Django

Le Framework Django

1 Historique

— Développé à l'origine à partir de 2003 pour un journal local de la ville de Lawrence dans le Kansas par Adrian Holovaty et Simon Willison
— But : Réaliser une sorte de CMS, simple à utiliser pour les non informaticiens
— En Open Source sous Licence BSD depuis 2005
— Beaucoup d'évolutions depuis
— Nommé ainsi en souvenir de Django Reinhardt, musicien de Jazz

4.1.2 Caractéristiques

— Framework Web de référence
— ORM, Templates
— Cache
— Migrations (intégrées tardivement, l'un des meilleurs système de migrations existant)
— Formulaires, Vues génériques
— Authentification
— Internationalisation, Unicode
— Gestion complète des exceptions
— Bonne documentation

En bref : *Batteries included*

4.2 Framework Web généraliste offrant

— MVT = Modèle Vue Template
— Système de templates
— ORM = Object Relational Mapper (comme SQLAlchemy ou Doctrine)
— Serveur Web intégré
— Interface d'Admin complète, souple et extensible

4.3 Versions

— OpenSource (BSD) en 2005
— Version 1.0 en 2008
— Version 1.4 LTS en 2012
— Version 1.8 LTS en 2015
— Actuelle (2020) : 3.1
— Version 1.11 est la dernière avec le support Python 2
— La distribution django 2.2 en 2019 est LTS et la version actuelle est la 3.1.1
— La prochaine release LTS de django sera la 3.2 qui sortira en 2021

4.4 Alternatives

En Python les alternatives sont notamment :
— Flask (microframework avec Jinja2 pour les templates et SQLAlchemy pour ORM)
— Pyramid
— Tornado (orienté programmation asynchrone et Web Sockets)
— FastAPI pour construire rapidement une API en Python

4.5 Installation Django

Indication : Il faut savoir utiliser un virtualenv en Python …

4.5.1 Virtualenv

— Environnement virtuel Python
— Permet d'installer ses propres paquets
— Peut ou non utiliser les libs présentes dans le système
— Permet de fixer et de restaurer l'environnement logiciel nécéssaire à un projet
— Habituellement, 1 virtualenv par projet
— Pas besoin sauvegarder le venv ou de le versionner, juste conserver les *requirements* qui contiennent la liste des dépendances à installer

Attention : On ne déplace jamais un venv sinon il ne fonctionnera plus !

L'interface administration de django est une application au même titre que celle que vous êtes en train de créer

4.5.2 Création

```
virtualenv -p python3 myenv
```

ou plus rarement avec les librairies déjà installées sur votre système :

```
virtualenv --system-site-packages -p python3 myenv
```

4.5.3 Activation virtualenv

```
source myenv/bin/activate
```

4.5.4 Avec conda

Si vous avez installé python avec anaconda, l'outil conda sera installé et on crée l'équivalent du venv avec :

```
conda create -n myenv python=3.7 python
```

4.5.5 Activer un environnement conda

```
conda activate myenv
```

4.5.6 Installation de Django

Dans tous les cas on installe Django avec pip :

```
pip install  django
```

4.6 Versionnage des requirements

4.6.1 Initialiser le versionnage avec git et premiers commits

Vérifiez que vous avez déjà fait les réglages de base de git (user, email, editeur) :

```
git config --list
```

Vous devriez avoir user.name, user.email et core.editor de configurés. Sinon :

```
git config --global user.name "Alice Torvalds"
git config --global user.email "alice@linux.org"
git config --global core.editor emacs
```

Indication : Vous pouvez choisir un autre éditeur qu'emacs : atom, vscode, vim, etc.

Pour choisir vscode :

```
editor = code --wait
```

4.6.2 .gitignore

On se place à la racine du projet , on y ajoute un fichier .gitignore contenant :

```
__pycache__
local_settings.py
```

Le fichier *.gitignore* contient les fichiers ou dossiers qui doiventéchapper au versionnage. Ici nous indiquons que tous les dossiers *__pycache__* doivent être ignorés ainsi que le fichier *local_settings.py* qui pourra contenir des informations sensibles (comme des logins et passwd de BD qui n'ont pas à être versionnés).

