

# TP : Dashboard Météo avec Django et MQTT

---

## 1 Introduction

Django est un Framework Python, open-source et gratuit qui permet de créer un site web. Il facilite le processus de développement en fournissant tous les outils nécessaires pour créer des sites web dynamiques.

### 1.1 Objectifs de l'activité

À la fin de cette activité, vous serez capable de :

- Installer et configurer Django
- Comprendre l'architecture MVT de Django
- Créer des modèles de données

## 2 Installation et Configuration Initiale

### 2.1 Préalables

Installez d'abord :

- Python 3.8 ou supérieur
- pip
- Un broker MQTT (pour les tests utilisez test.mosquitto.org)
- Un éditeur de code (VS Code)

### 2.2 Installation de Django et dépendances

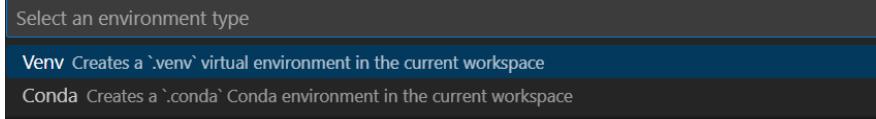
#### 2.2.1 Importer le projet

```
git clone https://github.com/a-bulcke/django-meteo.git
cd django_meteo
```

#### 2.2.2 Crée un environnement virtuel

```
python -m venv env
```

Sous VSCode tapez shift+Ctrl+P puis :



Cet environnement protège votre projet des conflits avec d'autres bibliothèques Python.

#### 2.2.3 Activer l'environnement

```
# Sur Windows :
.venv\Scripts\Activate.ps1
# Sur macOS/Linux :
source env/bin/activate
```

Avec VScode l'environnement virtuel est automatiquement sélectionné. Il suffit d'ouvrir un terminal dans le menu Affichage.

Vous remarquerez que le nom de l'environnement apparaît entre parenthèses dans votre terminal.

```
PS C:\Users\Arnaud\Documents\Python\django> & C:/Users/Arnaud/Documents/Python/django/.venv/Scripts/Activate.ps1  
(.venv) PS C:\Users\Arnaud\Documents\Python\django>
```

## 2.2.4 Installer les dépendances

```
pip install requirements.txt
```

Attendez que l'installation se termine, puis vérifiez que Django est correctement installé en tapant :

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

## 2.3 Projet Django meteo\_dashboard

### 2.3.1 Structure du projet

Lorsque qu'un projet Django est généré (voir [Comment créer un projet Django](#)), une structure de dossiers complète avec tous les fichiers de configuration nécessaires est créée.

Ci-dessous voici la structure du projet **meteo dashboard** :

```
meteo_dashboard/
└── meteo_dashboard/
    ├── settings.py
    ├── urls.py
    └── asgi.py
meteo/
└── models.py
└── views.py
└── urls.py
└── management/
    └── commands/
        └── mqtt_client.py
└── templates/
    └── meteo/
        ├── base.html
        ├── dashboard.html
        └── statistiques.html
static/
└── meteo/
    ├── css/
    │   ├── style.css
    │   └── dashboard.css
    └── js/
        └── charts.js
└── admin.py
└── mqtt_client.py
└── manage.py
└── db.sqlite3
└── .env
```

