

Framework **django**

Support de cours/TD Framework django

Classes: CII-3-GLSI

Année universitaire 2023/2024

Réalisé et enseigné par: Nizar MAATOUG

Aperçu du contenu (1 / 2)

- ▶ Framework django: Big Picture
- ▶ Framework django: Architecture
- ▶ Installation des outils, premier projet django
- ▶ Anatomie, concepts Projet/Application django
- ▶ Couche Modèle:
 - ▶ Concept ORM
 - ▶ Modèle: définition, manipulation, relationships
 - ▶ Configuration de la base de données
- ▶ Interface Admin: découverte, publier, manipuler des modèles
- ▶ Couche View: définir des actions: FBV (Function Based Views), CBV (Class Based Views)
- ▶ django templates
- ▶ django Forms

Aperçu du contenu (2/2)

- ▶ Sécurité: Authentification, rôles, permissions, groupes.
- ▶ Application: développer une application web avec le framework **django**

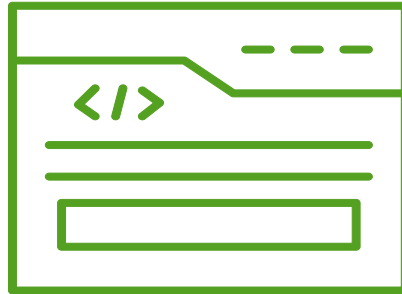
objectifs

- ▶ Maîtriser l'architecture du framework **django**
- ▶ Développer et tester une application web **django**

Prérequis



Framework django



Développement web
HTML, CSS, JavaScript



Base de données
Concepts de base
Table, rows
Insert, update

django

Framework web Python

Développer rapidement des applications web

Avec le minimum de code

Développé entre 2003 et 2005

Philosophie: Piles incluses

ORM

Templates

Forms

Admin

URL Mapping

Packages

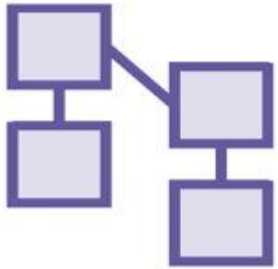
Polyvalent

- ▶ Django peut être (et a été) utilisé pour créer presque tous les genres de sites:
 - ▶ Site d'actualité
 - ▶ Gestionnaire de données
 - ▶ Wikis
 - ▶ Réseaux sociaux
- ▶ Qui utilise django ?
 - ▶ Instagram
 - ▶ Spotify
 - ▶ Youtube
 - ▶ Dropbox
 - ▶ ...

Autres

- ▶ Sécurisé
- ▶ Maintenable
- ▶ Scalable
- ▶ Portable

django: Architecture MVT



Model

Représente les données,
Assure le mapping
Objet/Relationnel

MVC: "Controller"



View

Reçoit une requête http,
effectue un traitement,
retourne une réponse http

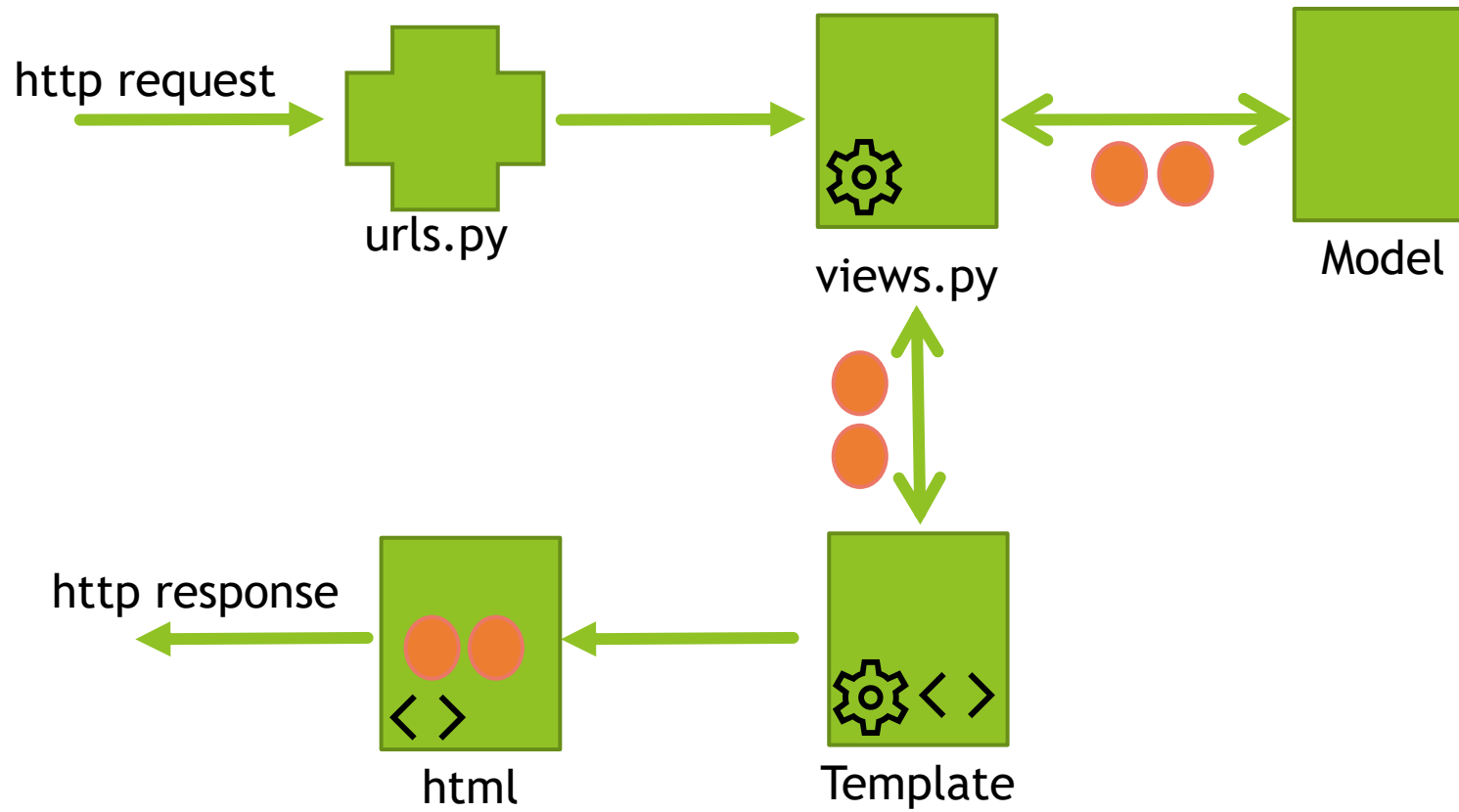
MVC: "View"



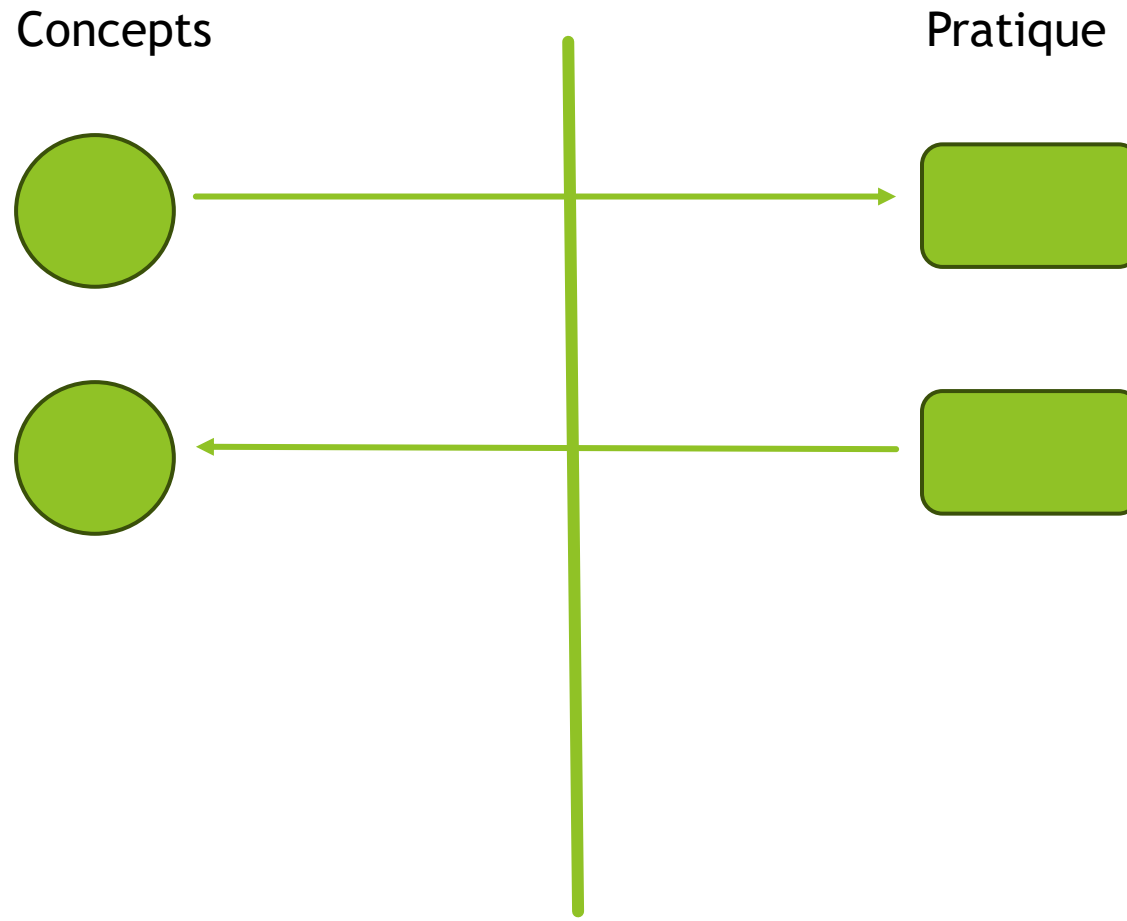
Template

Définit, génère la
présentation HTML

django: Architecture MVT

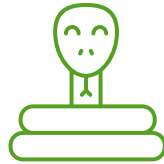


Approche





Système d'exploitation: windows, Linux, Mac OS



Python: Latest version 3.11



Éditeur de code: VS code, PyCharm

django: Première application

Étape par Étape

- ▶ Installation et configuration des outils
- ▶ Création et activation d'un environnement virtuel pour le projet
- ▶ Installation du framework django (virtual env activé)
- ▶ Sauvegarder les dépendances
- ▶ Création du projet django nommé **hello_word_project**
- ▶ Création de l'application **pages**
- ▶ Déclaration de l'application **pages**
- ▶ Première page web

Installation et configuration des outils

- ▶ Installer Python: <https://www.python.org/downloads/>
- ▶ Installer VS Code ou PyCharm

Environnement virtuel

Étape par Étape



- ▶ Ne pas installer les packages python globalement
- ▶ Toujours travailler dans un environnement virtuel
- ▶ Éviter les conflits de dépendances
- ▶ Travailler dans un contexte isolé.

Environnement virtuel

Étape par Étape

#créer un dossier pour le projet

```
$ cd framework_django\projects
```

```
$ mkdir helloworld
```

```
$ cd helloworld
```

#créer et activer l'environnement virtuel

```
$ python -m venv .venv
```

```
$ .venv\Scripts\Activate.ps1
```

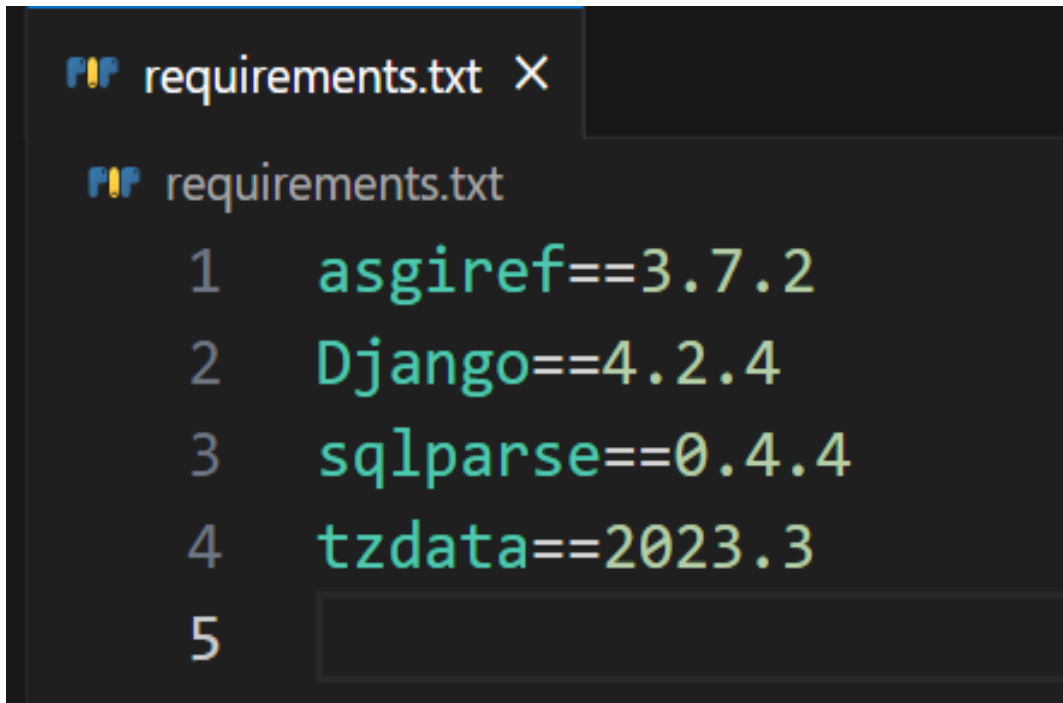
```
(.venv) $ python -m pip install django
```

```
$ source .venv/bin/activate
```

Sauvegarder les dépendances

#sauvegarder les dépendances

(.venv) \$ pip freeze > requirements.txt



```
requirements.txt X
requirements.txt
1  asgiref==3.7.2
2  Django==4.2.4
3  sqlparse==0.4.4
4  tzdata==2023.3
5
```

Créer le projet django_project

Étape par Étape

```
# créer le projet django_project
```

```
(.venv) $ django-admin startproject django_project .
```

```
# lancer le projet dans VS code
```


```
(.venv) $ code .
```

django_project: Anatomie


Étape par Étape

✓  django_project

Marque le répertoire comme package => pouvoir importer les composants


>  __pycache__

Standard Python pour application et serveur asynchrone

 __init__.py

 asgi.py

Contrôle l'application django via des configurations


 settings.py

Binding entre requête http et traitement associé

 urls.py

Web Server Gateway Interface

 wsgi.py

 manage.py

Ne fait pas partie du projet, mais utile pour exécuter des commandes: runserver, créer nouvelle application,...

Lancer le projet

démarrer le projet

(.venv) \$ python manage.py runserver

appliquer les migrations

(.venv) \$ python manage.py migrate

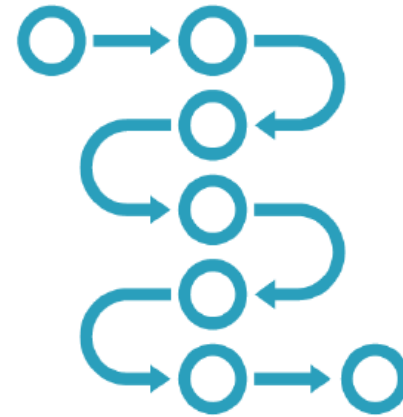
Étape par Étape

Migrations



Models

Classes Python
Associées aux tables de
la BD



Migrations

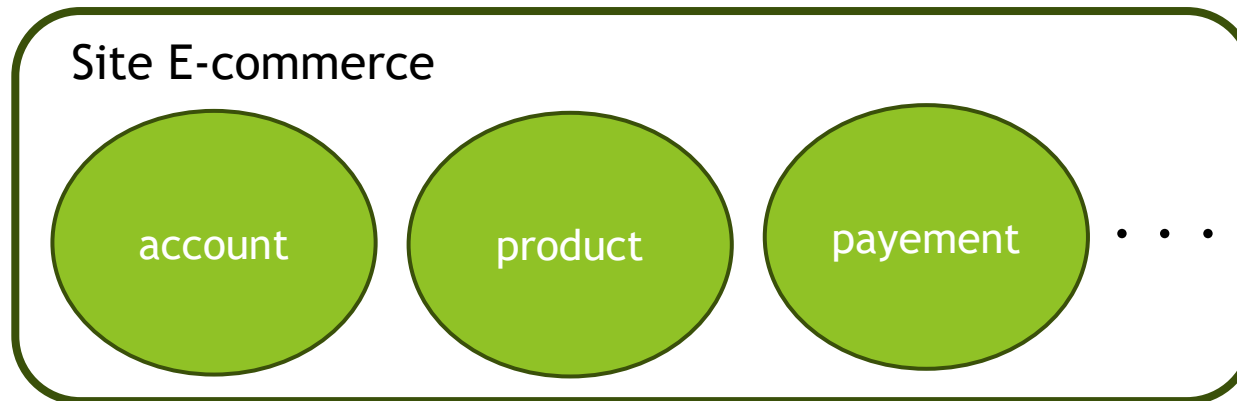
Ensemble de scripts
décrivant le schéma de
la BD

Étape par Étape

Application django: pages

Étape par Étape

- ▶ django utilise les concepts **projet** et **application** pour maintenir le code "clean"
- ▶ un projet django peut contenir plusieurs applications
- ▶ Chaque application contrôle une fonctionnalité isolée du projet.



Application django: pages

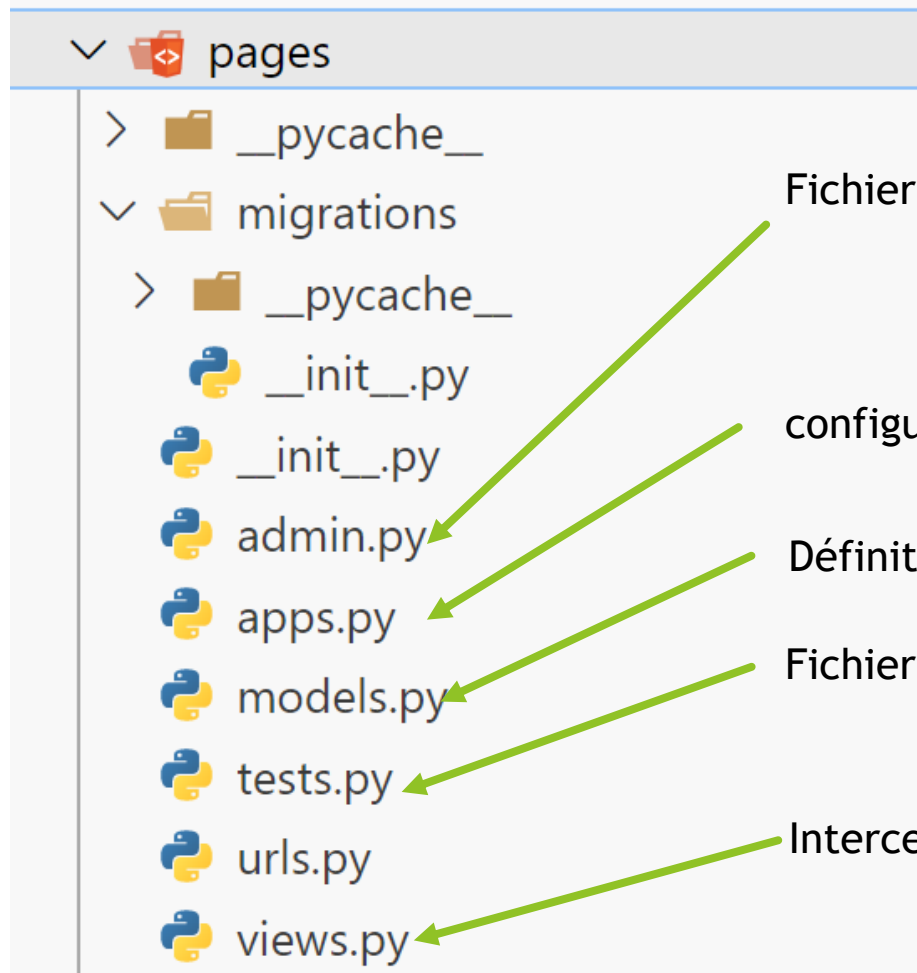
| # créer l'application pages

| (.venv) \$ python manage.py startapp pages

Étape par Étape

Application django: Anatomie

Étape par Étape



Fichier de configuration pour le module Admin-app

configuration de l'application pages

Définition du modèle

Fichier dans lequel on code les tests

Intercepter les requêtes, traitement, retourner une réponse

Déclaration de l'application pages

Étape par Étape

Application definition

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

Première page web: views.py

Étape par Étape

```

views.py ×
pages > views.py > ...
1  from django.shortcuts import render
2  from django.http import HttpResponse
3
4  # Create your views here.
5
6  def homePageView(request):
7      return HttpResponse("Hello word")
8

