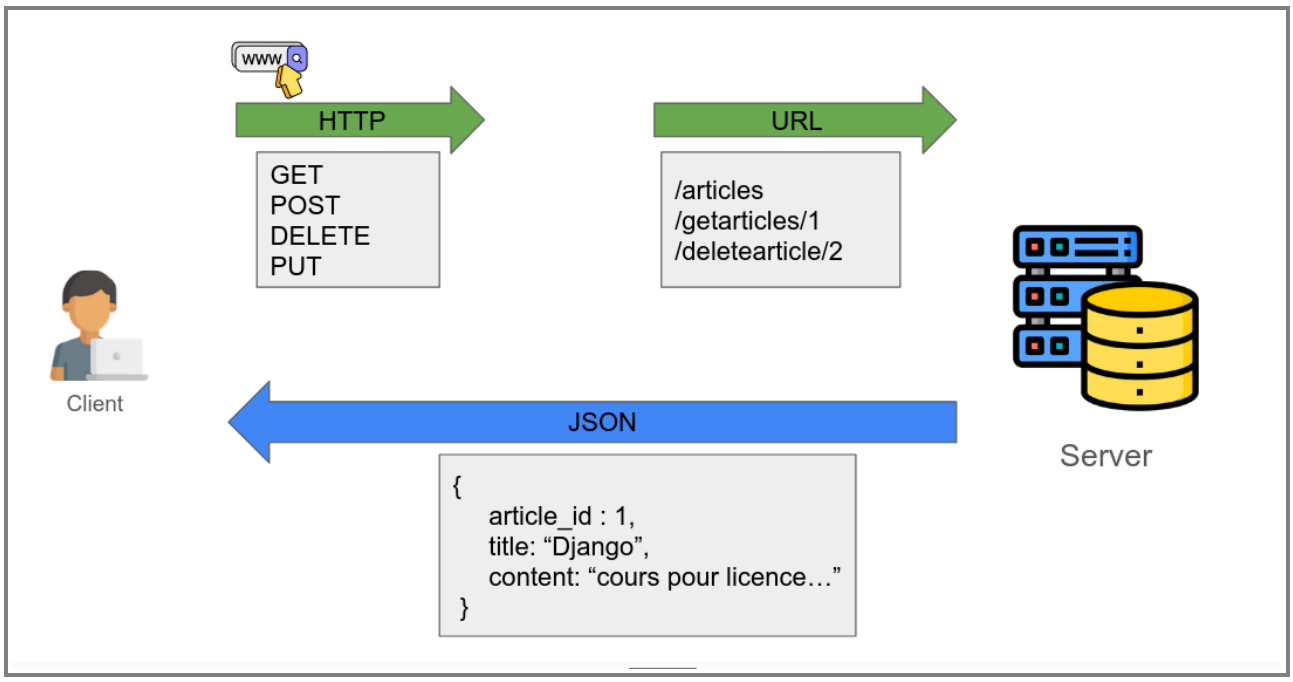
— Récupérer des données depuis votre base de données.
— Envoyer de nouvelles données (comme un nouvel article ou un commentaire).
— Mettre à jour ou supprimer des données existantes.

Une API (Application Programming Interface) est comme un serveur dans un restaurant. Vous (l'application mobile) donnez une commande, et le serveur (l'API) va chercher ce que vous avez demandé dans la cuisine (la base de données) et vous le rapporte.

3.2 QU'EST-CE QUE REST ?

REST (Representational State Transfer) est un style d'architecture pour les APIs. C'est une série de règles que suivent beaucoup d'APIs modernes :

1. Utilise HTTP proprement : Chaque type d'action correspond à une méthode HTTP :
   — `GET` : Récupérer des données
   — `POST` : Créer de nouvelles données
   — `PUT` : Mettre à jour des données existantes
   — `DELETE` : Supprimer des données
2. Sans état (stateless) : Chaque requête contient toute l'information nécessaire. Le serveur ne garde pas de mémoire de l'état du client entre les requêtes.
3. Ressources identifiables par URL : Chaque élément (utilisateur, article, commentaire) a sa propre URL.