# Configuration du projet  
# Paramètres du projet  
# Routage principal

# Application météo  
# Modèles de données  
# Logique métier  
# Routage de l'app  
# commandes personnalisées

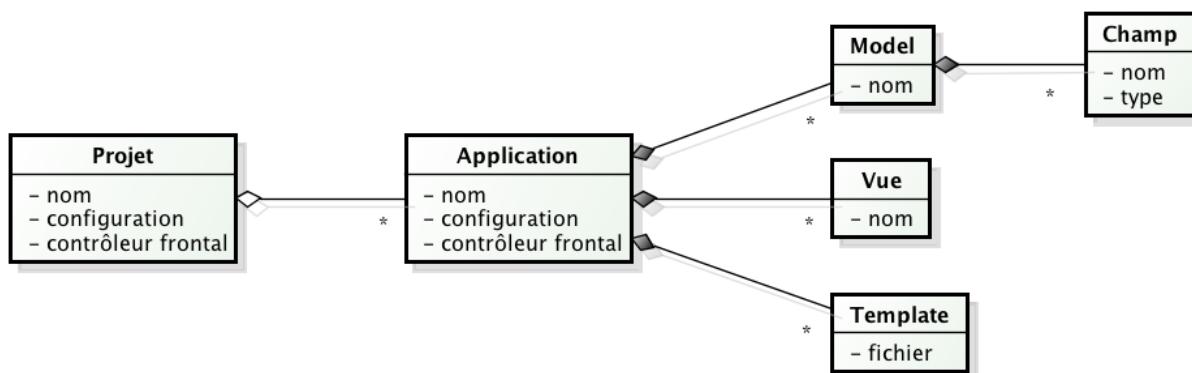
# Templates HTML

# Fichiers CSS, JS, images

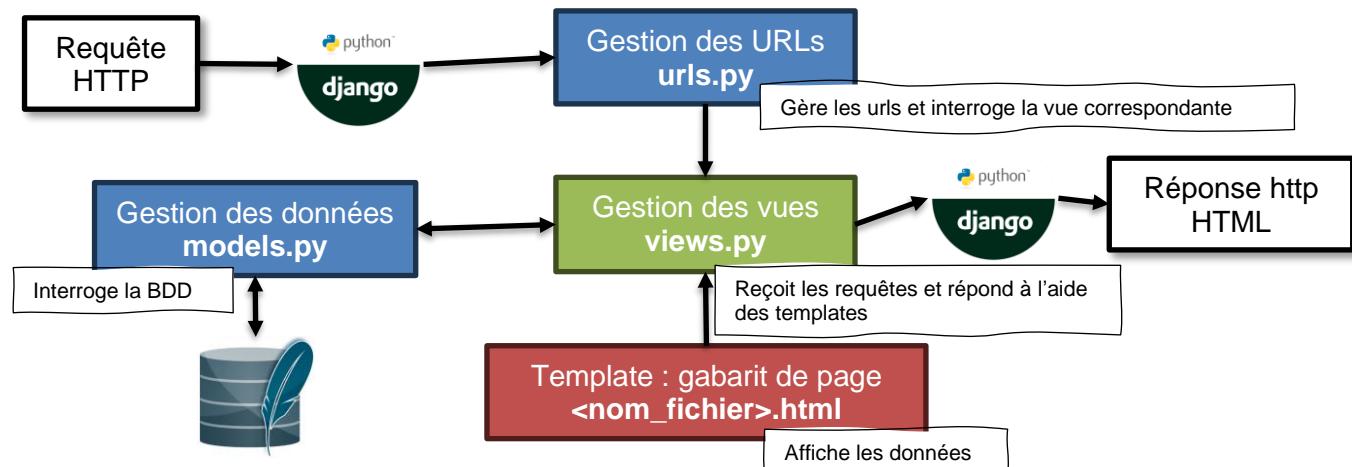
# Configuration admin  
# Script MQTT  
# Outil de gestion Django  
# Base de données  
# Variables d'environnement

### 3 Principes de fonctionnement de Django

### 3.1 Ecosystème Django



### 3.2 Modèle MVT (Modèle-Vue-Template)



## 4 Exécution du Projet

### 4.1 Créer un superutilisateur pour l'administration

```
python manage.py createsuperuser
```

Utilisez un nom et un mot de passe que vous n'oublierez pas. Par exemple admin pour le login. En cas d'oubli, vous pourrez toujours relancer la commande ci-dessus pour créer un nouveau superutilisateur avec un nom différent.

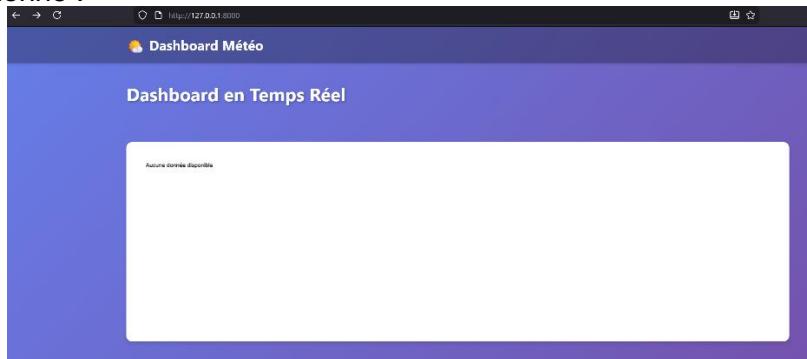
### 4.2 Démarrer le serveur Django

```
python manage.py runserver
```