```

Première page web: pages.urls.py

Étape par Étape

```

urls.py  ×
pages > urls.py > ...
1  from django.urls import path
2
3  from .views import homePageView
4
5
6  urlpatterns=[
7      path('',homePageView, name="home"),
8  ]

```

Première page web: urls.py

Étape par Étape

```

urls.py  ×
django_project > urls.py > ...

17  from django.contrib import admin
18  from django.urls import path,include
19
20  urlpatterns = [
21      path('admin/', admin.site.urls),
22      path('',include("pages.urls")),
23  ]
24

```



- ▶ Django: framework web Python
- ▶ Développement web simple et rapide
- ▶ Piles incluses: ORM, Admin module, ...
- ▶ Architecture MVT (Model, View, Template)
- ▶ Un projet django, plusieurs applications
- ▶ Virtual environnement

Application: Meeting Planner

Résultats

- ▶ Maîtriser l'architecture MVT
- ▶ Découvrir les concepts de base de django:
 - ▶ Model
 - ▶ View
 - ▶ Template
 - ▶ URLs
 - ▶ Admin panel
 - ▶ Forms

Meeting Planner

- ▶ Permet de planifier les réunions
- ▶ Planifier une **réunion**:
 - ▶ Définir la date, la durée
 - ▶ affecter une **salle** à cette réunion

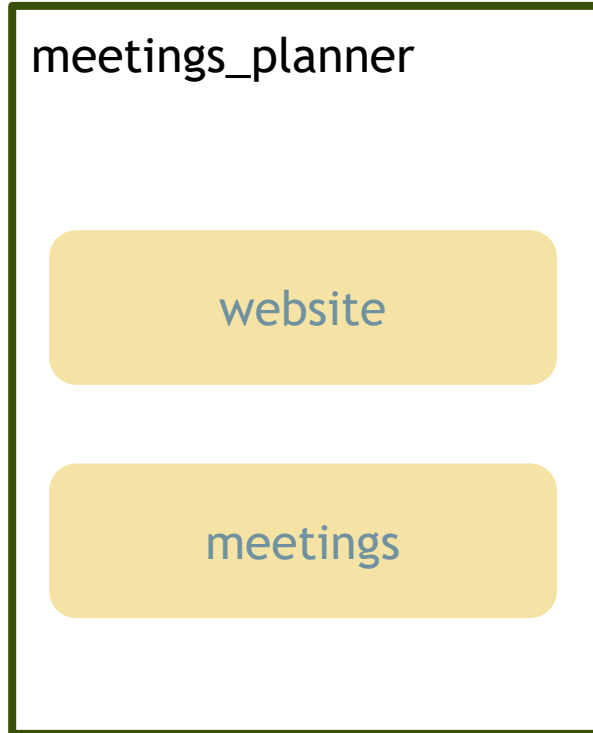
Démo de l'application

Démarche

- ▶ Créer le projet **meetings_planner**
 - ▶ Créer et activer l'environnement virtuel
 - ▶ Installer les dépendances
 - ▶ Créer le projet et les applications django
- ▶ Implémenter les pages web (accueil, about)
- ▶ Implémenter le modèle **Meeting**
- ▶ Admin Panel:
 - ▶ Créer un **superuser**
 - ▶ Gérer le modèle **Meeting**
- ▶ Gérer les **Meeting**
- ▶ implémenter le modèle Room
- ▶ Définir l'association Meeting-----Room

Étape par Étape

Architecture du projet



django apps



Python package

Contient models, views, templates, urls

Les projets django contiennent plusieurs apps

Les Apps peuvent être réutilisées

Maintenir les apps petites et simples

Création du projet

Étape par Étape

#créer un dossier pour le projet

```
$ cd framework_django\projects
```

```
$ mkdir meetings_planner
```

```
$ cd meetings_planner
```

#créer et activer l'environnement virtuel

```
$ python -m venv my_env
```

```
$ my_env\Scripts\Activate
```

```
(my_env) $ python -m pip install django
```

Sauvegarder les dépendances

#sauvegarder les dépendances

(my_env) \$ pip freeze > requirements.txt

```
requirements.txt X
requirements.txt
1  asgiref==3.7.2
2  Django==4.2.4
3  sqlparse==0.4.4
4  tzdata==2023.3
5
```

Création projet et applications

créer le projet django_project

(my_env) \$ django-admin startproject meetings_planner .

lancer le projet dans VS code

(my_env) \$ code .

créer l'application website

(my_env) \$ python manage.py startapp website









créer l'application meetings

(my_env) \$ python manage.py startapp meetings

Étape par Étape

Structure du projet

✓ MEETINGS_PLANNER

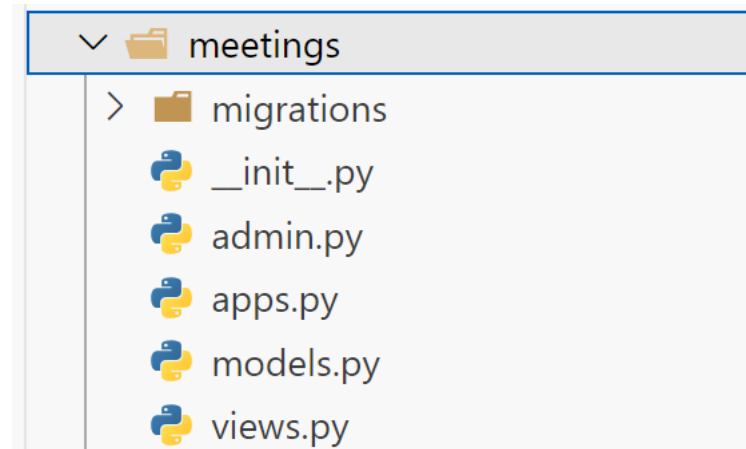
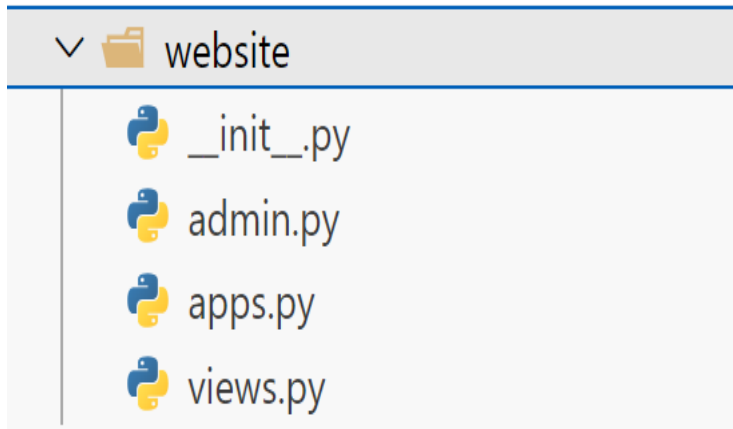
- >  meetings
- >  meetings_planner
- >  my_env
- >  website
-  .gitignore
-  db.sqlite3
-  manage.py
-  requirements.txt

Framework django

Étape par Étape

Structure du projet

Étape par Étape



Settings.py: Déclaration des applications

Étape par Étape

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

Application des migrations

Étape par Étape

appliquer les migrations

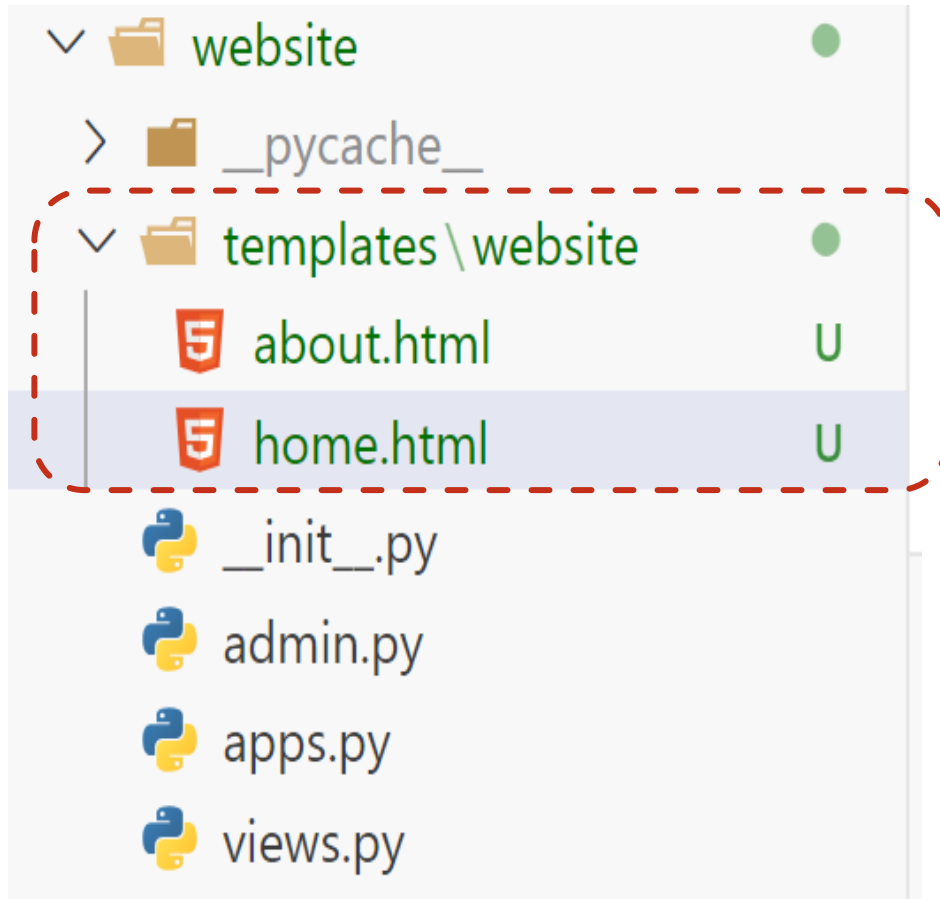
(my_env) \$ python manage.py migrate

démarrer le projet

(my_env) \$ python manage.py runserver

website: home.html & about.html

Étape par Étape



website: home.html

Étape par Étape

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Meeting planer: home page</title>
</head>
<body>
  <h2>Meeting planner application from Tek-up</h2>
  <h3>Lite des réunions</h3>
</body>
</html>
```

website: about.html

Étape par Étape

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Meeting planer: about page</title>
</head>
<body>
  <h2>Meeting planner application from Tek-up</h2>

</body>
</html>
```

website: views.py

```
from django.shortcuts import render
```

```
# Create your views here.
```

```
def home_view(request):  
    return render(request, "website/home.html")
```

```
def about_view(request):  
    return render(request, "website/about.html")
```

Étape par Étape

Étape par Étape

website: urls.py

```
from django.urls import path

from . import views

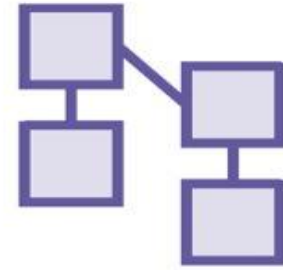
#domain.com/website/...
urlpatterns=[
    path('',views.home_view, name='home'),
    path('about',views.about_view, name='about'),
]
```

meetings_planner: urls.py

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('website/', include('website.urls')), #new
]
```

Créer le modèle



Django Models

Enregistrer les objets Python dans la BD
Les classes Models sont mappés à des tables
Les attributs sont mappés à des colonnes
SQL est généré
Create/Update tables (migrations)
Insert/Update/Delete lignes (admin)

Models

- Documentation: <https://docs.djangoproject.com/fr/4.2/topics/db/models/>

Démarche: Models

- ▶ Créer les classes modèles
- ▶ Créer les migrations
- ▶ appliquer les migrations
- ▶ Gérer le modèle avec Admin-interface

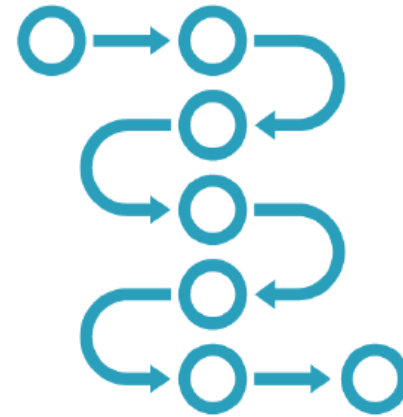
Étape par Étape

Migrations



Models

Classes Python
Associées aux tables de
la BD



Migrations

Ensemble de scripts
décrivant le schéma de
la BD

Étape par Étape

Show migrations

Étape par Étape

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
(.venv) PS C:\pc-nizar\Tek-up\AU2023-2024-S1\django\projects\meeting> python .\manage.py showmigrations
admin
[ ] 0001_initial
[ ] 0002_logentry_remove_auto_add
[ ] 0003_logentry_add_action_flag_choices
auth
[ ] 0001_initial
[ ] 0002_alter_permission_name_max_length
[ ] 0003_alter_user_email_max_length
[ ] 0004_alter_user_username_opts
[ ] 0005_alter_user_last_login_null
[ ] 0006_require_contenttypes_0002
[ ] 0007_alter_validators_add_error_messages
[ ] 0008_alter_user_username_max_length
[ ] 0009_alter_user_last_name_max_length
[ ] 0010_alter_group_name_max_length
[ ] 0011_update_proxy_permissions
[ ] 0012_alter_user_first_name_max_length
contenttypes
[ ] 0001_initial
[ ] 0002_remove_content_type_name
sessions
[ ] 0001_initial
website
```

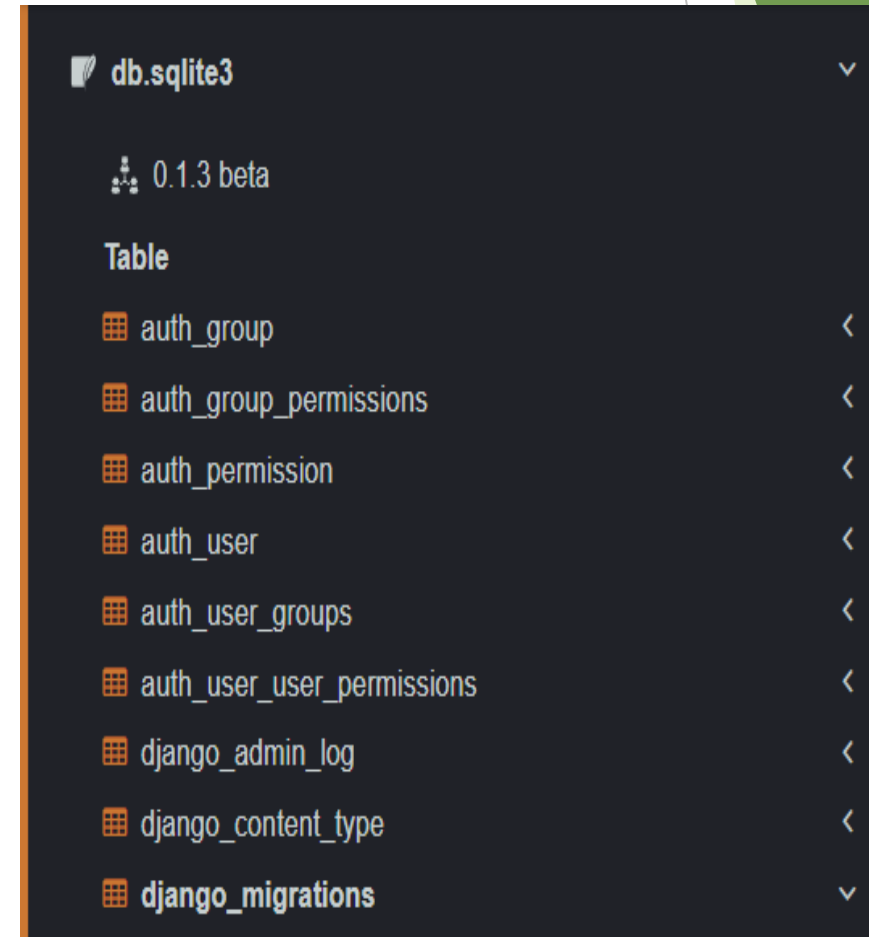

Appliquer les migrations initiales

Étape par Étape

appliquer les migrations

(my_env) \$ python manage.py migrate

Afficher le schéma de la BD: <https://sqliteonline.com/>



Implémenter la classe modèle Meeting

Étape par Étape

meetings.models.py

```
from django.db import models

# Create your models here.

class Meeting(models.Model):
    title=models.CharField(max_length=200)
    date=models.DateField()
```

Étape par Étape

Créer et appliquer la migration

créer la migration

(my_env) \$ python manage.py makemigrations

afficher SQL de la migration

(my_env) \$ python manage.py sqlmigrate meetings 0001

appliquer la migration

(my_env) \$ python manage.py migrate meetings

<https://sqliteonline.com/>

Migration Workflow

Step 1: Change Model code

Step 2: Generate migration script (check it!)

```
python manage.py makemigrations
```

Optional: Show migrations

```
python manage.py showmigrations
```

Optional: Show SQL for specific migration

```
python manage.py sqlmigrate appname migrationname
```

Step 3: Run migrations

```
python manage.py migrate
```

Django Admin

Deux parties de toute application web



End-users



Admin

django Admin

CRUD Model

Gestion Sécurité

Admin panel: éditer les modèles

Étape par Étape

- ▶ Enregistrer le modèle dans **admin.py**
- ▶ Créer un **superuser** pour pouvoir administrer l'application
- ▶ Démarrer l'application et connecter en tant que **superuser**
- ▶ Gérer le modèle

meetings.admin.py: enregistrer Meeting

Étape par Étape

```
from django.contrib import admin  
  
from .models import Meeting  
  
admin.site.register(Meeting)
```

Admin interface

Étape par Étape

127.0.0.1:8000/admin/login/?next=/admin/

Livre_ENI tekup Courrier - Nizar MA...

Django administration

Username:

Password:

Log in

Admin interface: créer superuser

Étape par Étape

```
# créer superuser
```

```
(my_env) $ python manage.py createsuperuser
```

Admin interface: ajouter des meetings

Étape par Étape

← → ↻ 🏠 ⓘ 127.0.0.1:8000/admin/meetings/meeting/add/

★ Bookmarks 📁 Livre_ENI 📁 tekup 📧 Courrier - Nizar MA...

Django administration

Home › Meetings › Meetings › Add meeting

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

MEETINGS

Meetings + Add

Add meeting

Title:

Date: Today | 📅

Note: You are 2 hours ahead of server time.

Admin interface: ajouter des meetings

Étape par Étape

← → ↻ 🏠 ⓘ 127.0.0.1:8000/admin/meetings/meeting/

★ Bookmarks 📁 Livre_ENI 📁 tekup 📧 Courrier - Nizar MA...

