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