Accédez à :

- Dashboard: <http://localhost:8000/>
- Admin: <http://localhost:8000/admin/>
- Statistiques: <http://localhost:8000/statistiques/>

Vérifiez que tout fonctionne :



Si vous souhaitez arrêter le serveur, faire un CTRL+C.

## 5 Concepts Clés de Django

### 5.1 Architecture MVT

Django utilise le pattern **MVT** (Modèle-Vue-Template) :

Composant	Rôle
Modèle (M)	Structure des données (modèle de base de données)
Vue (V)	Logique métier (traitement des requêtes)

Composant	Rôle
<b>Template (T)</b>	Présentation HTML (interface utilisateur)

### 5.1.1 Flux de requête HTTP :

1. Utilisateur accède à une URL (ex: /statistiques/)  
↓
2. Django cherche l'URL dans le fichier `urls.py`  
↓
3. Django appelle la vue correspondante (view) : fichier `views.py`  
↓
4. La vue interroge le modèle (base de données) : fichier `models.py`  
↓
5. La vue rend le template avec les données : fichiers `.html` du dossier `templates`  
↓
6. Django retourne le HTML au navigateur

### 5.2 ORM Django

L'ORM (Object Relational Mapper) permet de manipuler la base de données sans SQL :

```
# Au lieu de :
# SELECT * FROM weather_mesure WHERE temperature > 25

# Vous écrivez :
mesures = Mesure.objects.filter(temperature__gt=25)
```

Aide en ligne : <https://python.doctor/page-django-query-set-queryset-manager>

### 5.3 Migrations

Les migrations sont la manière par laquelle Django propage des modifications que vous apportez à des modèles (ajout d'un champ, suppression d'un modèle, etc.) dans un schéma de base de données (fichier `models.py`) :

```
python manage.py makemigrations # Créer les migrations
python manage.py migrate # Appliquer les migrations
```

## 6 Configuration de la Base de Données

Django propose une architecture bien structurée : séparation entre la logique métier (modèles), la présentation (templates) et le contrôle (vues). Cette organisation facilite la maintenance et l'évolution de votre code.

Le fichier de configuration `meteo_dashboard/settings.py` contient les paramètres de configuration. Vérifiez que l'application créée "meteo" est indiquée dans `INSTALLED_APPS` ainsi que la configuration pour la langue et le fuseau horaire. Le dossier où seront stockés les fichiers statiques (feuilles de style et javascript) doit être indiqué également :

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

# Base de données SQLite
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

```
# Fuseau horaire
TIME_ZONE = 'Europe/Paris'
USE_TZ = True

# Langue
LANGUAGE_CODE = 'fr-fr'

# Configuration des fichiers statiques
STATIC_URL = '/static/'
STATICFILES_DIRS = [BASE_DIR / 'weather' / 'static']
```

Pour sauvegarder les paramètres importants (connexion MQTT et communication sécurisée avec votre application Django par l'intermédiaire d'une SECRET-KEY), un fichier fichier **.env** est utilisé :

```
MQTT_BROKER=localhost
MQTT_PORT=1883
MQTT_USERNAME=user
MQTT_PASSWORD=password
DJANGO_SECRET_KEY=your-secret-key-here
```

La clé DJANGO\_SECRET\_KEY est une clé de sécurité secrète utilisée par Django pour :

1. **Chiffrer les données sensibles** : Sessions utilisateur, tokens CSRF, mots de passe réinitialisés, etc.
2. **Signer les données** : Pour s'assurer que les données n'ont pas été modifiées
3. **Générer des tokens de sécurité** : Pour les formulaires et les sessions

**Pourquoi c'est important :**

- Si quelqu'un connaît votre SECRET\_KEY, il peut forger des sessions, contourner les protections CSRF, et accéder aux données chiffrées
- Elle **ne doit jamais être exposée** publiquement (GitHub, serveurs, etc.)
- Elle doit être **unique et aléatoire**

La DJANGO\_SECRET\_KEY pourra être générée par :

```
python -c 'from django.core.management.utils import get_random_secret_key; print(get_random_secret_key())'
```

- 1) Chercher à quoi sert un token CSRF.
- 2) Créer votre SECRET\_KEY, modifier le fichier **.env**.