Django administration

Home > Meetings > Meetings

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

MEETINGS

Meetings + Add

Select meeting to change

Action: 0 of 2 selected

<input type="checkbox"/>	MEETING
<input type="checkbox"/>	Meeting object (2)
<input type="checkbox"/>	Meeting object (1)

2 meetings

Exercice: Mettre à jour Meeting

```
from django.db import models
from datetime import time
```

Create your models here.

```
class Meeting(models.Model):
    title=models.CharField(max_length=200)
    date=models.DateField()
    start_time=models.TimeField(default=time(9))
    duration=models.IntegerField(default=1)

    def __str__(self):
        return f"{self.title} at {self.start_time} on {self.date}"
```

Créer et appliquer la migration

Étape par Étape

Exercice: Ajouter le modèle Room

```
class Room(models.Model):
    name = models.CharField(max_length=50)
    floor = models.IntegerField()
    room_number = models.IntegerField()

    def __str__(self):
        return f"{self.name}: room {self.room_number} on floor {self.floor}"
```


Étape par Étape

Ajouter association one to many

```
class Meeting(models.Model):
    title = models.CharField(max_length=200)
    date = models.DateField()
    start_time = models.TimeField(default=time(9))
    duration = models.IntegerField(default=1)
    room = models.ForeignKey(Room, on_delete=models.CASCADE)

    def __str__(self):
        return f"{self.title} at {self.start_time} on {self.date}"
```

Créer et appliquer la migration

Étape par Étape

Étape par Étape

meetings.views

```
def detail(request, id):
    meeting = Meeting.objects.get(pk=id)
    return render(request, "meetings/detail.html", {"meeting": meeting})
```

```
def detail(request, id):
    meeting = get_object_or_404(Meeting, id)
    return render(request, "meetings/detail.html", {"meeting": meeting})
```

Étape par Étape

website.views

```
def home_view(request):
    context={'nbre_meeting': Meeting.objects.count()}
    return render(request, "website/home.html", context=context)

def about_view(request):
    return render(request, "website/about.html")
```

Templates

Générer les pages web

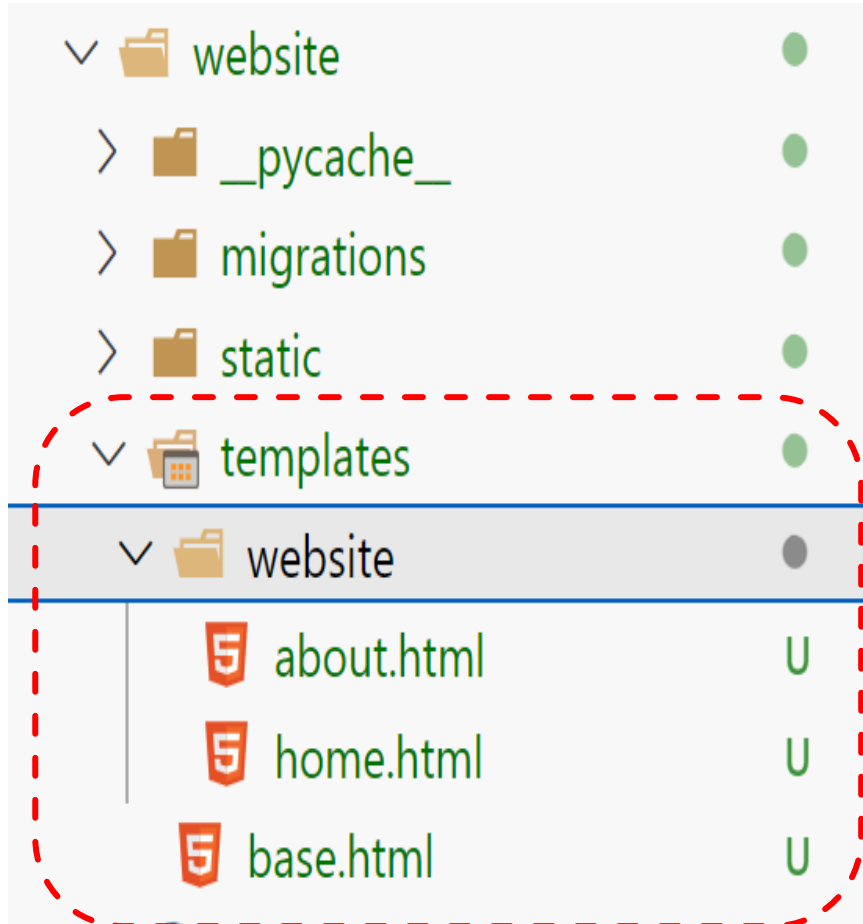
Template variables

Template tags

Template inheritance

Templates: héritage

Étape par Étape



Framework django

Étape par Étape

Templates: héritage (base.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

  <title>{% block title %}{% endblock %}</title>

</head>
<body>
  {% block content %}
  {% endblock %}
</body>
</html>
```

Templates: héritage (about.html)

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

{% block title %}about{% endblock %}

{% block content %}
    <h2>Meeting planner application from Tek-up</h2>

{% endblock %}
```


Étape par Étape

website: home.html v1

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

{% block title %}HOME{% endblock %}

{% block content %}
    <h2>Welcome to the meeting planner</h2>
    <p>there are currently {{nbre_meeting}} meetings</p>
{% endblock %}
```

Étape par Étape

website: home.html v2

```
<h2>Meetings</h2>
  <ul>
    {% for meeting in meetings %}
      <li>
        <a href="{% url 'detail' meeting.id %}">
          {{ meeting.title }}
        </a>
      </li>
    {% endfor %}
  </ul>
```

Étape par Étape

website: home.html v3

```
<h2>Meetings</h2>
  <ul>
    {% for meeting in meetings %}
      <li>
        <a href="{% url 'detail' meeting.id %}">
          {{ meeting.title }}
        </a>
      </li>
    {% endfor %}
  </ul>
  <a href="{% url 'rooms' %}">Rooms list</a>
```

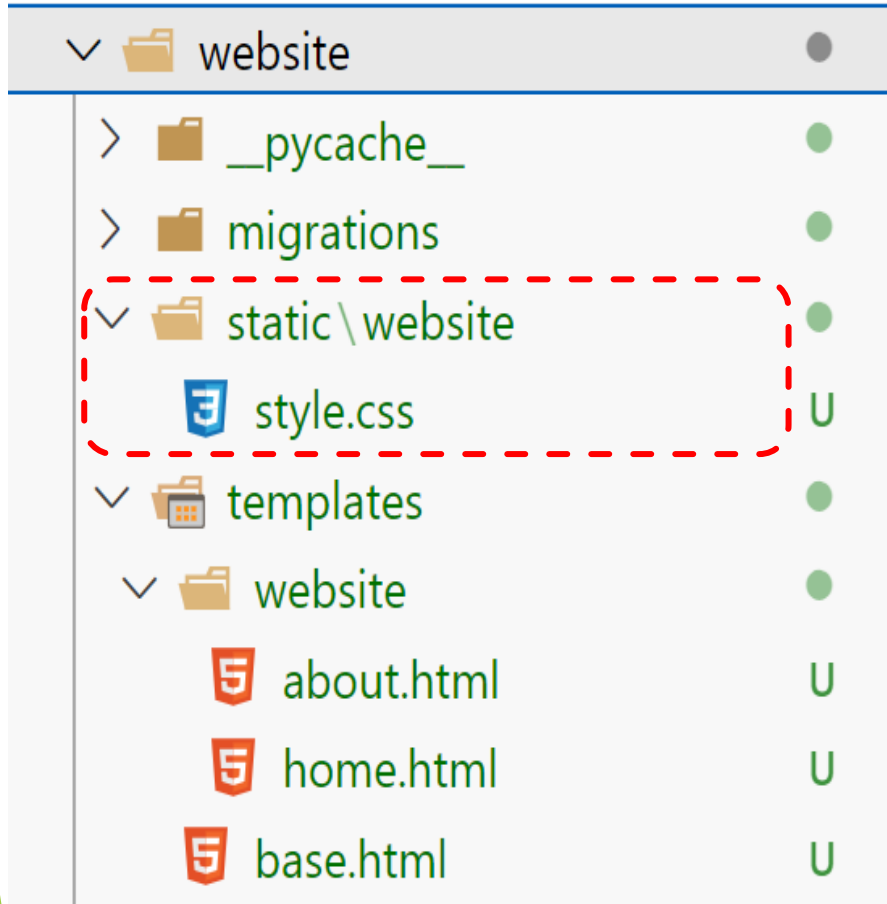
Étape par Étape

website.views v3

```
def home_view(request):
    context={'meetings': Meeting.objects.all()}
    return render(request, "website/home.html", context=context)
```

Templates: static files

Étape par Étape



```
body {
    font-family: sans-serif;
    color: cornflowerblue;
    background-color: floralwhite;
}
```

Étape par Étape

Templates: static files (settings.py)

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/4.2/howto/static-files/

STATIC_URL = 'static/'
```

Étape par Étape

Templates: static files (base.html)

```
{% load static %}

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}{% endblock %}</title>
    <link rel="stylesheet"
        href="{% static 'website/style.css' %}">
</head>
<body>
    {% block content %}
    {% endblock %}
</body>
</html>
```

Exercice

- ▶ Ajouter une page qui liste tous les rooms:
 - ▶ Views
 - ▶ template
 - ▶ url mapping

django forms

Valider les entrées utilisateurs

Automatise le code répétitif
sécurisé

Hautement personnalisé

Étape par Étape

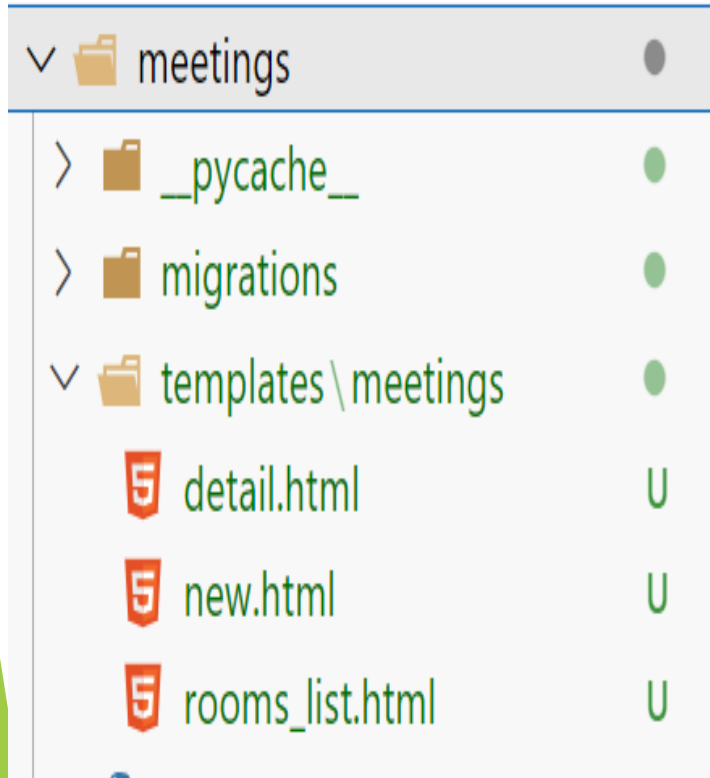
Formulaire: Meeting

```
class MeetingForm(ModelForm):
    class Meta:
        model = Meeting
        fields = '__all__'
        widgets = {
            'date': DateInput(attrs={"type": "date"}),
            'start': TimeInput(attrs={"type": "time"}),
            'duration': TextInput(attrs={"type": "number", "min": "1", "max": "4"})
        }

    def clean_date(self):
        d = self.cleaned_data.get("date")
        if d < date.today():
            raise ValidationError("Meetings cannot be in the past")
        return d
```

Formulaire: new.html

Étape par Étape



```
{% extends "base.html" %}
{% block title %}New Meeting{% endblock %}
```

```
{% block content %}
<h1>Plan a new meeting</h1>
<form method="post">
    <table>
        {{ form }}
    </table>
    {% csrf_token %}
    <button type="submit">Create</button>
</form>
{% endblock %}
```

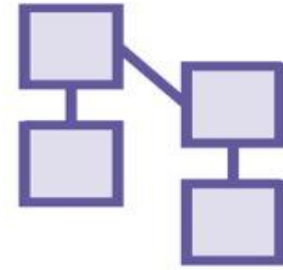
Exercice

- ▶ Ajouter tout le nécessaire pour permettre l'ajout d'une salle de réunion



- ▶ Architecture MVT
- ▶ Models: définition, manipulation, associations
- ▶ Admin-panel: CRUD, sécurité
- ▶ Templates: variables, tags, static files, héritage
- ▶ Forms: mapping entre templates et Models

django Models



Résultats

À la fin de cette séance, vous serez en mesure de:

- ▶ Maîtriser le composant **Models** dans l'architecture **MTV** de **django**
 - ▶ django ORM
 - ▶ django Models
 - ▶ Manipulation des modèles
 - ▶ Migrations, optimisation

Éléments du contenu

- ▶ Modéliser vos données
- ▶ Écrire des requêtes avancées
- ▶ Personnaliser le comportement de vos Models
- ▶ Créer et appliquer des migrations
- ▶ Optimiser les performances

Model-Template-View

Model
Données

Classes Modèle
Mapped to DB

Template
Présentation

Générer HTML

View
Comportement

Fonction python
Mapped to URL

Model

View

Controller

ORM

Object-Relational Mapping

Un autre nom de la couche **Models**

Relational data

- Lignes dans des tables de base de données
- SQL

Python data

- Classes et objets, valeurs et variables

ORM bridges the gaps

- les classes modèles sont mappés à des tables
- SQL est généré
- coder uniquement en python

ORM inconvenients

Moins de contrôle sur SQL

Moins de performances

Mais

- Possible de faire des optimisations avec django ORM
- Possible d'exécuter Raw SQL

django Models

Mapped to DB
tables

Générer UI
(ModelForm)

Valider
Forms

Générer
Admin interface

Ajouter
custom methods

django Models

Bases de données supportées:

- PostgreSQL, MariaDB, MySQL, Oracle, SQLite
- avec packages: DB2, MS SQL, ...

Projet de démonstrations

- ▶ Application web de e-commerce (gestion des produits)
- ▶ Focaliser le développement dans la couche Model
- ▶ Utiliser deux BD:
 - ▶ SQLite
 - ▶ PostgreSQL

Création du projet

Étape par Étape

#créer un dossier pour le projet

```
$ cd framework_django\projects\django_models
```

```
$ mkdir e_commerce
```

```
$ cd e_commerce
```

#créer et activer l'environnement virtuel

```
$ python -m venv my_env
```

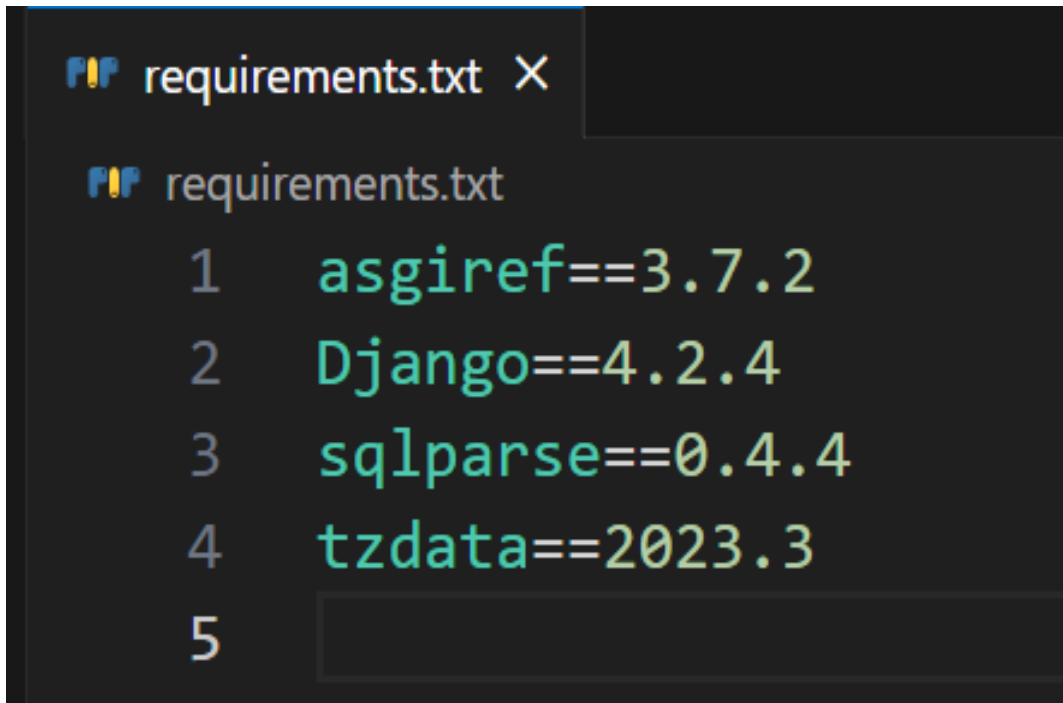
```
$ my_env\Scripts\Activate
```

```
(my_env) $ python -m pip install django
```

Sauvegarder les dépendances

#sauvegarder les dépendances

(my_env) \$ pip freeze > requirements.txt

A screenshot of a code editor window with a dark background. The title bar shows a file named 'requirements.txt' with a close button. The editor contains a list of dependencies, each on a new line, numbered 1 through 5. The dependencies are: asgiref==3.7.2, Django==4.2.4, sqlparse==0.4.4, tzdata==2023.3, and an empty line for number 5.

```
requirements.txt X
requirements.txt
1 asgiref==3.7.2
2 Django==4.2.4
3 sqlparse==0.4.4
4 tzdata==2023.3
5
```


Création du projet et applications

Étape par Étape

créer le projet e_commerce_site

(my_env) \$ django-admin startproject e_commerce_site .

lancer le projet dans VS code

(my_env) \$ code .

créer l'application products

(my_env) \$ python manage.py startapp products

settings.py: installer l'application

Étape par Étape

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'products.apps.ProductsConfig', #new
]
```

Installation de PostgreSQL

- <https://www.postgresql.org/download/>



settings.py: Configuration du Log

Étape par Étape

```
#logging all SQL
LOGGING={
    "version": 1,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        }
    },
    'loggers': {
        'django.db.backends': {
            'level': 'DEBUG',
            'handlers': ['console'],
        }
    }
}
```



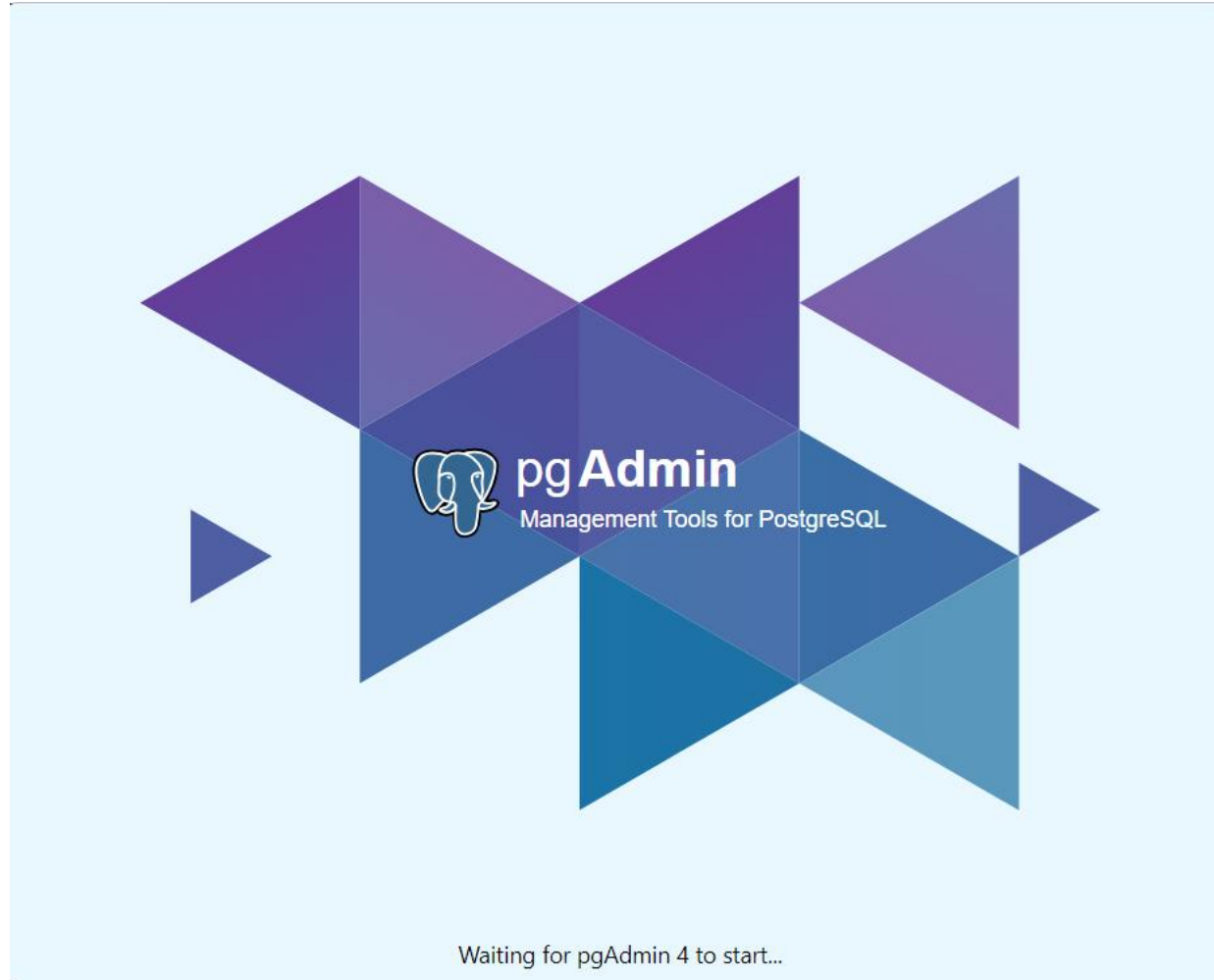
Étape par Étape

settings.py: configuration du

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'products_db',  
        'HOST': '127.0.0.1',  
        'USER': 'postgres',  
        'PASSWORD': 'adminadmin',  
    }  
}
```