3.3 POURQUOI DJANGO REST FRAMEWORK (DRF) ?

DRF est à l'API ce que Django est au site web traditionnel. C'est une boîte à outils complète pour créer des APIs RESTful avec Django.

Sans DRF, créer une API simple pour gérer des articles ressemblerait à :

```
from django.http import JsonResponse
from .models import Article
import json

def api_articles(request):
    if request.method == 'GET':
        articles = Article.objects.all()
        data = []
        for article in articles:
            data.append({
                'id': article.id,
                'titre': article.titre,
                'contenu': article.contenu
            })
        return JsonResponse(data, safe=False)

    elif request.method == 'POST':
        data = json.loads(request.body)
        article = Article.objects.create(
            titre=data['titre'],
```

```
            contenu=data['contenu']
        )
        return JsonResponse({'id': article.id}, status=201)
```

Avec DRF, le même code devient :

```
from rest_framework import serializers, viewsets
from .models import Article

class ArticleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Article
        fields = '__all__'

class ArticleViewSet(viewsets.ModelViewSet):
    queryset = Article.objects.all()
    serializer_class = ArticleSerializer
```

Et DRF génère automatiquement toutes les URLs nécessaires !

3.4 LES COMPOSANTS PRINCIPAUX DE DRF

3.4.1 Les Serializers

Les serializers sont comme les formulaires de Django, mais pour les données JSON. Ils font deux choses :

— **Sérialisation** : Convertir un objet Python (comme un modèle Article) en JSON.
— **Désérialisation** : Convertir du JSON en objet Python.

Exemple de serializer :

```
from rest_framework import serializers
from .models import Article, Commentaire

class CommentaireSerializer(serializers.ModelSerializer):
    auteur = serializers.ReadOnlyField(source='auteur.username')

    class Meta:
        model = Commentaire
        fields = ['id', 'contenu', 'auteur', 'date_creation']

class ArticleSerializer(serializers.ModelSerializer):
    # Relation avec les commentaires
```

```
    commentaires = CommentaireSerializer(many=True, read_only=True)

    # Champ calculé
    nombre_commentaires = serializers.SerializerMethodField()

    class Meta:
        model = Article
        fields = ['id', 'titre', 'contenu', 'date_publication',
                  'auteur', 'commentaires', 'nombre_commentaires']

    def get_nombre_commentaires(self, obj):
        return obj.commentaires.count()

    def validate_titre(self, value):
        """Validation personnalisée"""
        if len(value) < 10:
            raise serializers.ValidationError(
                "Le titre doit faire au moins 10 caractères"
            )
        return value
```

3.4.2 Les Views (vues) de l'API

DRF offre plusieurs types de vues, du plus simple au plus complexe :

3.4.2.1 APIView (la plus basique)

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .models import Article
from .serializers import ArticleSerializer

class ListeArticles(APIView):
    def get(self, request):
        articles = Article.objects.all()
        serializer = ArticleSerializer(articles, many=True)
        return Response(serializer.data)

    def post(self, request):
        serializer = ArticleSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
```

```
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

3.4.2.2 ViewSets (le plus puissant)

Les ViewSets combinent plusieurs actions dans une seule classe :

```
from rest_framework import viewsets, permissions, filters
from rest_framework.decorators import action
from rest_framework.response import Response
from .models import Article
from .serializers import ArticleSerializer

class ArticleViewSet(viewsets.ModelViewSet):
    queryset = Article.objects.all()
    serializer_class = ArticleSerializer
    permission_classes = [permissions.IsAuthenticatedOrReadOnly]
    filter_backends = [filters.SearchFilter, filters.OrderingFilter]
    search_fields = ['titre', 'contenu']
    ordering_fields = ['date_publication', 'titre']

    # Action personnalisée: /api/articles/{id}/publier/
    @action(detail=True, methods=['post'])
    def publier(self, request, pk=None):
        article = self.get_object()
        article.publie = True
        article.save()
        return Response({'status': 'article publié'})

    # Action sur la collection: /api/articles/articles_recents/
    @action(detail=False)
    def articles_recents(self, request):
        articles = Article.objects.filter(
            date_publication__gte='2023-01-01'
        )[:10]
        serializer = self.get_serializer(articles, many=True)
        return Response(serializer.data)
```