## 7 Publication des températures

Utiliser MQTTEexplorer par exemple pour publier des températures sur votre broker sur le topic **meteo/temperature**.

Vous pouvez utiliser **test.mosquitto.org** pour faire des tests ou le broker du lycée (**172.21.28.1**).

Pour que le site Django place les températures dans la base de données, nous allons utiliser un script Python utilisant Paho-MQTT pour souscrire au topic meteo/temperature et inscrire les nouvelles mesures dans la base de données. Dans un autre terminal, lancer le script **mqtt\_client.py** (cf. 12.1) :

```
python manage.py mqtt_client
```

- 3) Publier plusieurs températures et observer le fonctionnement du site.
- 4) Comment réagit la page d'accueil (dashboard) ?
- 5) Quelles informations sont affichées sur la page statistiques ?
- 6) Quelles informations sont visibles sur la page d'administration ?



## 8 Modèles de Données

Pour traduire cette architecture de base de données en code Django, nous définissons des modèles qui représentent les données.

Un modèle est donc une classe python qui hérite de la classe **models.Model**. Les champs sont définis dans la classe, on leur donne un nom et un type.

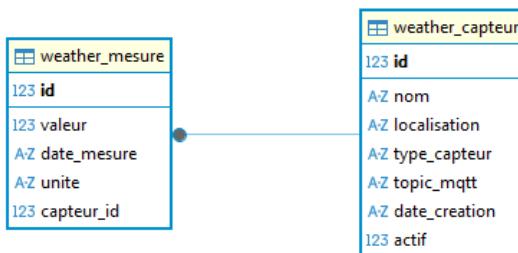


Figure 1 : BDD à obtenir

Ouvrir un terminal, aller dans le dossier **/meteo\_config** puis taper :

```
python manage.py shell
```

Importer les modèles :

```
>>> from meteo.models import *
```

Afficher tous les enregistrements de la table Mesure :

```
>>> Mesure.objects.all()
```

**7) A quoi sert la classe modèle ? Que fait la méthode all() ?**

Filter les mesures :

```
>>> Mesure.objects.filter(valeur__gt=20)
```

**8) Que fait la méthode filter ? A quoi sert \_\_gt=20 ?**

Aller sur la page d'administration (relancer le serveur si nécessaire) : <http://127.0.0.1:8000/admin>

Créez éventuellement un superuser pour l'accès :

```
python manage.py createsuperuser
```

Afficher la page des mesures : <http://127.0.0.1:8000/admin/weather/mesure/>

Ajouter une mesure inférieure à 0°C

**9) Ecrire la requête permettant d'afficher les valeurs des mesures inférieures à 0°C. Vérifier que votre mesure est visible.**

```
>>> quit()
```

Ouvrir **meteo/models.py**. Il y a une classe par table de la BDD :

```
from django.db import models

class Capteur(models.Model):
    """Modèle pour les capteurs"""
    nom = models.CharField(max_length=100, unique=True)
    type_capteur = models.CharField(
```

```

max_length=20,
choices=[
    ('temperature', 'Température'),
    # TODO : Ajouter les choix 'pression' et 'humidité'
]
)
topic_mqtt = models.CharField(max_length=100)
date_creation = models.DateTimeField(auto_now_add=True)

def __str__(self):
    return f"{self.nom} - {self.get_type_capteur_display()}"


class Meta:
    verbose_name = "Capteur"
    verbose_name_plural = "Capteurs"

class Mesure(models.Model):
    """Modèle pour les mesures des capteurs"""
    capteur = models.ForeignKey(Capteur, on_delete=models.CASCADE)
    valeur = models.FloatField()
    date_mesure = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"{self.capteur.nom}: {self.valeur}{self.unite}"

    class Meta:
        verbose_name = "Mesure"
        verbose_name_plural = "Mesures"
        ordering = ['-date_mesure']

```

10) Ajouter le champ ***localisation*** qui contiendra le texte du lieu du capteur (type CharField de 200 caractères maxi) dans la table Capteur

11) Ajouter le champ ***actif***(type booléen, vrai par défaut)

12) Dans la table Mesure, ajouter le champ ***unites*** de type texte (longueur max 20 caractères) et qui contiendra uniquement le choix possible pour le capteur de température : 'C' pour "C".