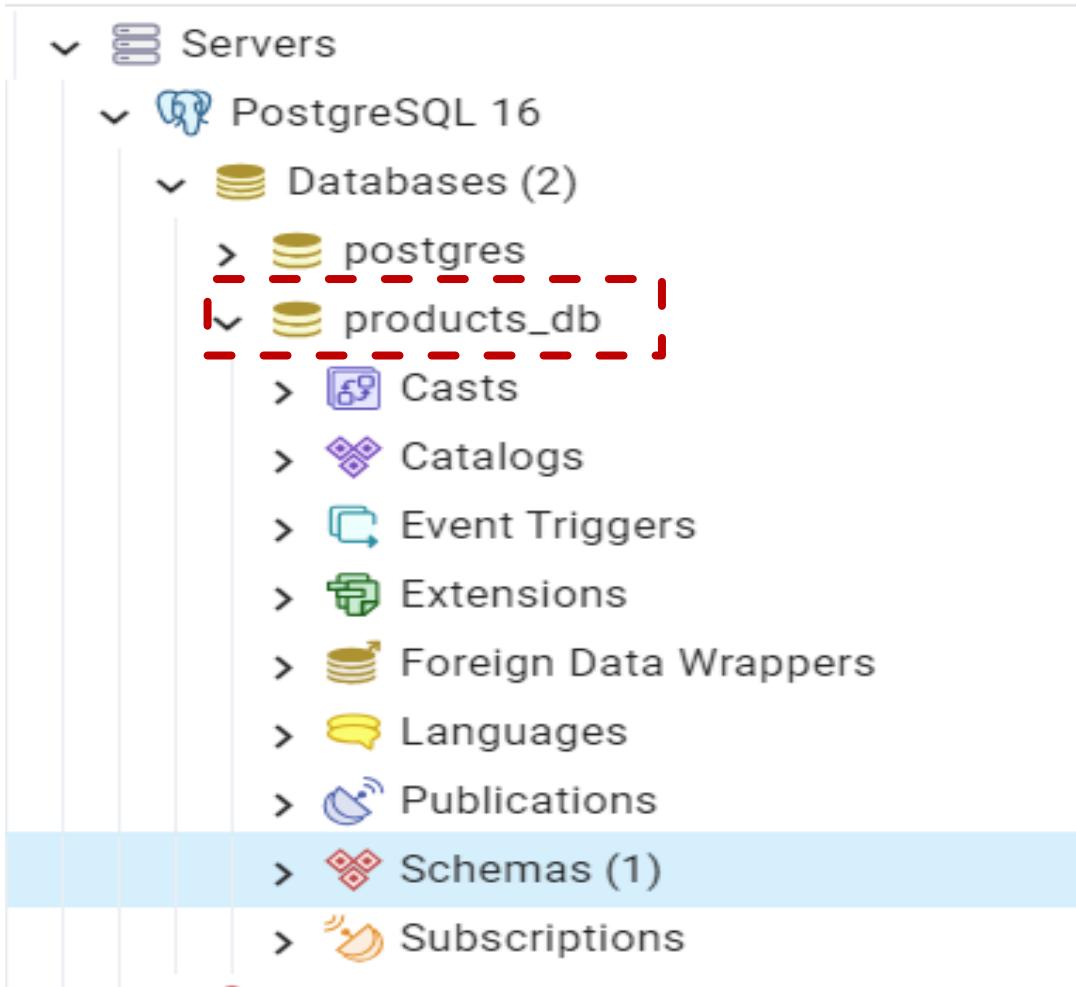
PostgreSQL: créer la base de données

Étape par Étape



PostgreSQL: créer la base de données

Étape par Étape



Installer psycopg2

```
python -m pip install psycopg2
```

```
pip freeze > requirements.txt
```

Étape par Étape

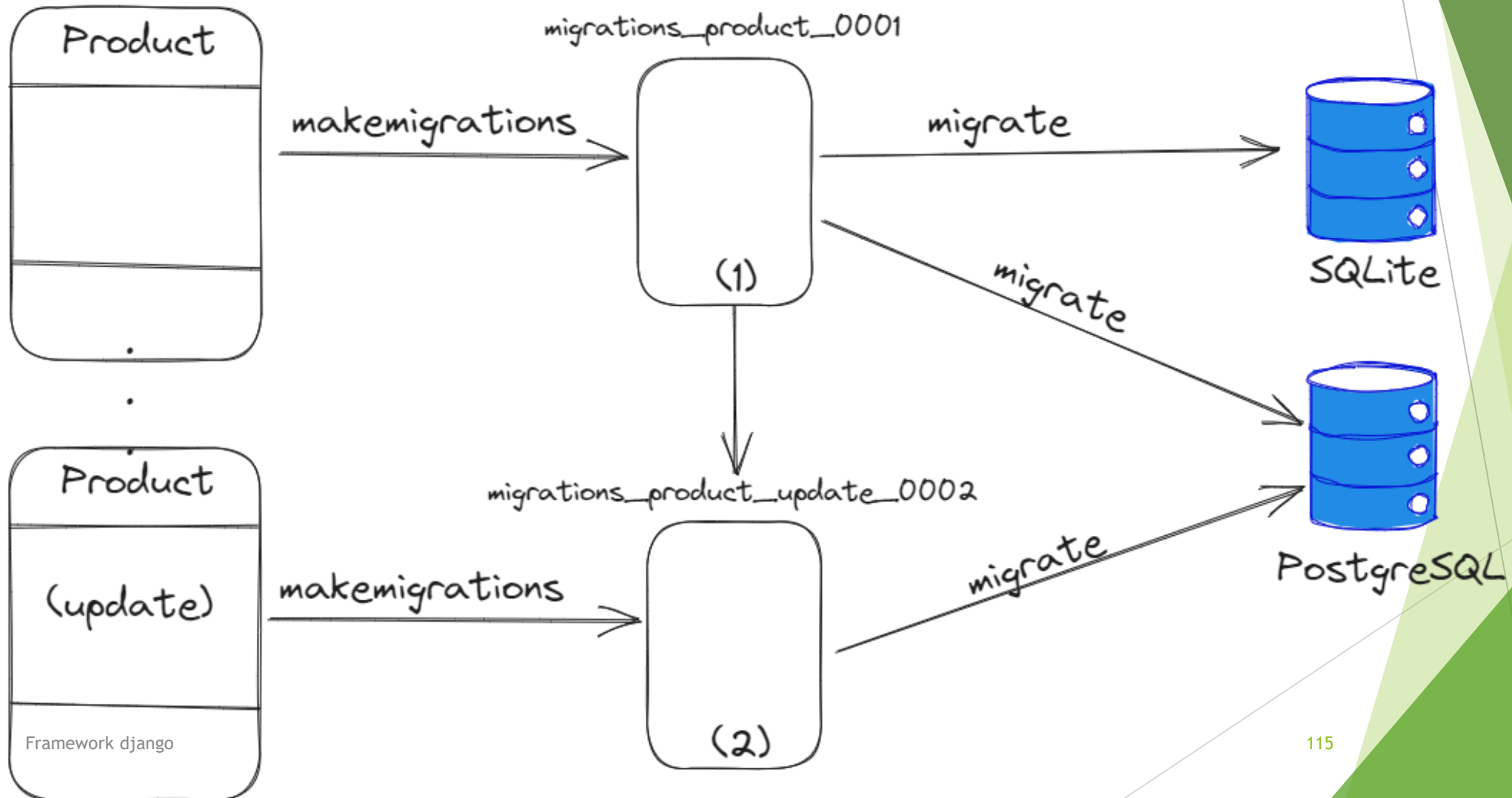
django Models: classe et instance

Démarche

- ▶ Créer classe Modèle
- ▶ Créer et appliquer les migrations
- ▶ Manipulation des instances du modèle:
 - ▶ Create
 - ▶ Update
 - ▶ Delete
- ▶ Différents backend

Étape par Étape

Models et Migrations

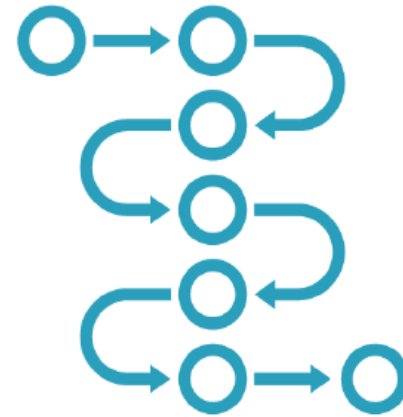


Migrations



Models

Classes Python
Associées aux tables de
la BD



Migrations

Ensemble de scripts
décrivant le schéma de
la BD

Migration Workflow

Important: Make sure your app is in INSTALLED_APPS

Step 1: Change model code

Step 2: Generate migration script (check it!)

```
python manage.py makemigrations
```

Step 3: Run migrations

```
python manage.py migrate
```

products.models.py: modèle Product

Étape par Étape

```
from django.db import models

class Product(models.Model):
    name=models.CharField(max_length=100)
    stock_count=models.IntegerField(default=0)
    price=models.DecimalField(max_digits=6,decimal_places=2)
```

settings.py: installer l'application

Étape par Étape

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'products.apps.ProductsConfig', #new
]
```

Créer et appliquer la migration

Étape par Étape

créer migrations

(my_env) \$ python manage.py makemigrations

afficher migrations

(my_env) \$ python manage.py sqlmigrate products 0001

appliquer les migrations

(my_env) \$ python manage.py migrate

Activité

- ▶ Changer la configuration du projet django pour se connecter à la base SQLite.
- ▶ Appliquer les migrations
- ▶ Interpréter le résultat: comparer les requêtes générées avec **PostgreSQL** et **SQLite**

Étape par Étape

Model Product: store data

lancer une console interactive

```
(my_env) $ python manage.py shell
```

Créer et sauvegarder un produit

```
>>> from products.models import Product
>>> p= Product(name="Clavier", stock_count=50, price=30)
>>> p.id
>>> p.id is None
>>> p.save()
>>> p.id
```

Model Product: update data

```
# update product
```

```
>>> p.price=40
```

```
>>> p.save()
```

Étape par Étape

Model Product: delete data

```
# delete product
```

```
>>> p.delete()
```

django Models: Fields

Éléments du contenu

- ▶ Model Fields
 - ▶ Type des Fields et options
 - ▶ Effet sur les formulaires et validation
- ▶ Relations:
 - ▶ ForeignKey
 - ▶ OneToOne
 - ▶ ManyToMany
 - ▶ Créer des relations entre les objets

```
class Person(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
```

Model Fields

Class attributes mapped to DB columns

Doivent être instances des **classes Field**

Exemple: CharField, IntegerField,...

Model Fields

Les Classes Fields déterminent:

- types de colonnes dans la BD (Integer, varchar,...)
- type de widget dans le formulaire

Field Options

- Validation au niveau de la BD
- Formulaire et validation
- valeurs par défaut
- . . .

documentation

- <https://docs.djangoproject.com/en/4.2/ref/models/fields/>

Storing Numbers

BooleanField

IntegerField
(et variantes)

FloatField

DecimalField

Storing Text

CharField

HTML: input text

Required
max_length

TextField

Text plus large

HTML: TextArea

EmailField

URLField

FilePathField

SlugField

Other Common Field Types

DateField

TimeField

DateTimeField

DurationField

FileField

ImageField

JSONField

BinaryField

Démo: TextFields

Étape par Étape

```
class Product(models.Model):
    name=models.CharField(max_length=100)
    stock_count=models.IntegerField(default=0)
    price=models.DecimalField(max_digits=6,decimal_places=2)
    description=models.TextField(default="")
```

migrations

rollback products migrations

(my_env) \$ python manage.py migrate products zero

créer migrations

(my_env) \$ python manage.py makemigrations

appliquer les migrations

(my_env) \$ python manage.py migrate

Étape par Étape

products.admin.py

```
from django.contrib import admin
from .models import Product

# Register your models here.

admin.site.register(Product)
```

Admin interface

Add product

Name:

Stock count:

Price:

Description:

SAVE

Save and add another

Save and continue editing

Étape par Étape

Options: Null et Blank

- ▶ Ajouter un nouveau produit sans description
- ▶ Interpréter le résultat
- ▶ Par défaut, tous les champs n'acceptent pas la valeur **null**
- ▶ Option **null**: validation **coté BD**
- ▶ Option **blank**: autorisation de valeur vide **coté formulaire**

Options: blank et null

```
class Product(models.Model):
    name=models.CharField(max_length=100)
    stock_count=models.IntegerField(default=0)
    price=models.DecimalField(max_digits=6,decimal_places=2)
    description=models.TextField(default="",blank=True)
```

Model Field Options 1

Make Field Nullable (default is non-NULL)

```
models.IntegerField(null = True)
```

Allow empty values in forms (Not db-related!)

```
models.CharField(blank = True)
```

Default value

```
models.CharField(default = 'Example')
```

Model Field Options 2

Add unique constraint

```
models.CharField(unique = True)
```

Add an index

```
models.IntegerField(db_index = True)
```

Set column name

```
models.BooleanField(db_column = "my_column_name")
```

Type-specific options

```
models.DateTimeField(auto_now = True)
```

Model Field Options 3

Set field label

```
iban = models.CharField(verbose_name = "Bank Account", ...)
```

Additional help text

```
name = models.CharField(help_text = "Enter your full name")
```

Model Field: Exercice

- ▶ Ajouter un Field **sku**:
 - ▶ Un code unique pour chaque produit
 - ▶ Chaîne de caractère de taille maximale 20.
 - ▶ Le label du formulaire doit afficher **stock keeping unit** au lieu de sku
- ▶ Appliquer migration workflow

Model Field: Exercice Solution

```
class Product(models.Model):

    name=models.CharField(max_length=100)

    stock_count=models.IntegerField(default=0,help_text="how many items are currently in stock")

    price=models.DecimalField(max_digits=6,decimal_places=2)

    description=models.TextField(default="",blank=True)

    sku=models.CharField(verbose_name="Stock Keeping Unit", max_length=20, unique=True)
```

migrations

rollback products migrations

(my_env) \$ python manage.py migrate products zero

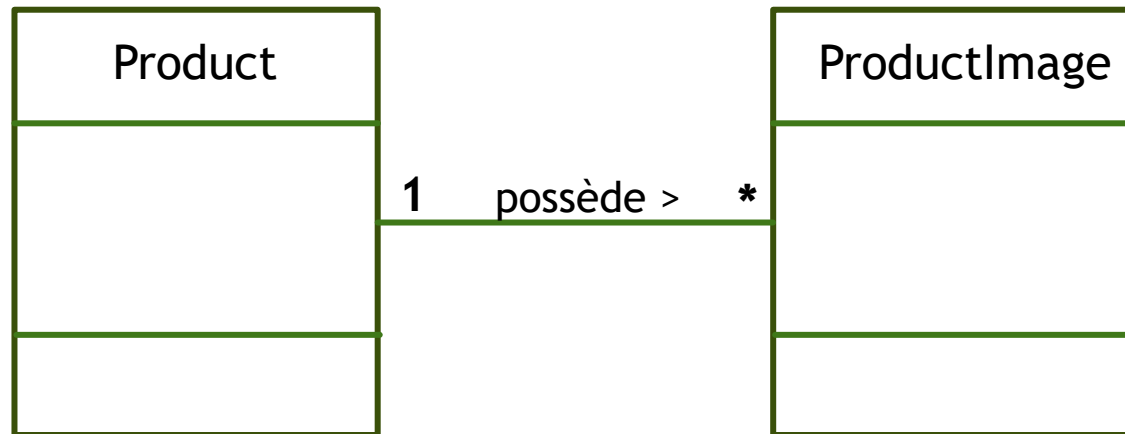
créer migrations

(my_env) \$ python manage.py makemigrations

appliquer les migrations

(my_env) \$ python manage.py migrate

Model Relations: One-To-Many



Définir le modèle ProductImage

Étape par Étape

```
class ProductImage(models.Model):
    image=models.ImageField()
    product=models.ForeignKey('Product',on_delete=models.CASCADE)
```

Install pillow package for ImageField

Étape par Étape

- > `python -m pip install Pillow`
- > `pip freeze > requirements.txt`

Étape par Étape

make and run migrations

- # rollback products migrations
(my_env) \$ python manage.py migrate products zero
- # créer migrations
(my_env) \$ python manage.py makemigrations
- # appliquer les migrations
(my_env) \$ python manage.py migrate

products.admin.py: register ProductImage

Étape par Étape

```
from django.contrib import admin
from .models import Product, ProductImage

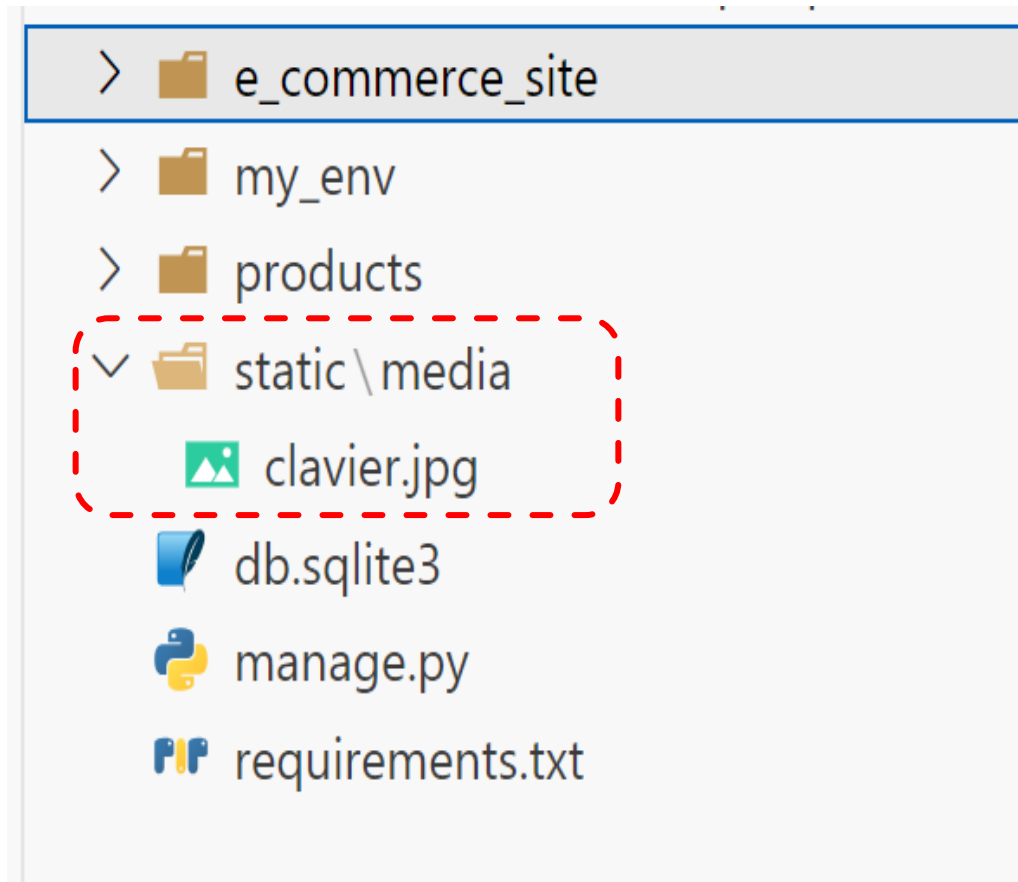
# Register your models here.

admin.site.register(Product)
admin.site.register(ProductImage)
```

settings.py: configuration media root

```
STATIC_ROOT = BASE_DIR / 'static'  
MEDIA_ROOT = STATIC_ROOT / 'media'  
MEDIA_URL = '/media/'
```

Créer le dossier static/media



urls.py: configuration media urls

```
from django.contrib import admin
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
]
```

```
product=models.ForeignKey(

    'Product',

    on_delete=models.CASCADE

)
```

← One-to-many relation
Created by **ForeignKey** field on the “many” side

← Target Model class for relation
Best practice: use name of target as a string

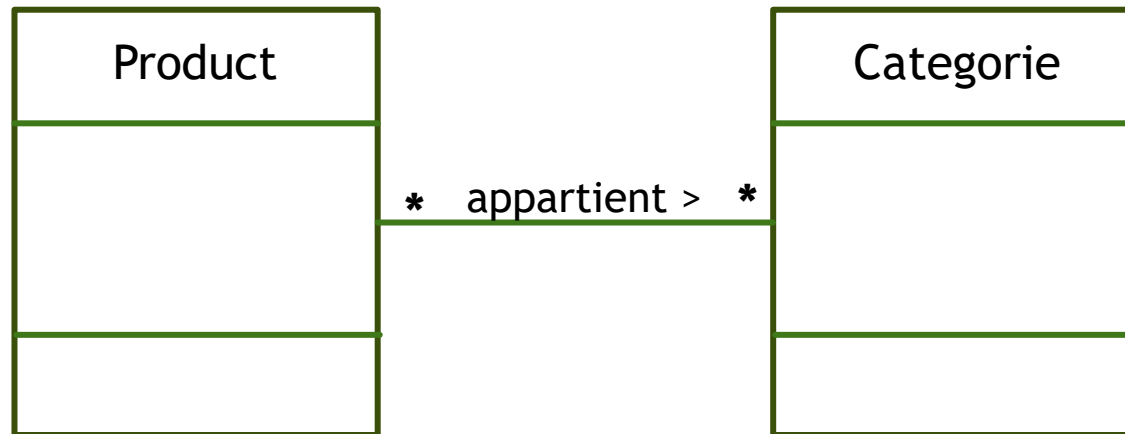
← Required: **on_delete**
Specify behaviour when related object is deleted

See:

<https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.ForeignKey>

Similar: **OneToOneField**
Like **ForeignKey** with **unique=True**

Model Relations: Many-To-Many



products.models.py: Categorie Model