3.4.3 Les Routers

Les routers génèrent automatiquement les URLs pour vos ViewSets :

```
from rest_framework.routers import DefaultRouter
from .views import ArticleViewSet, AuteurViewSet

router = DefaultRouter()
router.register(r'articles', ArticleViewSet)
router.register(r'auteurs', AuteurViewSet)

urlpatterns = router.urls
```

Cela crée automatiquement :

- `GET /api/articles/` : Liste tous les articles.
- `POST /api/articles/` : Crée un nouvel article.
- `GET /api/articles/{id}/` : Détail d'un article.
- `PUT /api/articles/{id}/` : Met à jour un article.
- `DELETE /api/articles/{id}/` : Supprime un article.
- `POST /api/articles/{id}/publier/` : Notre action personnalisée.

4 CHAPITRE 4 : TESTS AVANCÉS ; TESTS D'INTÉGRATION ET TESTS DE PERFORMANCE

4.1 POURQUOI TESTER SON CODE ?

Imaginez que vous construisez une chaise. Vous pourriez la construire sans la tester, et espérer qu'elle supporte le poids ; ou bien la construire, puis demander à quelqu'un de s'asseoir dessus pour voir si elle tient.

Les tests automatisés, c'est comme avoir une machine qui teste automatiquement chaque chaise que vous fabriquez. Si vous changez quelque chose (un nouveau type de vis, un autre bois), la machine reteste immédiatement pour s'assurer que la chaise tient toujours.

4.2 LES DIFFÉRENTS TYPES DE TESTS

4.2.1 Tests unitaires

Testent une petite unité de code en isolation (une fonction, une méthode).

Exemple : Tester une fonction qui calcule le total d'une commande.

```
def calculer_total(articles):
    return sum(article.prix for article in articles)

# Test unitaire
def test_calculer_total():
```

```
    articles = [Article(prix=10), Article(prix=20)]
    assert calculer_total(articles) == 30
```

4.2.2 Tests d'intégration

Testent comment différentes parties du système fonctionnent ensemble.

Exemple : Tester qu'un utilisateur peut se connecter, créer un article, et le voir apparaître sur le site.

4.2.3 Tests de performance

Testent si l'application est assez rapide sous charge.

Exemple : Vérifier que la page d'accueil se charge en moins de 2 secondes même avec 10 000 utilisateurs connectés.

4.3 EXÉCUTER LES TESTS

```
# Tous les tests
python manage.py test

# Tests d'une application spécifique
python manage.py test monapp

# Tests d'une classe spécifique
python manage.py test monapp.tests.BlogIntegrationTest

# Tests d'une méthode spécifique
python manage.py test monapp.tests.BlogIntegrationTest.test_flux_complet_visiteur

# Avec verbosité
python manage.py test -v 2

# Garder la base de données de test entre les runs (plus rapide)
python manage.py test --keepdb
```

Les tests ne sont pas un luxe, mais une nécessité. Ils vous permettent de :

1. Dormir tranquille : Savoir que vos modifications ne cassent rien.
2. Refactoriser en confiance : Changer le code sans peur.
3. Documenter le comportement : Les tests montrent comment le code est censé fonctionner.
4. Détecter les régressions : Quand un bug revient, vos tests le détectent.
5. Améliorer la qualité : Écrire des tests vous force à écrire du code plus modulaire et testable.

Commencez par tester les fonctionnalités critiques, puis étendez progressivement votre couverture de tests. Un bon objectif est d'atteindre 80% de couverture de code.

Une application bien testée est une application qui dure dans le temps. Les tests sont votre filet de sécurité quand vous modifiez du code, et votre assurance qualité avant de déployer en production.

4.4 CONCLUSION ET PERSPECTIVES

Django est plus qu'un outil technique. C'est une philosophie de développement qui vous apprend à structurer vos pensées, à prioriser la sécurité et à écrire du code maintenable. Ces compétences sont transférables à n'importe quel autre framework ou langage.

4.4.1 Ressources pour continuer

Documentation officielle : https://docs.djangoproject.com/fr/