Si une modification de models.py est faite, il faut appliquez les migrations (cf. 5.3) :

```
python manage.py makemigrations
python manage.py migrate
```

## 9 Vues Django

Dans Django, une vue représente la logique qui traite les requêtes des utilisateurs et retourne des réponses. Ouvrir ***meteo/views.py*** de votre application :

```

from django.shortcuts import render
from django.http import JsonResponse
from django.views.decorators.http import require_http_methods
from django.utils import timezone
from datetime import timedelta
from .models import Capteur, Mesure
import json

def dashboard(request):
    """Affiche le dashboard avec les données actuelles"""
    capteurs = Capteur.objects.all()

    # Récupérer la dernière mesure de température
    mesure_temp = Mesure.objects.filter(capteur__type_capteur='temperature').order_by('date_mesure').first()

    # TODO : Récupérer les dernières mesures de pression et humidité

    mesures_actuelles = {}
    if mesure_temp:
        mesures_actuelles['temperature'] = {
            'valeur': mesure_temp.valeur,
            'date': mesure_temp.date_mesure,
        }

    # TODO : Ajouter les mesures de pression et humidité au dictionnaire

```

```

context = {
    'capteurs': capteurs,
    'mesures_actuelles': mesures_actuelles,
}
return render(request, 'weather/dashboard.html', context)

def statistiques(request):
    """Affiche les statistiques sur les dernières 24 heures"""
    depuis = timezone.now() - timedelta(hours=24)

    stats = {}

    # Statistiques pour la température
    mesures_temp = Mesure.objects.filter(
        capteur__type_capteur='temperature',
        date_mesure__gte=depuis
    ).values_list('valeur', flat=True)

    if mesures_temp:
        stats['temperature'] = {
            'min': min(mesures_temp),
            'max': max(mesures_temp),
            'nombre': len(mesures_temp),
        }

    # TODO : Ajouter les statistiques pour pression (type_capteur='pression')
    # Utiliser la même structure que pour la température
    # stats['pression'] = { ... }

    # TODO : Ajouter les statistiques pour humidité (type_capteur='humidite')
    # Utiliser la même structure que pour la température
    # stats['humidite'] = { ... }

    context = {
        'stats': stats,
    }
    return render(request, 'weather/statistiques.html', context)

@require_http_methods(["GET"])
def mesures_json(request, type_capteur):
    """Retourne les mesures au format JSON pour les graphiques"""

    # Vérifier que le type_capteur est valide
    types_valides = ['temperature'] # TODO : Ajouter 'pression' et 'humidite'

    if type_capteur not in types_valides:
        return JsonResponse({'erreur': 'Type de capteur invalide'}, status=400)

    # Récupérer les mesures des dernières 24 heures
    depuis = timezone.now() - timedelta(hours=24)

    mesures = Mesure.objects.filter(
        capteur__type_capteur=type_capteur,
        date_mesure__gte=depuis
    ).order_by('date_mesure').values(
        'valeur', 'date_mesure', 'capteur__nom'
    )

    donnees = [
        {
            'timestamp': m['date_mesure'].isoformat(),
            'valeur': m['valeur'],
            'capteur': m['capteur__nom'],
        }
        for m in mesures
    ]

    return JsonResponse({'donnees': donnees})

```

Afficher <http://127.0.0.1:8000/>

Dans la vue **dashboard** (`def dashboard(request)`): ci-dessus), il faut afficher les dernières mesures.

13) Changer la requête pour obtenir la dernière mesure.

14) Ajouter la gestion de l'unité.

Dans la vue **statistiques**, il faut afficher, en plus des valeurs min et max, la valeur moyenne et le nombre de mesure.

Afficher <http://127.0.0.1:8000/statistiques>

15) Ajouter l'affichage de la valeur moyenne.