```
class Categorie(models.Model):  
    name=models.CharField(max_length=100)  
    products=models.ManyToManyField('Product')
```

Étape par Étape

make and run migrations

rollback products migrations

(my_env) \$ python manage.py migrate products zero

créer migrations

(my_env) \$ python manage.py makemigrations

appliquer les migrations

(my_env) \$ python manage.py migrate

products.admin.py: register Categorie

```
from django.contrib import admin
from .models import Product, ProductImage,
Categorie
```

```
# Register your models here.
```

```
admin.site.register(Product)
admin.site.register(ProductImage)
admin.site.register(Categorie)
```

```
class Category(models.Model):

    products = models.ManyToManyField(

        'Product'

    )
```

- ◀ **Many-to-many relation**
 - Created by ManyToManyField
 - Can be on either side of relation
 - Join table in database

- ◀ **Target Model class for relation**
 - Best practice: use name of target as a string

django Models: Managers et QuerySets

Éléments du contenu

- ▶ Créer des requêtes
 - ▶ Managers
 - ▶ QuerySets
 - ▶ filter() et exclude()
 - ▶ Laziness
 - ▶ Limiter et ordonner les résultats
 - ▶ Relations
 - ▶ Aggrégations
 - ▶ F() et Q()

Managers

Product.objects

- ▶ Un **gestionnaire** (objet **Manager**) est l'interface par laquelle les opérations de requêtes de base de données sont mises à disposition aux modèles django.
- ▶ il existe au moins un **Manager** pour chaque modèle.
- ▶ par défaut, django ajoute un **Manager** nommé **objects** à chaque modèle.

Manger et QuerySet

chaque classe Modèle a un Manager: **objects**

On utilise le **Manager** pour exécuter des requêtes sur des tables

```
Product.objects.count()
```

```
Product.objects.get(pk=1)
```

Many methods return a QuerySet instance

A QuerySet represents a database query

```
claviers = Product.objects.filter(category__name="Clavier")
```

Managers et QuerySets

Toutes les méthodes **QuerySet** sont disponibles dans le **Manager**

Exemples: `objects.count()`, `objects.filter()`, ...

Le **Manager** **a** un **QuerySet** **interne**

Les appels de méthodes **sont passés** à ce **QuerySet**

Laziness

QuerySets are **lazy**. The following **does NOT run** any SQL

```
in_stock = Product.objects.filter(stock_count__gt=0)
```

This allows **chaining** of filters

```
claviers_in_stock = in_stock.filter(category__name="Clavier")
```

```
moin_chers_claviers = claviers_in_stock.order_by("price")[:5]
```

Laziness

What will cause a QuerySet to run its SQL?

`str(in_stock)` or `template: {{ in_stock }}`

for `product` in `in_stock`:

...

`list(in_stock)`

filter() et exclude()

```
Product.objects.filter(name__contains="a")
```

```
Product.objects.exclude(stock_count=0, price=10)
```

Select 0 or more rows from the database

Multiple arguments are combined with **AND**

Return a QuerySet

get(): selecting a Single row

Select with primary key

```
Product.objects.get(pk=5)
```

```
Product.objects.get(id=5)
```

With a unique column

```
Product.objects.get(sku="abc123z")
```

Args work similar to filter()

Raises exception if not exactly 1 match

In views

```
get_object_or_404(Product, pk=5)
```

Lookups

Allow specifying more complex **WHERE** clauses with `get()`, `filter()`, `exclude()`

Syntax: `field__lookuptype=value`

```
in_stock = Product.objects.filter(stock_count__gt=0)
```

Lookup across relations

```
claviers_in_stock = in_stock.filter(category__name="Clavier")
```

Combining both: get all images of claviers that are in stock

```
ProductImage.objects.filter(product__category__name="Clavier", product__stock_count__gt=0)
```

Limiting and Ordering

Use order_by and reverse() to sort results

```
Product.objects.order_by("price")
```

```
Product.objects.order_by("-price")
```

```
Product.objects.order_by("name", "-price")
```

```
Product.objects.order_by("category__name").reverse()
```

Limit results by slicing

```
Product.objects.order_by("price")[:10]
```


Links

- ▶ All QuerySets methods: <https://docs.djangoproject.com/en/4.2/ref/models/queries/>
- ▶ All Lookups: <https://docs.djangoproject.com/en/4.2/ref/models/queries/#field-lookups>

Related Manager

lancer une console interactive

(my_env) \$ python manage.py shell

Related Manager

```
>>> from products.models import Category
```

```
>>> c= Category.objects.get(pk=1)
```

```
>>> c.products #related Manager object
```

```
>>> c.products.all()
```

```
>>> c.products.filter(price__lt=10)
```

Related Manager

Related Manager

```
>>> from products.models import Product
>>> p= Product.objects.get(pk=1)
>>> p.category_set #related Manager object
>>> p.category_set.all()
```

Relations: related_name

```
class Categorie(models.Model):  
    name=models.CharField(max_length=100)  
    products=models.ManyToManyField('Product', {related_name='categories'})  
  
    def __str__(self):  
        return self.name
```

Related Manager

Related Manager

```
>>> from products.models import Product
>>> p= Product.objects.get(pk=1)
>>> p.categories #related Manager object
>>> p.categories.all()
```

Related Manager

```
# Related Manager
```

```
>>> from products.models import ProductImage
>>> i= ProductImage.objects.get(pk=1)
>>> i.product #product reference
>>> i.product_id
>>> i.product.id
>>> i.product.name
```

Aggregate et annotate

```
>>> from products.models import Product, Category
>>> from django.db.models import Avg, Count
>>> Product.objects.aggregate(Avg('price'))
>>> Category.objects.annotate(Avg('products__price'))
>>> Category.objects.annotate(avg_price=Avg('products__price')).values()
```

F(): référencer les valeurs de Fields

F expressions

```
>>> from django.db.models import F
>>> from products.models import Category, Product
>>> from decimal import Decimal
>>> Product.objects.filter(description__contains=F('name'))
>>> Category.objects.get(name='clavier').products.
    update(price=Decimal(0,9)*F('price'))
```


Complex Lookups with Q() function

Q() function

```
>>> from django.db.models import Q
>>> in_stock=Q( stock_count__gt=0 )
>>> no_images=Q( images=None)

>>> Product.objects.filter(in_stock)
>>> Product.objects.filter(~in_stock)
>>> Product.objects.filter(no_images | ~in_stock)
>>> Product.objects.filter(no_images&~in_stock)
>>> no_images_or_not_in_stock=no_images|~in_stock
```

django Models: Personnaliser le comportement des modèles

Éléments de contenu

- ▶ Model Meta class
- ▶ Custom methods
- ▶ Custom managers

Model Meta class

```
class Product(models.Model):
    name=models.CharField(max_length=100)
    stock_count=models.IntegerField(default=0,help_text="how many...")
    price=models.DecimalField(max_digits=6,decimal_places=2)
    description=models.TextField(default="",blank=True)
    sku=models.CharField(verbose_name="Stock Keeping Unit", max_length=20, unique=True)
```

```
class Meta:
    ordering=['price']

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

>>> Product.objects.all()

Model Meta class: autres exemples

```
class Meta:  
    ordering=['-price', 'name']
```

Model Meta class: autres exemples

```
class Categorie(models.Model):
    name=models.CharField(max_length=100)
    products=models.ManyToManyField('Product', related_name='categories')
```

```
class Meta:
    verbose_name_plural='categories'
    ordering=['name']
```

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

<https://docs.djangoproject.com/en/4.2/ref/models/options/>

Model Meta class: constraints

```
class Product(models.Model):
    name=models.CharField(max_length=100)
    stock_count=models.IntegerField(default=0,help_text="how many items are currently in stock")
    price=models.DecimalField(max_digits=6,decimal_places=2)
    description=models.TextField(default="",blank=True)
    sku=models.CharField(verbose_name="Stock Keeping Unit", max_length=20, unique=True)
```

```
class Meta:
    ordering=['price']
    constraints=[
        models.CheckConstraint(check=models.Q(price__gte=0),
                                name='price_not_negative')
    ]
```

Essayer d'ajouter un produit avec un prix négatif

Autres meta-options

<https://docs.djangoproject.com/en/4.2/ref/models/options/>

Custom methods

```
class Product(models.Model):
    name=models.CharField(max_length=100)
    stock_count=models.IntegerField(default=0,help_text="how many items are currently in stock")
    price=models.DecimalField(max_digits=6,decimal_places=2)
    description=models.TextField(default="",blank=True)
    sku=models.CharField(verbose_name="Stock Keeping Unit", max_length=20, unique=True)
```

```
@property
def tva(self):
    return Decimal(.2)*self.price
```

```
def get_absolute_url(self):
    return reverse('product-detail', kwargs={'pk':self.id})
```

Advanced custom method

```
[ from django.utils.text import slugify ]
```

```
class Product(models.Model):
    name=models.CharField(max_length=100)
    stock_count=models.IntegerField(default=0,help_text="how many ...in stock")
    price=models.DecimalField(max_digits=6,decimal_places=2)
    description=models.TextField(default="",blank=True)
    sku=models.CharField(verbose_name="Stock Keeping Unit", max_length=20, unique=True)
    slug=models.SlugField()
```

```
[ def save(self, *args, **kwargs): # new
    if not self.slug:
        self.slug = slugify(self.name)
    return super().save(*args, **kwargs) ]
```

Custom manager

```
class ProductInStockQuerySet(models.QuerySet):
    def in_stock(self):
        return self.filter(stock_count__gt=0)

class Product(models.Model):
    #.....
    objects = models.Manager()
    in_stock = ProductInStockQuerySet.as_manager()
```

Product.in_stock.all()

django Models: Migrations

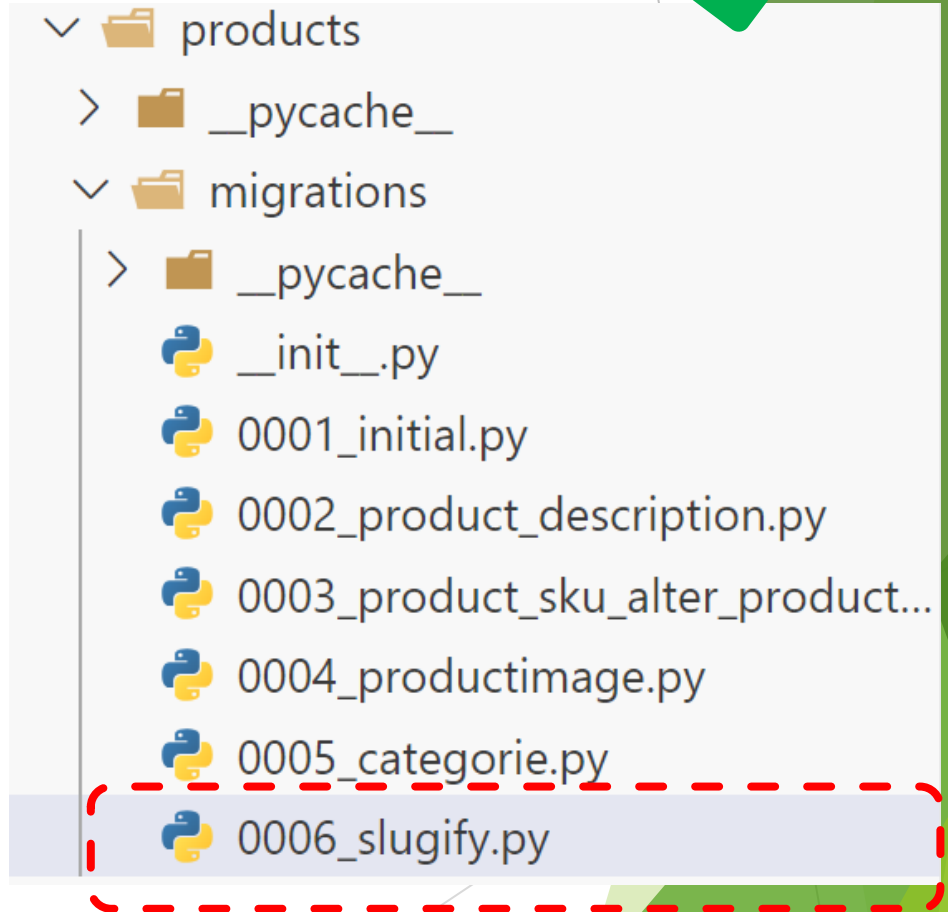
Custom migration

- ▶ objectifs
 - ▶ Créer une migration personnalisée permettant la génération automatique de Slug pour les produits existants dans la base de données

Créer une migration vide

Étape par Étape

(my_env) > python manage.py --empty products



0006.slugify.py: Ajouter le traitement

Étape par Étape

```
from django.db import migrations
from django.utils.text import slugify

def slugify_product_titles(apps, schema_editor):
    Product = apps.get_model("products", "Product")

    # Better: filter(...).update(F)
    for p in Product.objects.filter(slug=""):
        p.slug = slugify(p.name)
        p.save()

def undo_slugify(apps, schema_editor):
    pass
```

0006.slugify.py: définir les opérations

Étape par Étape

```
class Migration(migrations.Migration):

    dependencies = [
        ('products', '0005_categorie'),
    ]

    operations = [
        migrations.RunPython(slugify_product_titles, reverse_code=undo_slugify)
    ]
```


django Models: Optimisation du ORM

Raw SQL

- Pour exploiter toutes les possibilités SQL, **django ORM** permet aux développeurs d'exécuter des requêtes SQL:

```
# lancer une console interactive
```

```
(my_env) $ python manage.py shell
```

```
# Raw SQL
```

```
>>> from products.models import Product
```

```
>>> products=Product.objects.raw("Select * from products_product where price < 100")
```

```
>>> list(products)
```

```
>>> products=Product.objects.raw("Select * from products_product where price < %s",[100])
```

Projet: Blog Application

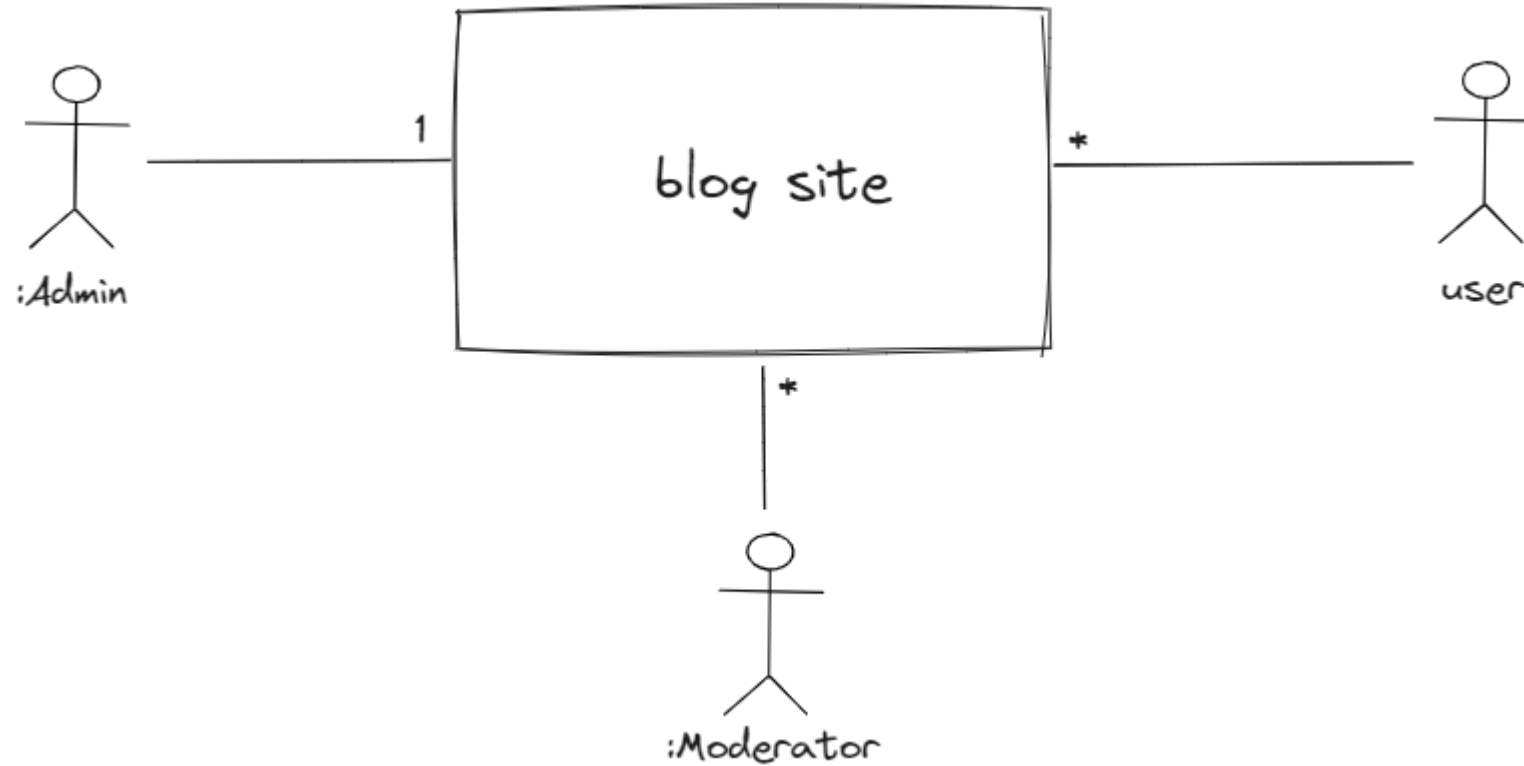
Framework django

195

Cahier des charges

- Nous souhaitons développer une application web permettant aux internautes de partager, commenter, consulter des **posts**.

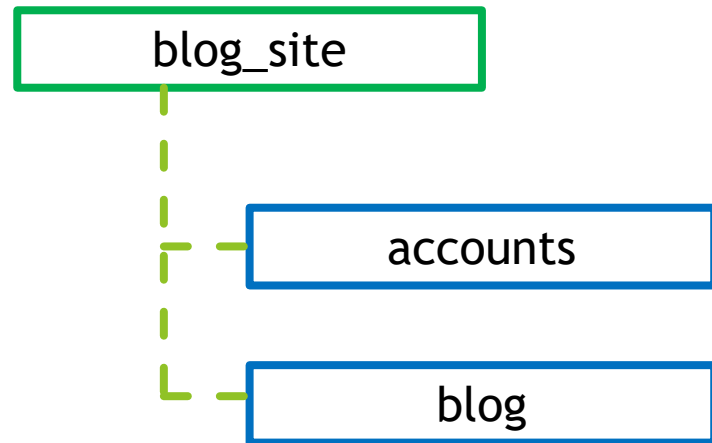
Diagramme de contexte statique



Rôles/permissions

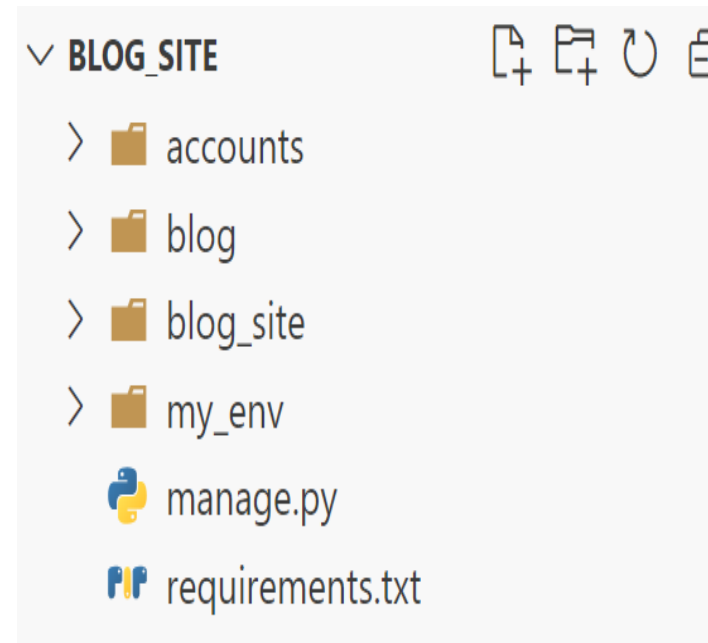
Permissions Rôle	consulter les posts	commenter les posts	gérer les posts	gérer les utilisateurs
user	✓	✓	ses propres posts	✗
moderator	✓	✓	✓	✗
Admin	✓	✓	✓	✓

Structure du projet



Manipulation

- ▶ Créer l'environnement virtuel
- ▶ Installer **django**
- ▶ Créer le projet **blog_site**
- ▶ Créer les deux applications **accounts** et **blog**
- ▶ Créer et configurer la connexion à une base de données **blog_site_db**
- ▶ appliquer les migrations initiales
- ▶ Activer l'application **blog**



Models: Post

- ▶ Un Post est caractérisé par:
 - ▶ Un **title** de type chaîne de caractère de taille max 250
 - ▶ Un **slug** de taille max 250
 - ▶ Un body de type texte
 - ▶ Date de publication (**publish**) de type datetime
 - ▶ **status** de type chaîne de caractère (**DRAFT**, **PUBLISHED**)
 - ▶ Un **utilisateur** peut publier plusieurs **posts**
- ▶ Ajouter le nécessaire pour:
 - ▶ Par défaut, les posts doivent être ordonnés par date de publication (ordre décroissant)
 - ▶ Indexer les posts par date de publication (ordre décroissant)
 - ▶ Un manager personnalisé pour les **posts** publiés

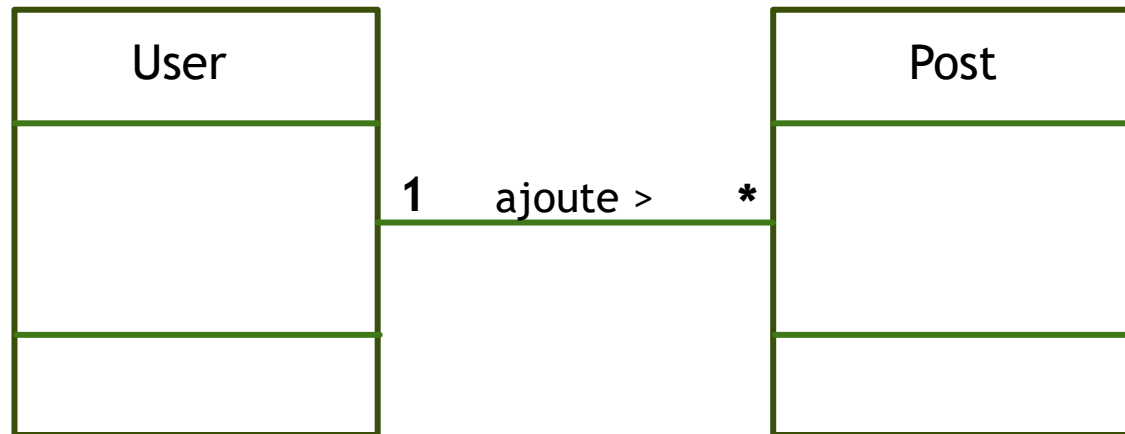
blog.models.py

```
class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
        PUBLISHED = 'PB', 'Published'

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=2,
                             choices=Status.choices,
                             default=Status.DRAFT)
```

Model Relations: One-To-Many



blog.models.py: one to many relation

```
from django.contrib.auth.models import User

class Post(models.Model):

    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
```

blog.models.py: custom manager

```
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset()\
            .filter(status=Post.Status.PUBLISHED)
```

```
class Post(models.Model):

    objects = models.Manager() # The default manager.
    published = PublishedManager() # Our custom manager.
```

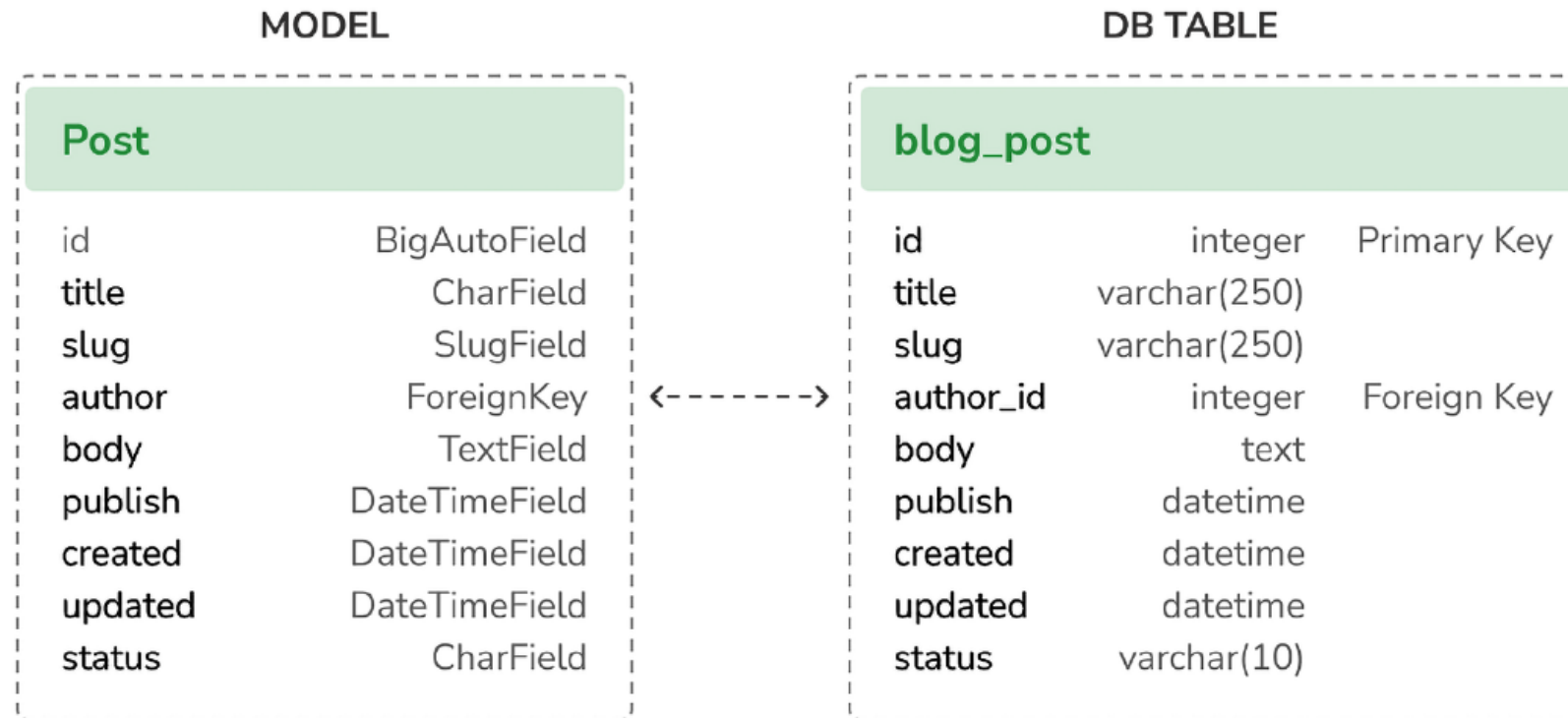
blog.models.py: meta-options

```
class Post(models.Model):

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]

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

Migration workflow



Créer administration site pour les models

Installer les applications

Créer les migrations

Appliquer les migrations

Créer superuser

Register Post dans admin.py

Personnaliser Admin panel

```
from django.contrib import admin

from .models import Post

# Register your models here.

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'author', 'publish', 'status']
```

Personnaliser Admin panel

```
@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'author', 'publish', 'status']
    list_filter = ['status', 'created', 'publish', 'author']
    search_fields = ['title', 'body']
    prepopulated_fields = {'slug': ('title',)}
    raw_id_fields = ['author']
    date_hierarchy = 'publish'
    ordering = ['status', 'publish']
```

Working with QuerySets and Managers

Ajouter un post

lancer une console interactive

(my_env) \$ python manage.py shell

```
>>> from blog.models import Post
```

```
>>> from django.contrib.auth.models import User
```

```
>>> user = User.objects.get(username='admin')
```

```
>>> post = Post(title='Another post', slug='another-post', body='Post body.', author=user)
```

```
>>> post.save()
```

```
>>> Post.objects.create(title='One more post', slug='one-more-post', body='Post body.', author=user)
```

```
>>> Post.objects.filter(publish__year=2023).exclude(title__startswith='first')
```

```
>>> Post.published.filter(title__startswith='first')
```

Post: build list and detail views

Étape par Étape

- ▶ Créer et implémenter list view
- ▶ Créer et implémenter detail view
- ▶ Définir les routes (urls.py)
- ▶ Créer et implémenter les templates `post_list.html` et `post_detail.html`

Étape par Étape

blog.views.py: post_list

```
from django.shortcuts import render
from .models import Post

# Create your views here.

def post_list(request):
    posts=Post.published.all()
    return render(request, 'blog/post/list.html',{'posts':posts})
```

blog.views.py: post_detail v1

Étape par Étape

```
def post_detail(request,id):
    try:
        post=Post.published.get(id=id)

    except Post.DoesNotExist:
        raise Http404('no Post found')

    return render(request, 'blog/post/detail.html',{'post':post})
```

blog.views.py: post_detail v2

Étape par Étape

```
from django.shortcuts import get_object_or_404

def post_detail(request,id):
    post=get_object_or_404(Post,id=id,status=Post.Status.PUBLISHED)

    return render(request,'blog/post/detail.html',{'post':post})
```


Adding urls patterns to views: blog.urls.py

Étape par Étape

```
from django.urls import path

from . import views

app_name='blog' #define application namespace

#domain.com/blog/...
urlpatterns=[
    path('',views.post_list, name='post_list'),
    path('<int:id>/',views.post_detail,name='post_detail'),
]
```

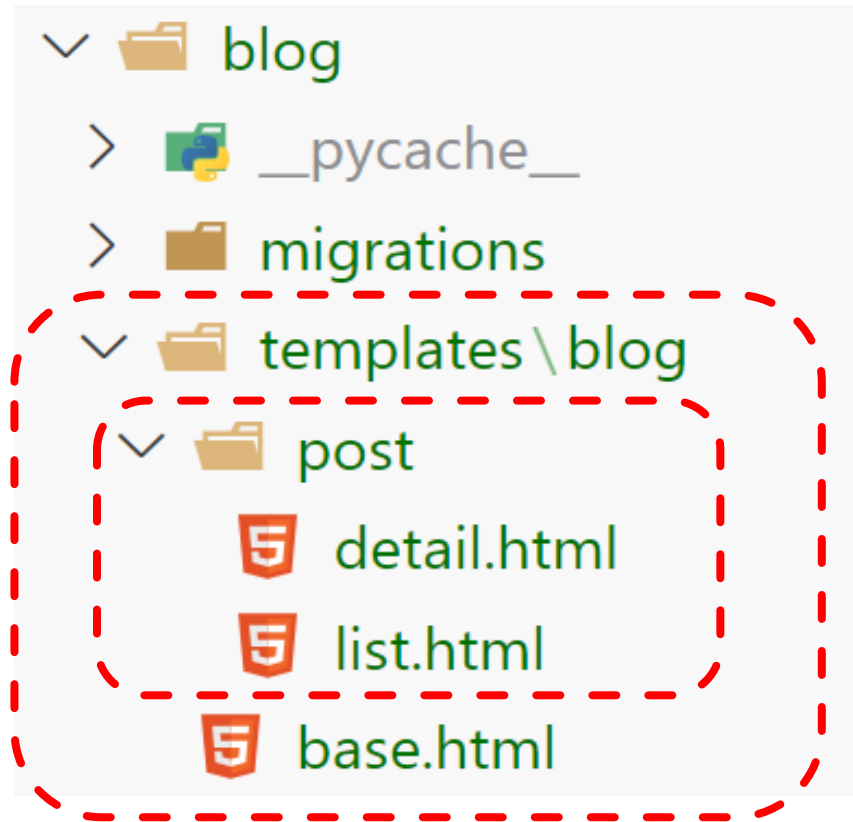
Adding urls patterns to views: blog_site.urls.py

Étape par Étape

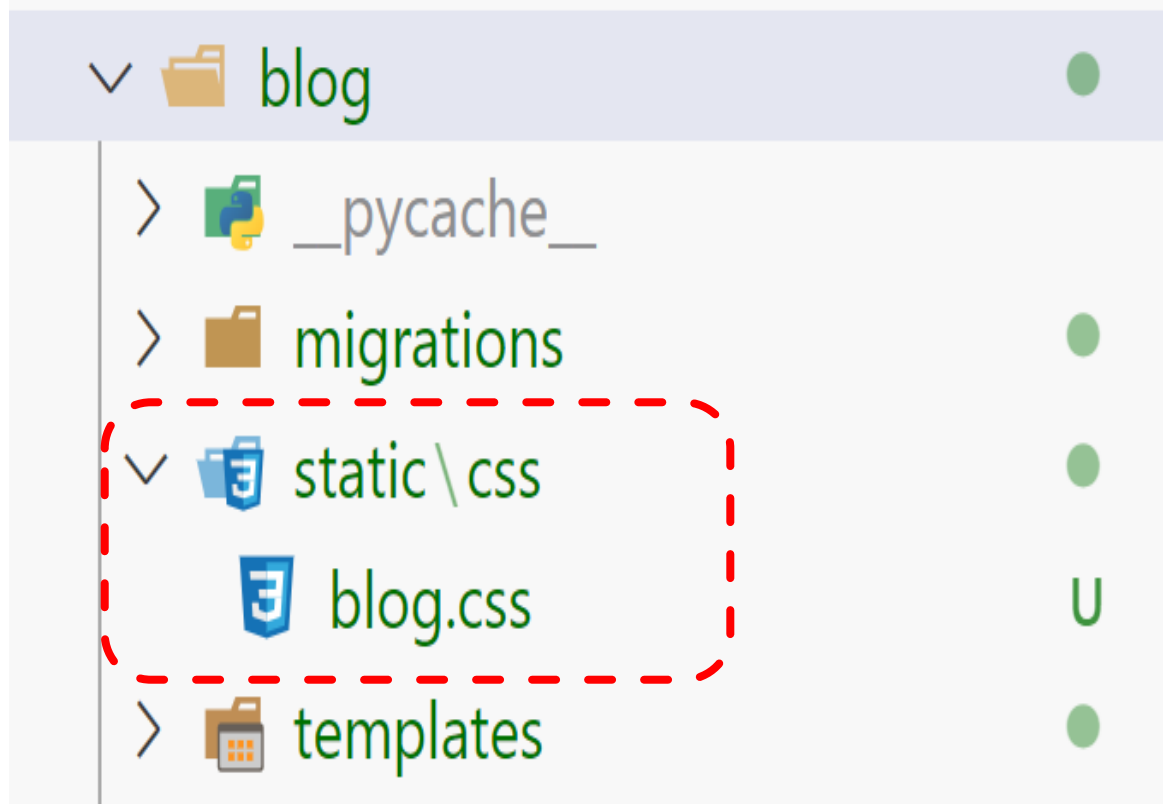
```
from django.contrib import admin
from django.urls import path, include

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

Post: Creating templates



blog.css



Étape par Étape

Étape par Étape

blog: base.html

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static 'css/blog.css' %}" rel="stylesheet">
</head>
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
    <p>This is my blog.</p>
  </div>
</body>
</html>
```

Étape par Étape

blog: post/list.html

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

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{% url 'blog:post_detail' post.id %}">
        {{ post.title }}
      </a>
    </h2>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
{% endblock %}
```

Étape par Étape

blog: post/detail.html

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

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
    <h1>{{ post.title }}</h1>
    <p class="date">
        Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|linebreaks }}
{% endblock %}
```

Post: add canonical URLs

Étape par Étape

```
from django.urls import reverse

class Post(models.Model):

    def get_absolute_url(self):
        return reverse('blog:post_detail', args=[self.id])
```


blog: update post/list.html

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

{% block title %}My Blog{% endblock %}

{% block content %}
    <h1>My Blog</h1>
    {% for post in posts %}
        <h2>
            <a href="{{ post.get_absolute_url }}">
                {{ post.title }}
            </a>
        </h2>
        <p class="date">
            Published {{ post.publish }} by {{ post.author }}
        </p>
        {{ post.body|truncatewords:30}}
    {% endfor %}
{% endblock %}
```

Creating SEO-friendly URLs for posts

Étape par Étape

- ▶ The canonical URL for a blog post detail view currently looks like `/blog/1/`.
- ▶ We will change the URL pattern to create **SEO-friendly** URLs for posts.
- ▶ We will be using both the **publish date** and **slug** values to build the URLs for single posts:

Post: slug field unique for date

Étape par Étape

```
class Post(models.Model):  
    slug = models.SlugField(max_length=250, unique_for_date='publish')
```

blog: make and run migrations

Étape par Étape

Blog: urls.py

Étape par Étape

```
#domain.com/blog/...
urlpatterns=[
    path('',views.post_list, name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
        views.post_detail,
        name='post_detail'),
]
```

blog: update post_detail

Étape par Étape

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                             status=Post.Status.PUBLISHED,
                             slug=post,
                             publish__year=year,
                             publish__month=month,
                             publish__day=day)

    return render(request, 'blog/post/detail.html', {'post':post})

post=Post.published.get(slug=post,publish__year=year,.....)
```

blog: Post: update canonical url

Étape par Étape

```
def get_absolute_url(self):
    return reverse('blog:post_detail',
                   args=[self.publish.year,
                         self.publish.month,
                         self.publish.day,
                         self.slug])
```

Pagination

- ▶ Il est impossible d'afficher des milliers de posts dans une seule page.
- ▶ Il faut subdiviser les posts en plusieurs parties (5 posts par exemple) appelées [pages](#)
- ▶ Avec django, il existe une classe prédéfinie [Paginator](#)

Adding pagination to post list view

Étape par Étape

```
from django.core.paginator import Paginator
def post_list(request):

    post_list=Post.published.all()

    # Pagination with 3 posts per page
    paginator = Paginator(post_list, 3)

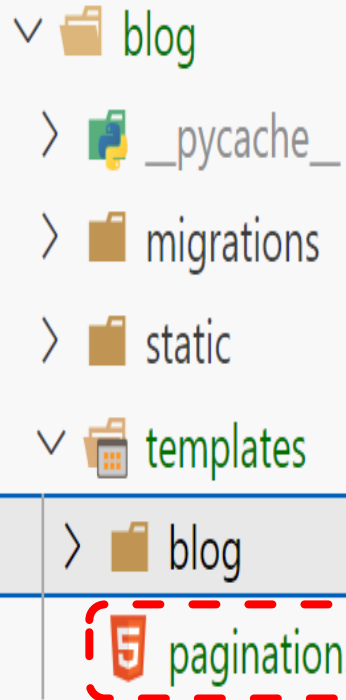
    page_number = request.GET.get('page', 1)

    posts=paginator.page(page_number)

    return render(request, 'blog/post/list.html',{'posts':posts})
```

Add pagination template

Étape par Étape



```
<div class="pagination">
  <span class="step-links">
    {% if page.has_previous %}
      <a href="?page={{ page.previous_page_number }}">Previous</a>
    {% endif %}
    <span class="current">
      Page {{ page.number }} of {{ page.paginator.num_pages }}.
    </span>
    {% if page.has_next %}
      <a href="?page={{ page.next_page_number }}">Next</a>
    {% endif %}
  </span>
</div>
```

include pagination template

```
{% extends "blog/base.html" %}
{% block title %}My Blog{% endblock %}
{% block content %}
<h1>My Blog</h1>
{% for post in posts %}
  <h2>
    <a href="{{ post.get_absolute_url }}">
      {{ post.title }}
    </a>
  </h2>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|truncatewords:30}}
{% endfor %}
{% include "pagination.html" with page=posts%}
{% endblock %}
```

Pagination: Traiter les exceptions

Étape par Étape

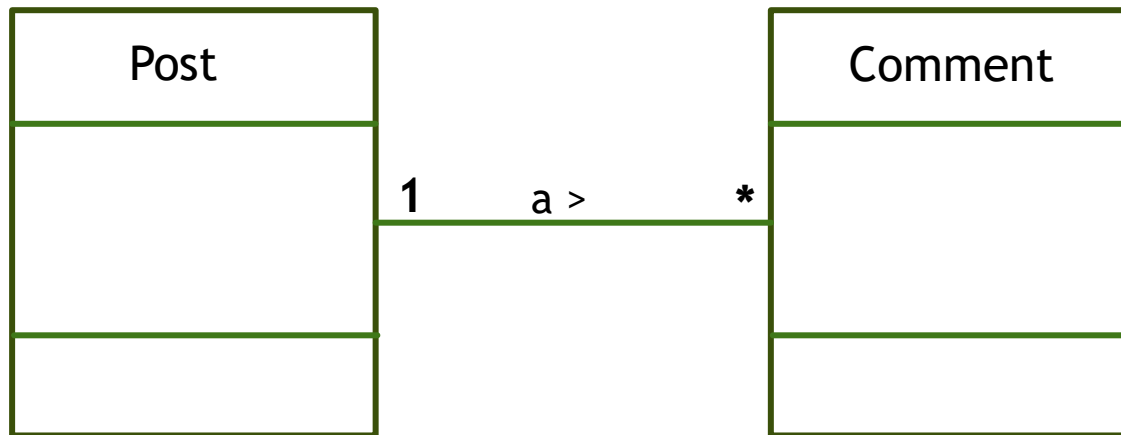
```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
```