16) Modifiez la variable capteurs pour prendre en compte que les capteurs actifs.

## 10 Routage (URLs)

Ouvrir le fichier **meteo/urls.py**:

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

urlpatterns = [
    path('', views.dashboard, name='dashboard'),
    path('api/mesures/<str:type_capteur>', views.mesures_json, name='mesures_json'),
    path('statistiques/', views.statistiques, name='statistiques'),
]
```

17) Quelle url utiliser pour afficher les mesures de température au formats json ? Vérifiez.

Ouvrir **meteo\_dashboard/urls.py** qui permettra de prendre en compte ce fichier :

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

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

18) Quelle ligne permet de définir la page d'administration ?

19) Changer pour accéder à l'interface d'administration par l'url

<http://127.0.0.1:8000/administration>

## 11 Templates

Les templates permettent la présentation visuelle de votre site :

- fichier qui contient des variables et des tags, et qui sert à générer le document final.
- on peut l'utiliser pour générer du html, du csv ou n'importe quel autre fichier basé sur du texte.
- les variables sont évaluées par `{% myVar %}`.
- les tags sont à un format du type `{% myTag ... %}`.

### 11.1 base.html

Ouvrir le fichier **meteo/templates/meteo/base.html**

```
{% load static %}
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Dashboard Météo{% endblock %}</title>
    <link rel="stylesheet" href="{% static 'meteo/css/style.css' %}">
    {% block extra_css %}{% endblock %}
</head>
<body>
    <nav class="navbar">
        <div class="container">
            <span class="navbar-brand"><img alt="Dashboard Météo logo" /> Dashboard Météo</span>
        </div>
    </nav>

    <div class="container">
        {% block content %}{% endblock %}
    </div>

```

```
</div>

<script src="https://cdn.jsdelivr.net/npm/chart.js@3.9.1/dist/chart.min.js"></script>
<script src="{% static 'weather/js/charts.js' %}"></script>
{% block extra_js %}{% endblock %}

</body>
</html>
```

La page **base.html** sert de base aux pages **dashboard.html** et **statistiques.html**. Les blocs **extra\_css** (délimité par le tag `{% block extra_css %}{% endblock %}`), **content** et **extra\_js** sont donc implémentés dans **base.html** mais seront définis différemment dans les pages dashboard.html et statistiques.html.

20) Changer le texte de la balise **navbar-brand** (vous pouvez utiliser

[https://www.w3schools.com/charsets/ref\\_emoji\\_weather.asp](https://www.w3schools.com/charsets/ref_emoji_weather.asp) pour trouver d'autres icônes)

Commenter les lignes `{% load static %}` et `<link rel="stylesheet" href="{% static 'meteo/css/style.css' %}">` (en mettant entre balises `<!-- -->`)

21) A quoi sert le tag `{% load static %}` ?

22) Comment est chargé la feuille de style ?

Commenter la ligne `{% block extra_css %}{% endblock %}`

23) Où se trouve la feuille de style concernée ?

## 11.2 dashboard.html

Ouvrir **meteo/templates/meteo/dashboard.html** :

```
{% extends 'meteo/base.html' %}
{% load static %}

{% block title %}Dashboard - Météo{% endblock %}

{% block extra_css %}
    <link rel="stylesheet" href="{% static 'meteo/css/dashboard.css' %}">
{% endblock %}

{% block content %}
<h1>Dashboard en Temps Réel</h1>


<div class="metrics-grid">
    {% if mesures_actuelles.temperature %}
        <div class="metric-card">
            <div class="metric-icon">🌡</div>
            <div class="metric-label">Température</div>
            <div class="metric-value">
                {{ mesures_actuelles.temperature.valeur|floatformat:1 }}°C
            </div>
            <small class="metric-date">
                Mise à jour: {{ mesures_actuelles.temperature.date|date:"H:i:s" }}
            </small>
        </div>
    {% endif %}
</div>


<div class="charts-grid">
    <div class="chart-container">
        <canvas id="temperatureChart"></canvas>
    </div>
</div>
{% endblock %}
```

24) Que signifie le tag `{% if mesures_actuelles.temperature %}`

25) Dans quel fichier cette variable a-t-elle été définie ?

## 12 Configuration Admin Django

Ouvrir `meteo/admin.py`:

```
from django.contrib import admin
from .models import Capteur, Mesure

@admin.register(Capteur)
class CapteurAdmin(admin.ModelAdmin):
    list_display = ('nom', 'type_capteur', 'date_creation')
    list_filter = ('type_capteur', 'date_creation')
    search_fields = ('nom',)
    readonly_fields = ('date_creation',)

@admin.register(Mesure)
class MesureAdmin(admin.ModelAdmin):
    list_display = ('capteur', 'valeur', 'date_mesure')
    list_filter = ('capteur__type_capteur', 'date_mesure')
    search_fields = ('capteur__nom',)
    readonly_fields = ('date_mesure',)
    date_hierarchy = 'date_mesure'
```

26) Modifier `admin.py` pour prendre en compte l'affichage des champs ajoutés précédemment dans `models.py`

### 12.1 Démarrer le client MQTT

Afin de lancer le script Python `mqtt_client.py` situé dans le dossier `meteo` il faut utiliser une commande Django. Ce programme est placé dans le dossier `meteo/management/commands/` et se nomme `mqtt_client.py`:

```
from django.core.management.base import BaseCommand
from meteo.mqtt_client import meteoMQTTClient

class Command(BaseCommand):
    help = 'Démarrer le client MQTT pour collecter la température'

    def handle(self, *args, **options):
        client = meteoMQTTClient()
        client.start()
```

Pour le lancer, utilisez le nom du fichier :