```
def post_list(request):
    post_list=Post.published.all()
    # Pagination with 3 posts per page
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)

    
        try:
            posts = paginator.page(page_number)
        except PageNotAnInteger:
            # If page_number is not an integer deliver the first page
            posts = paginator.page(1)
        except EmptyPage:
            # If page_number is out of range deliver last page of results
            posts = paginator.page(paginator.num_pages)
    

    return render(request, 'blog/post/list.html', {'posts':posts})
```

Creating a Comment System



Étape par Étape

blog.models.py: Comment Model

Étape par Étape

```
class Comment(models.Model):

    post = models.ForeignKey('Post',
                             on_delete=models.CASCADE,
                             related_name='comments')

    name = models.CharField(max_length=80)
    email = models.EmailField()
    body = models.TextField()
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    active = models.BooleanField(default=True)
```

blog.models.py: Comment Model

Étape par Étape

```
class Comment(models.Model):

    class Meta:
        ordering = ['created']
        indexes = [
            models.Index(fields=['created']),
        ]

    def __str__(self):
        return f'Comment by {self.name} on {self.post}'
```

Make and run migrations

Étape par Étape

Adding comments to Admin site

Étape par Étape

```
@admin.register(Comment)
class CommentAdmin(admin.ModelAdmin):
    list_display = ['name', 'email', 'post', 'created', 'active']
    list_filter = ['active', 'created', 'updated']
    search_fields = ['name', 'email', 'body']
```

blog.forms.py: Creating comment Form

Étape par Étape

```
from django import forms
from .models import Comment

class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ['name', 'email', 'body']
```

Étape par Étape

blog.views.py:

```
from django.views.decorators.http import require_POST
@require_POST
def post_comment(request, post_id):
    post = get_object_or_404(Post, id=post_id, status=Post.Status.PUBLISHED)
    comment = None
    # A comment was posted
    form = CommentForm(data=request.POST)
    if form.is_valid():
        # Create a Comment object without saving it to the database
        comment = form.save(commit=False)
        # Assign the post to the comment
        comment.post = post
        # Save the comment to the database
        comment.save()
    return render(request, 'blog/post/comment.html',
                  {'post': post,
                   'form': form,
                   'comment': comment})
```

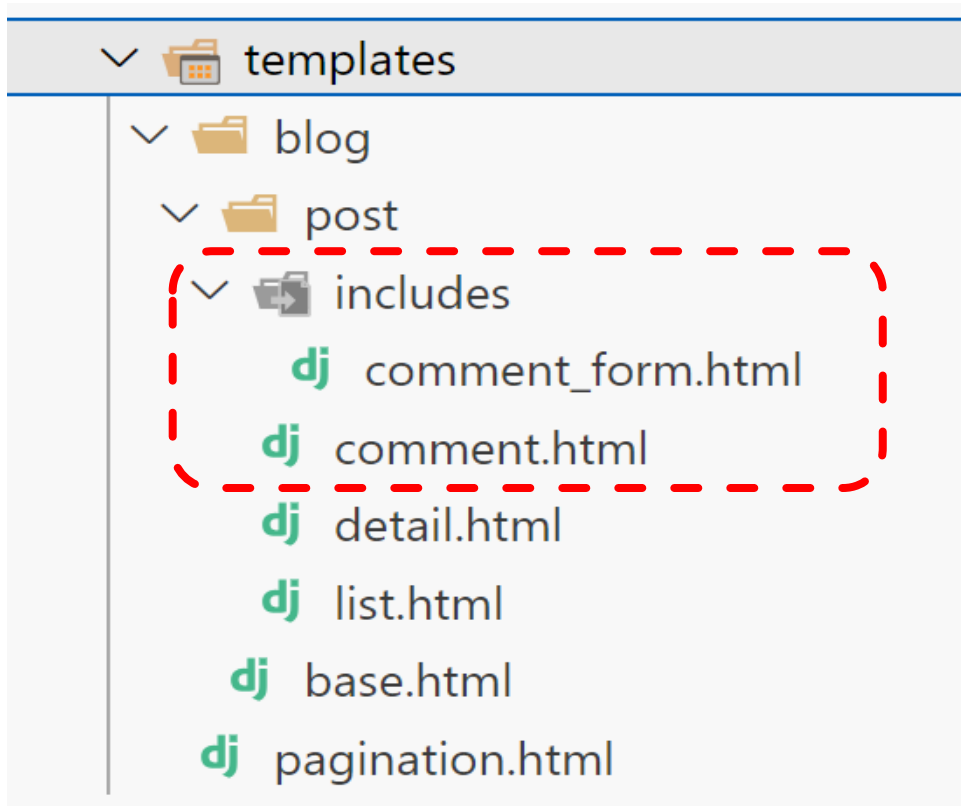
blog.urls.py: add comment path

Étape par Étape

```
#domain.com/blog/...
urlpatterns=[
    path('',views.post_list, name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
        views.post_detail,
        name='post_detail'),
    path('<int:post_id>/comment/',
        views.post_comment, name='post_comment'),
]
```

Create templates for comment

Étape par Étape



comment_form.html

Étape par Étape

```
<h2>Add a new comment</h2>
<form action="{% url 'blog:post_comment' post.id %}" method="post">

    {{ form.as_p }}
    {% csrf_token %}

    <p><input type="submit" value="Add comment"></p>
</form>
```

comment.html

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

{% block title %}Add a comment{% endblock %}

{% block content %}
    {% if comment %}
        <h2>Your comment has been added.</h2>
        <p><a href="{{ post.get_absolute_url }}">Back to the post</a></p>
    {% else %}
        {%- include "blog/post/includes/comment_form.html" %}
    {% endif %}
{% endblock %}
```

Adding comment to post_detail views

Étape par Étape

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                             status=Post.Status.PUBLISHED,
                             slug=post,
                             publish__year=year,
                             publish__month=month,
                             publish__day=day)

    # List of active comments for this post
    comments = post.comments.filter(active=True)
    # Form for users to comment
    form = CommentForm()

    return render(request,
                  'blog/post/detail.html',
                  {'post': post,
                  'comments': comments,
                  'form': form})
```


Adding comment to the post_detail.html

Étape par Étape

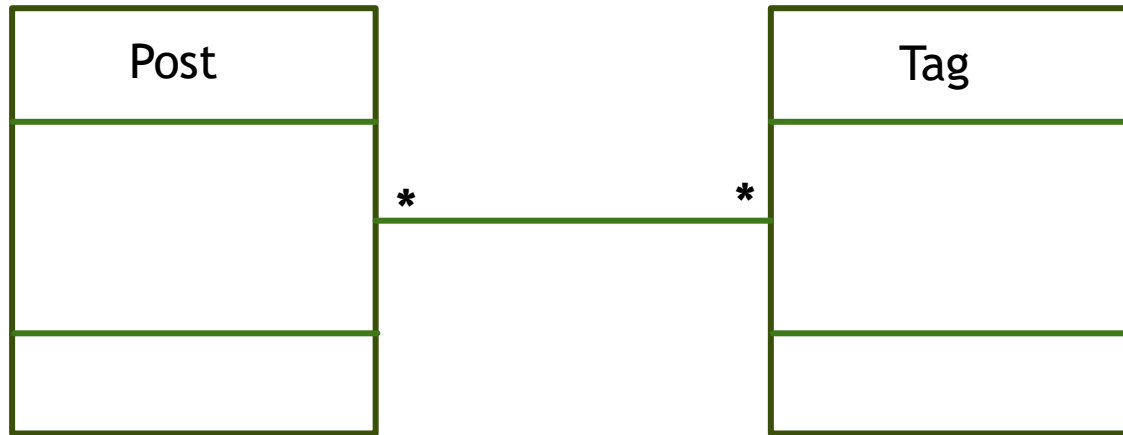
```
{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
  {% endwith %}
```

Adding comment to the post_detail.html

Étape par Étape

```
{% for comment in comments %}
  <div class="comment">
    <p class="info">
      Comment {{ forloop.counter }} by {{ comment.name }}
      {{ comment.created }}
    </p>
    {{ comment.body|linebreaks }}
  </div>
{% empty %}
  <p>There are no comments yet.</p>
{% endfor %}
{% include "blog/post/includes/comment_form.html" %}
{% endblock %}
```

Tagging post



Template: Custom Tag

- Django définit des template Tags dites **Buil-in template tags**

```
{% load library %}
```

```
{% block name %}
```

```
{% extends 'template' %}
```

```
{% include 'template' %}
```

```
{% static %}
```

```
{%if %}
```

```
{% elif %}
```

```
{% else %}
```

```
{% endif %}
```

```
{% for var in list %}
```

```
{% empty %}
```

```
{% endfor %}
```

```
{% comment %}
```

```
{% endcomment %}
```

<https://docs.djangoproject.com/en/4.2/ref/templates/builtins/>

Template: Custom Tag

- ▶ Django offre la possibilité d'ajouter des **Tags personnalisés**
- ▶ Par exemple, ajouter un **tag** permettant l'exécution d'un **QuerySet**
- ▶ Factoriser un traitement commun entre plusieurs templates



Custom tags: types (helper functions)

`simple_tag`

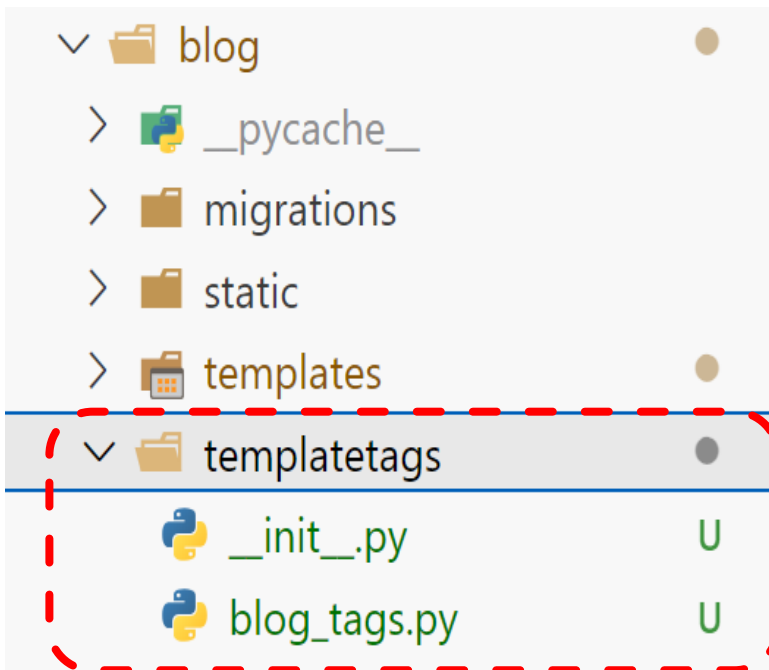
return value

`inclusion_tag`

render template

Template tags must live inside Django applications.

Étape par Étape



Creating a simple template tag

Étape par Étape

blog_tags.py

```
from ..models import Post
from django import template
```

```
register = template.Library()
```

```
@register.simple_tag
def total_posts():
    return Post.published.count()
```

template.Library instance, used to register custom tags and filters of blog application

base.html:Using simple custom tag

```
{% load blog_tags %}
```

```
{% load static %}
```

```
<!DOCTYPE html>
```

```
<html>
```

```
...
```

```
<body>
```

```
...
```

```
<div id="sidebar">
```

```
<h2>My blog</h2>
```

```
<p>This is my blog.</p>
```

```
I've written {% total_posts %} posts so far.
```

```
</div>
```

```
</body>
```

```
</html>
```

Framework django

Creating an inclusion template tag

Étape par Étape

blog_tags.py

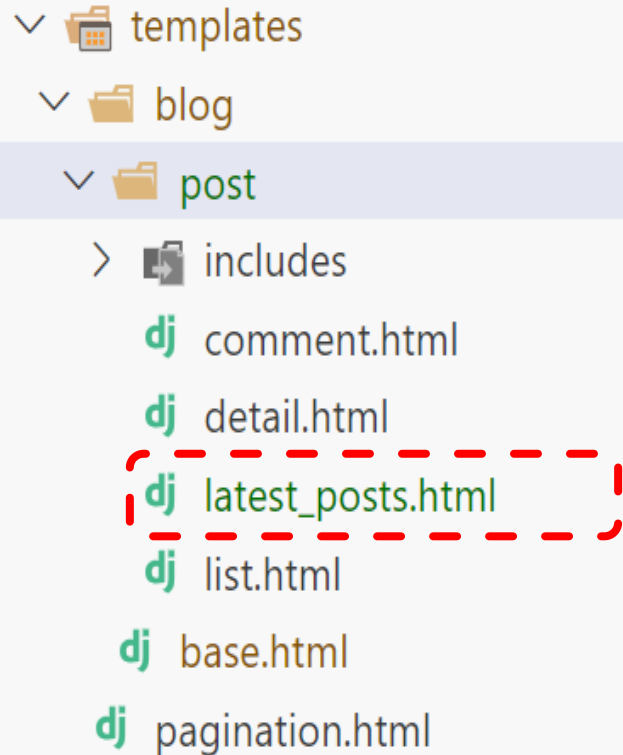
```
from ..models import Post
from django import template

register = template.Library()

@register.inclusion_tag('blog/post/latest_posts.html')
def show_latest_posts(count=5):
    latest_posts = Post.published.order_by('-publish')[:count]
    return {'latest_posts': latest_posts}
```

Creating latest_posts.html

Étape par Étape



```
<ul>
  {% for post in latest_posts %}
  <li>
    <a href="{{ post.get_absolute_url }}">
      {{ post.title }}
    </a>
  </li>
  {% endfor %}
</ul>
```

base.html:Using inclusion custom tag

Étape par Étape

```
{% load blog_tags %}

{% load static %}
<!DOCTYPE html>
<html>

...
<body>

...
<div id="sidebar">
  <h2>My blog</h2>
  <p>This is my blog.</p>
  I've written {% total_posts %} posts so far.
```

```
  <h3>Latest posts</h3>
  {% show_latest_posts 3 %}
</div>
</body>
</html>
```

Framework django

Custom tag: display most commented posts

blog_tags.py

```
from django import template
from django.db.models import Count
from ..models import Post

register = template.Library()

@register.simple_tag
def get_most_commented_posts(count=5):
    return Post.published.annotate(
        total_comments=Count('comments')
    ).order_by('-total_comments')[:count]
```

base.html

```
<div id="sidebar">
  <h2>My blog</h2>
  <p>This is my blog.</p>
  I've written {% total_posts %} posts so far.

  <h3>Latest posts</h3>
  {% show_latest_posts 3 %}
  <h3>Most commented posts</h3>
  {% get_most_commented_posts as most_commented_posts %}
  <ul>
    {% for post in most_commented_posts %}
      <li>
        <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
      </li>
    {% endfor %}
  </ul>
</div>
```

Custom template filter

- Django has a variety of built-in template filters that allow you to alter variables in templates

```
{{ value|add:"2" }}
```

```
{{ value|first }}
```

```
{{ value|capfirst }}
```

```
{{ value|length }}
```

```
{{ value|cut:" " }}
```

```
{{ value|date:"D d M Y" }}
```

<https://docs.djangoproject.com/en/4.2/ref/templates/builtins/#built-in-filter-reference>

```
{{ value|default:"nothing" }}
```

Creating a template filter to support Markdown syntax

- We will create a custom filter to enable you to use Markdown syntax in your blog posts and then convert the post body to HTML in the templates.

Basic Text Elements

Name	Markdown	HTML Output
Headings <small>Ids only allowed in extended markdown</small>	# Head 1	<h1>Head 1</h1>
	## Head 2	<h2>Head 2</h2>
	### Head 3 {#my-id}	<h3 id="my-id">Head 3</h3>
	#### Head 4	<h4>Head 4</h4>
	##### Head 5	<h5>Head 5</h5>
	##### Head 6	<h6>Head 6</h6>

Creating a template filter to support Markdown syntax

- We will create a custom filter to enable you to use Markdown syntax in your blog posts and then convert the post body to HTML in the templates.

Tables Extended

```
| Name | Age |
| ---- | --- |
| Kyle | 28 |
| Sally | 45 |
```

Name	Age
Kyle	28
Sally	45

```
| Right | Center | Left |
| ----: | :----: | :--- |
| Kyle | 28 | Hi |
| Sally | 45 | Bye |
```

Right	Center	Left
Kyle	28	Hi
Sally	45	Bye

Creating a template filter to support Markdown syntax

- We will create a custom filter to enable you to use Markdown syntax in your blog posts and then convert the post body to HTML in the templates.

Checklist

Extended

- [] Must include space ☐
- [x] Completed ☒

Install Markdown module

```
(my_env) > pip install markdown
```

```
(my_env) > pip freeze > requirements.txt
```

blog_tags.py: define custom filter

```
from django.utils.safestring import mark_safe
import markdown

@register.filter(name='markdown')
def markdown_format(text):
    return mark_safe(markdown.markdown(text))
```

blog: detail.html

```
% extends "blog/base.html" %}

{% load blog_tags %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
    <h1>{{ post.title }}</h1>
    <p class="date">
        Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|markdown }}
...

```

blog: list.html

```
{% block content %}
    <h1>My Blog</h1>
    {% for post in posts %}
        <h2>
            <a href="{{ post.get_absolute_url }}">
                {{ post.title }}
            </a>
        </h2>
        <p class="date">
            Published {{ post.publish }} by {{ post.author }}
        </p>
        {{ post.body|markdown|truncatewords_html:30 }}
    {% endfor %}
    {% include "pagination.html" with page=posts %}
{% endblock %}
```

Class Based Views (CBV)

Problématique

```
def post_list(request):
    posts=Post.objects.all()
    return render(request, 'blog/post/list.html',{'posts':posts})

def car_list(request):
    cars= Car.objects.all()
    return render(request, 'cars/list.html',{'cars':cars})

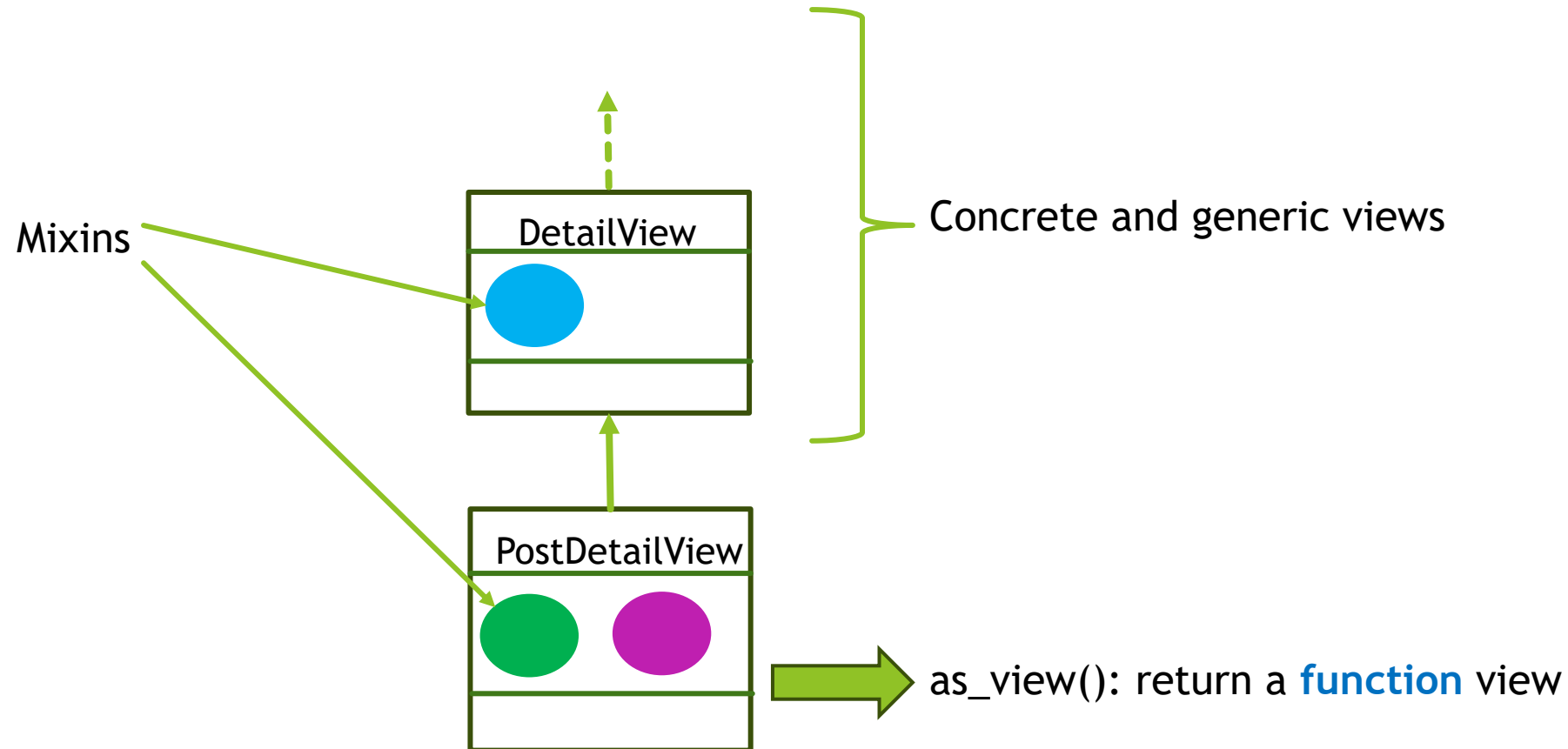
def post_detail(request,id):
    try:
        post=Post.objects.get(id=id)
    except Post.DoesNotExist:
        raise Http404('no Post found')

    return render(request, 'blog/post/detail.html',{'post':post})
```


Problématique

- ▶ Code répétitif
- ▶ Solution 01:
 - ▶ Créer des fonctions views **génériques**
 - ▶ Inconvénient: impossible de l'adapter ou l'enrichir avec d'autres fonctionnalités
- ▶ Solution 02:
 - ▶ Créer des views de type **class**
 - ▶ Avantage: possibilité de surcharge avec **héritage** et **Mixin**
 - ▶ Problème: interface avec les autres couches de l'architecture
 - ▶ Solution: créer et retourner une fonction view **ClassView.as_view()**

Class Based Views (CBV)

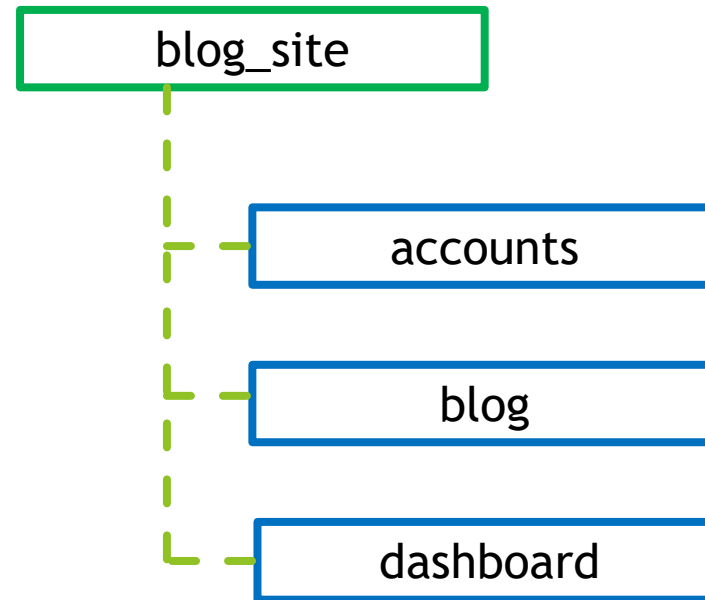


<https://ccbv.co.uk/projects/Django/4.0/>

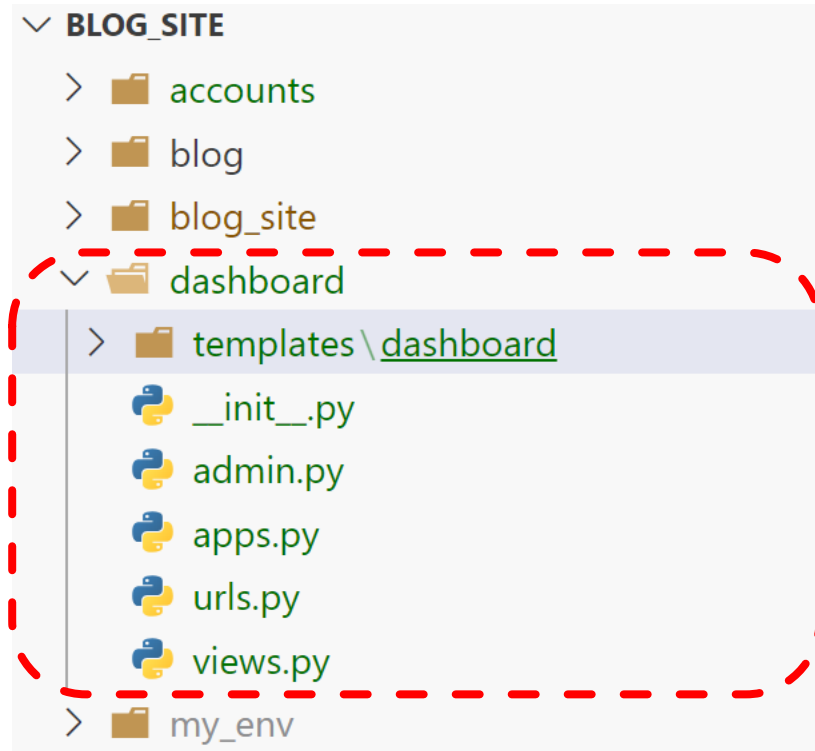
Application dashboard

► Un utilisateur **authentifié**:

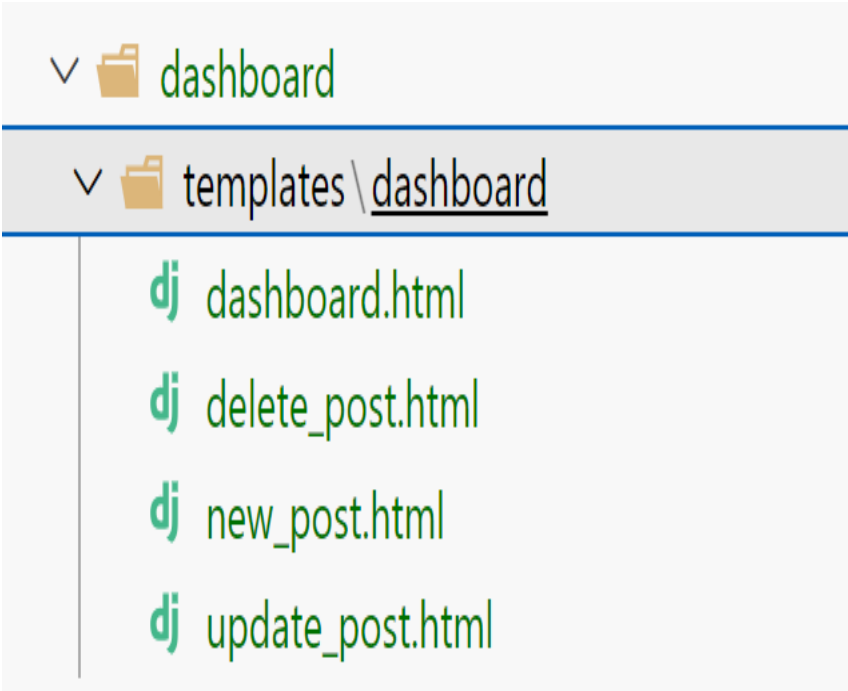
- Lister **ses** posts
- Ajouter un post
- Modifier **ses** posts
- Supprimer **ses** posts



Créer l'application dashboard



Dashboard: templates



dashboard.views: Lister ses posts

```
from django.views.generic import ListView
from blog.models import Post

class PostListView(ListView):
    model=Post
    template_name="dashboard/dashboard.html"
    paginate_by=3
    context_object_name='posts'# default post_list
```

dashboard.views: Lister ses posts

```
from django.views.generic import ListView
from blog.models import Post
```

```
class PostListView(ListView):
    #model=Post
    template_name="dashboard/dashboard.html"
    paginate_by=3
    context_object_name='posts'# default post_list
```

```
def get_queryset(self):
    user=self.request.user
    return Post.objects.filter(author__id=user.id)
```

dashboard: dashboard.html

```
{% extends "blog/base.html" %}
{% load blog_tags %}

{% block title %}My posts{% endblock %}

{% block content %}
{% for post in posts %}
    <h2>
        <a href="{{ post.get_absolute_url }}">
            {{ post.title }}
        </a>
    </h2>
    <p class="date">Published {{ post.publish }} by {{ post.author }}</p>
    {{ post.body|markdown|truncatewords_html:30 }}
{% endfor %}
{% include "pagination.html" with page=page_obj %}
{% endblock %}
```


dashboard.urls: add path

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

app_name='dashboard' #define application namespace

#domain.com/dashboard/...
urlpatterns=[
    path('',views.PostListView.as_view(), name='post_list'),
]
```

blog_site.urls: include dashboard urls

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls', namespace='blog')),
    path('dashboard/', include('dashboard.urls', namespace='dashboard')),
]
```

dashboard.views: add post

```
from django.views.generic import ListView, CreateView
from blog.models import Post
```

```
class PostCreateView(CreateView):
    model=Post
    template_name="dashboard/new_post.html"
    fields=["title", "body"]
```

dashboard.views: add post

```
from django.views.generic import ListView, CreateView
from blog.models import Post
```

```
class PostCreateView(CreateView):
    model=Post
    template_name="dashboard/new_post.html"
    fields=["title", "body"]

    def form_valid(self, form):
        form.instance.author=self.request.user
        form.instance.slug=slugify(form.instance.title)
        return super().form_valid(form)
```

dashboard: new_post.html

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

{% block title %}New Post{% endblock %}

{% block content %}
    <form method="post">
        {{form.as_p}}
        {% csrf_token %}
        <input type="submit" value="save">
    </form>
{% endblock %}
```

dashboard.urls: add path

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

app_name='dashboard' #define application namespace

#domain.com/dashboard/...
urlpatterns=[
    path('',views.PostListView.as_view(), name='post_list'),
    path('new',views.PostCreateView.as_view(), name='new_post'),
]
```

dashboard.views: update post

```
from django.views.generic import ListView, CreateView, UpdateView

class PostUpdateView(UpdateView):
    model=Post
    template_name="dashboard/update_post.html"
    fields=["title", "body"]

    def form_valid(self, form):
        form.instance.slug=slugify(form.instance.title)
        return super().form_valid(form)

    def get_success_url(self):
        return reverse('dashboard:post_list')
```

dashboard: update_post.html

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

{% block title %}Update Post{% endblock %}

{% block content %}
    <form method="post">
        {{form.as_p}}
        {% csrf_token %}
        <input type="submit" value="update">
    </form>

{% endblock %}
```


dashboard.urls: update path

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

app_name='dashboard' #define application namespace

#domain.com/dashboard/...
urlpatterns=[
    path('',views.PostListView.as_view(), name='post_list'),
    path('new',views.PostCreateView.as_view(), name='new_post'),
    path('<int:pk>/edit/',views.PostUpdateView.as_view(), name='update_post'),
]
```

dashboard.views: delete post

```
from django.urls import reverse, reverse_lazy

class BlogDeleteView(DeleteView):
    model = Post
    template_name = "dashboard/delete_post.html"
    success_url = reverse_lazy("dashboard:post_list")
```

dashboard: delete_post.html

```
{% extends "blog/base.html" %}
{% block content %}
    <h1>Delete post</h1>
    <form action="" method="post">{% csrf_token %}
        <p>Are you sure you want to delete "{{ post.title }}"?</p>
        <input type="submit" value="Confirm">
    </form>
{% endblock content %}
```

dashboard.urls: delete path

```
app_name='dashboard' #define application namespace

#domain.com/dashboard/...
urlpatterns=[
    path('',views.PostListView.as_view(), name='post_list'),
    path('new',views.PostCreateView.as_view(), name='new_post'),
    path('<int:pk>/edit/',views.PostUpdateView.as_view(), name='update_post'),
    path("<int:pk>/delete/", views.BlogDeleteView.as_view(),name="post_delete"),
]
```

Permissions: logged-in users

```
from django.contrib.auth.mixins import LoginRequiredMixin, UserPassesTestMixin

class PostListView(LoginRequiredMixin, ListView):
    #model=Post
    template_name="dashboard/dashboard.html"
    paginate_by=3
    context_object_name='posts'# default post_list
    def get_queryset(self):
        user=self.request.user
        return Post.objects.filter(author__id=user.id)
```

Permissions: owner post

```
from django.contrib.auth.mixins import LoginRequiredMixin, UserPassesTestMixin
```

```
class PostUpdateView(LoginRequiredMixin, UserPassesTestMixin, UpdateView):
```

```
    model=Post
```

```
    template_name="dashboard/update_post.html"
```

```
    fields=["title", "body"]
```

```
    def form_valid(self, form):
```

```
        form.instance.slug=slugify(form.instance.title)
```

```
        return super().form_valid(form)
```

```
    def get_success_url(self):
```

```
        return reverse('dashboard:post_list')
```

```
( def test_func(self):
```

```
    obj = self.get_object()
```

```
    return obj.author == self.request.user
```

User Authentication

User Authentication

- ▶ Django comes with a built-in **authentication framework** that can handle:
 - ▶ User authentication
 - ▶ Sessions
 - ▶ Permissions
 - ▶ User groups
- ▶ The authentication system includes **views** for common user actions:
 - ▶ logging in,
 - ▶ logging out,
 - ▶ password change,
 - ▶ password reset

settings.py

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

contains the core of the authentication framework, and its default models.

django content type system, which allows permissions to be associated with models

Application: accounts

```
INSTALLED_APPS = [
    'accounts.apps.AccountsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig',
    'dashboard.apps.DashboardConfig',
]
```

Middleware

- Permettent d'exécuter une chaine de traitements sur les requêtes

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Authentication: User model

- L'application django auth définit une classe modèle User par défaut

Never use the built-in Django User model directly, even if the built-in Django User implementation fulfill all the requirements of your application.

Authentication: custom user

```
from django.db import models
from django.contrib.auth.models import AbstractUser

# Create your models here.
class User (AbstractUser):
    ROLES=(
        ('user', 'user'),
        ('moderator', 'moderator'),
        ('admin', 'admin')
    )
    #unique email
    email=models.EmailField(unique=True)

    role=models.CharField(max_length=30, choices=ROLES, default='user')

    description=models.TextField(blank=True, default='')

```

settings.py: custom user model

```
AUTH_USER_MODEL='accounts.User'
```

Update Post model

```
from accounts.models import User

class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
        PUBLISHED = 'PB', 'Published'

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250, unique_for_date='publish')
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
```

Rollback migrations

- ▶ Delete database
- ▶ Delete all migrations
- ▶ Delete all cache files

Make and run migrations

accounts: admin.py

```
from django.contrib import admin  
  
from .models import User  
  
# Register your models here.  
  
admin.site.register(User)
```

accounts: template

▼	accounts	●
>	__pycache__	
>	migrations	●
▼	templates \ <u>registration</u>	●
	login.html	U
	signup.html	U

accounts: forms.py: register

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import get_user_model

class UserRegistrationForm(UserCreationForm):
    email=forms.EmailField(help_text='a valid email please', required=True)

    class Meta:
        model= get_user_model()
        fields=['first_name', 'last_name', 'username', 'email', 'password1','password2']

    def save(self,commit=True):
        user=super(UserRegistrationForm,self).save(commit=False)
        user.email=self.cleaned_data['email']
        if commit:
            user.save()
        return user
```

accounts: views.py: register

```
def register_view(request):
    if request.user.is_authenticated:
        return redirect(reverse('blog:post_list'))
    if request.method == "POST":
        form = UserRegistrationForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)
            return redirect(reverse('blog:post_list'))
    else:
        form = UserRegistrationForm()
    return render(
        request=request,
        template_name = "registration/signup.html",
        context={"form": form}
    )
```

accounts:signup.html

```
{% extends "blog/base.html" %}
{% block title %}Sign Up{% endblock title %}
{% block content %}
<h2>Sign Up</h2>
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Sign Up</button>
</form>
{% endblock content %}
```

accounts: views.py: login

```
from django.contrib.auth.forms import AuthenticationForm

def login_view(request):
    if request.user.is_authenticated:
        return redirect(reverse('blog:post_list'))

    if request.method == "POST":
        form = AuthenticationForm(request=request, data=request.POST)
        if form.is_valid():
            user = authenticate(
                username=form.cleaned_data["username"],
                password=form.cleaned_data["password"],
            )
            if user is not None:
                login(request, user)
                return redirect(reverse('blog:post_list'))
```

accounts: views.py: login

```
form = AuthenticationForm()

return render(
    request=request,
    template_name="registration/login.html"
,
    context={"form": form}
)
```


accounts: login.html

```
{% extends "blog/base.html" %}
{% block title %}Log-in{% endblock %}
{% block content %}
    <h1>Log-in</h1>
    <div class="login-form">
        <form action="{% url 'accounts:login' %}" method="post">
            {{ form.as_p }}
            {% csrf_token %}
            <p><input type="submit" value="Log-in"></p>
        </form>
    </div>
{% endblock %}
```

accounts: urls.py

```
from django.urls import path
from django.contrib.auth import views as auth_views
from . import views
app_name='accounts'

urlpatterns=[
    path('login/', views.login_view,name='login'),
    path('logout/', views.logout_view, name='logout'),
    path('signup/', views.register_view, name='signup')
]
```

blog_site: urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path("accounts/", include("accounts.urls", namespace="accounts")), # new
    path('blog/', include('blog.urls', namespace='blog')),
    path('dashboard/', include('dashboard.urls', namespace='dashboard')),
]
```

blog: base.html

```
<div id="content">
  {% if user.is_authenticated %}
    <p>Hi {{ user.username }}!</p>
    <p><a href="{% url 'accounts:logout' %}">Log Out</a></p>
  {% else %}
    <p>You are not logged in</p>
    <a href="{% url 'accounts:login' %}">Log In</a> |
    <a href="{% url 'accounts:signup' %}">Sign Up</a>
  {% endif %}
  {% block content %}
  {% endblock %}
</div>
```

blog: base.html

```
{% if user.is_authenticated %}  
  <div id="sidebar">  
  </div>  
{% endif %}
```

blog: detail.html

```
{% if user.is_authenticated %}
    {% include "blog/post/includes/comment_form.html" %}
{% endif %}
```

blog: views.py

```
@require_POST
@login_required
def post_comment(request, post_id):
    post = get_object_or_404(Post,
id=post_id,status=Post.Status.PUBLISHED)
    comment = None
    # A comment was posted
    form = CommentForm(data=request.POST)
```

Permissions

- ▶ Django comes with a built-in permissions system. It provides a way to assign permissions to specific users and groups of users.

<https://docs.djangoproject.com/en/4.2/topics/auth/default/#:~:text=Permissions%20and%20Authorization,it%20in%20your%20own%20code.>

Permissions: niveau modèle

blog.add_post

blog.change_post

blog.delete_post

blog.view_post

<app>.action_<model>

Permissions: niveau modèle (custom)

```
class MyModel(models.Model):
    # fields and methods

    class Meta:
        permissions = [
            ("can_do_something", "Can do something"),
            ("can_view_something", "Can view something"),
            # ...
        ]
```

Object-Level Permissions

```
class MyModel(models.Model):
    # fields and methods

    def can_edit(self, user):
        # Custom logic to check if the user can edit this instance
        return user == self.author

    def can_view(self, user):
        # Custom logic to check if the user can view this instance
        return self.is_published or user == self.author
```

Assigning Permissions to Users and Groups

```
# Assigning permissions to a user
user = User.objects.get(username="nizar")
permission = Permission.objects.get(codename="can_do_something")
user.user_permissions.add(permission)
```

```
# Assigning permissions to a group
group = Group.objects.get(name="developers")
group.permissions.add(permission)
```

Checking Permissions in Views

```
from django.contrib.auth.decorators import permission_required

@permission_required("myapp.can_do_something")
def my_view(request):
    # View logic
```

Checking Permissions in Templates

```
{% if perms.myapp.can_do_something %}
    <!-- Display content for users with the 'can_do_something' permission -->
{% endif %}
```

Permissions: niveau group