```
python manage.py mqtt_client
```

## 13 Test avec MQTTEexplorer

### 13.1 Publier des données de test



## 14 EXERCICES : Ajouter la Pression et l'Humidité

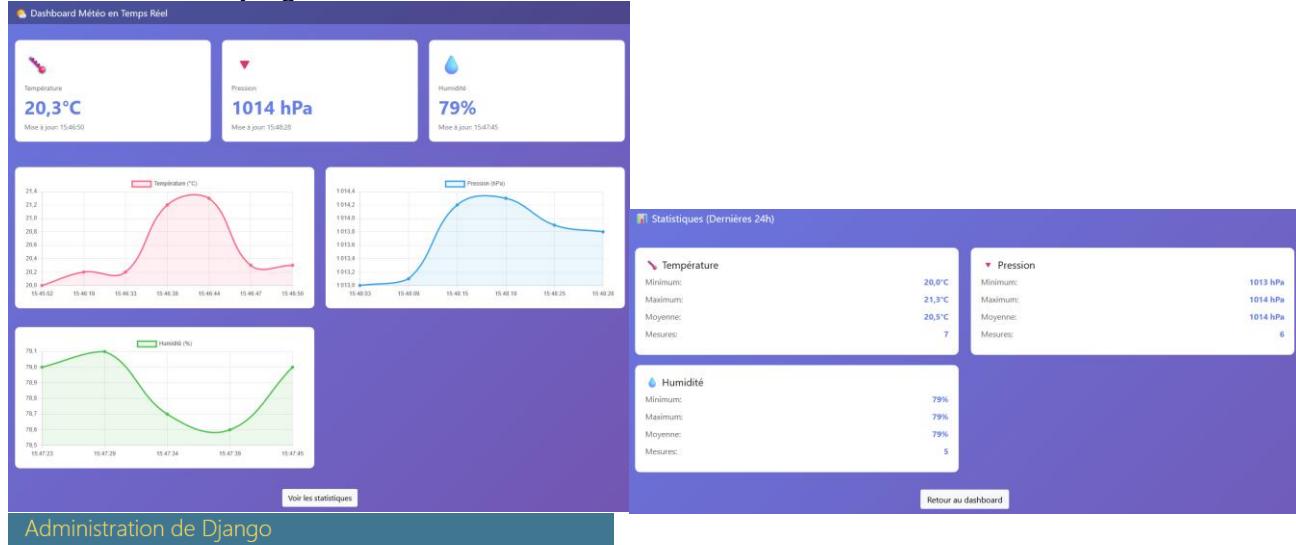
27) Modifiez les fichiers suivants pour ajouter support pour la pression et l'humidité :

1. **`models.py`** : Ajouter 'pression' et 'humidite' aux choix du type\_capteur et unite
2. **`views.py`** : Ajouter le traitement pour pression et humidité (remplacer les TODO)
3. **`urls.py`** : Ajouter les routes si nécessaire
4. **`dashboard.html`** : Ajouter les cartes pour pression et humidité

5. **dashboard.css** : Adapter le CSS si nécessaire
6. **charts.js** : Ajouter les appels createChart pour les deux nouveaux graphiques
7. **mqtt\_client.py** : Ajouter les configurations pour les topics MQTT (décommenter la ligne pour prendre en compte l'unité de la température)

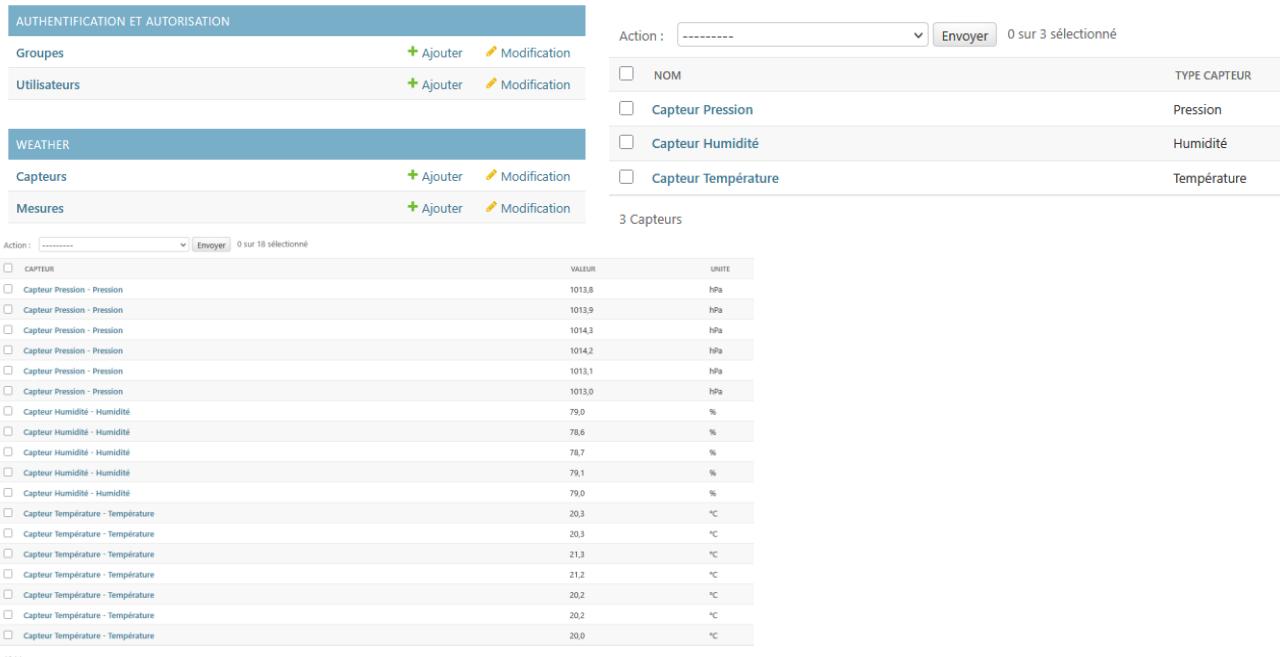
### Topics MQTT à utiliser :

- meteo/pression → valeur en hPa
- meteo/humidite → valeur en %
- Et sur votre site Django :



Administration de Django

Site d'administration



The admin interface includes:
 

- AUTHENTIFICATION ET AUTORISATION:** Groups and Users management.
- WEATHER:** Capteurs (Sensors) and Mesures (Measurements) management.
- Capteurs List:** A table listing 18 selected sensors with columns: Action, NOM, TYPE CAPTEUR.
- Mesures Table:** A table listing 18 measurements with columns: Action, CAPTEUR, VALEUR, UNITE.

## 15 Comment créer un projet Django

### 15.1 Créer le projet

```
django-admin startproject nom_du_projet
cd nom_du_projet
```

### 15.2 Créer l'application

```
python manage.py startapp nom_de_l_application
```