4.6.3 Puis on initie un dépôt git

```
git init
git add .
```

Attention : On ne versionne pas le virtualenv (ou conda) Python. Il est propre à chaque environnement et on ne le change pas d'emplacement une fois qu'il est créé !

On peut extraire les dépendances de notre projet (pour l'instant essentiellement une version spécifique de django) avec la sous-commande *freeze* de pip :

4.6.4 requirements

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

On stocke ces dépendances dans le fichier *requirements.txt* qu'il est bon de versionner :

```
git add requirements.txt
git commit -m "ajout requirements.txt au projet"
```

4.7 Création du projet de base

4.7.1 La commande django-admin

Permet de créer un projet, une application ou d'inspecter un projet existant (base de données ,etc)

```
django-admin startproject GestionTaches
```

Puis consultons l'arborescence du projet créé :

```
tree GestionTaches
```

qui donne :

```
GestionTaches/
|-- GestionTaches
|   |-- __init__.py
|   |-- __pycache__
|   |
|   |-- settings.py
|   |-- urls.py
|   `-- wsgi.py
|-- db.sqlite3
|-- manage.py
```

4.7.2 sous-commandes de django-admin

Sans arguments, donne les sous-commandes disponibles :

```
Type 'django-admin help <subcommand>' for help on a specific␣
↪subcommand.

 Available subcommands:
```

(suite sur la page suivante)

(suite de la page précédente)

```
[django]
    check
    compilemessages
    createcachetable
    dbshell
    diffsettings
    dumpdata
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    runserver
    sendtestemail
    shell
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver

    Note that only Django core commands are listed as settings are␣
↪not properly configured (error: Requested setting INSTALLED_APPS,␣
↪but settings are not configured. You must either define the␣
↪environment variable DJANGO_SETTINGS_MODULE or call settings.
↪configure() before accessing settings.).
```

Danger : Attention, django-admin et manage.py ne sont pas strictement identiques même si ils sont très semblables. django-admin se charge des tâches administratives et manage.py est créé dans chaque projet que vous réalisez et assure certaines tâches automatiquement.

4.7.3 Lancement du serveur

```
./manage.py runserver
```

4.7.4 Test du serveur

Localhost:8000 (http ://localhost :8000)

4.7.5 Modif .gitignore

La base de données n'a pas normalement vocation à être versionnée. On va donc l'ajouter au .gitignore et commiter.

```
echo db.sqlite3 >> .gitignore
git commit -am "ajout fichier db au .gitignore"
```

4.7.6 Création d'une app lesTaches dans le projet

```
django-admin startapp lesTaches
```

Puis observons l'arborescence obtenue :

```
tree GestionTaches
```

qui donne :

```
GestionTaches/
|-- GestionTaches
|   |-- __init__.py
|   |-- __pycache__
|   |-- settings.py
|   |-- urls.py
|   `-- wsgi.py
|-- db.sqlite3
|-- manage.py
|-- lesTaches
|   |-- __init__.py
|   |-- admin.py
|   |-- apps.py
|   |-- migrations
|   |   `-- __init__.py
|   |-- models.py
|   |-- tests.py
|   `-- views.py
```

Commitons

4.8 Configuration

Editez le fichier *settings.py* pour y configurer

- l'internationalisation
- le format de Date par défaut
- et ajouter l'application lesTaches aux applications installées

En particulier modifier ou ajouter les lignes suivantes si besoin :

```
LANGUAGE_CODE = 'fr-fr'
TIME_ZONE = 'Europe/Paris'
USE_I18N = True
USE_L10N = True
USE_TZ = True
DATE_INPUT_FORMATS = ('%d/%m/%Y','%Y-%m-%d')
```

Puis ajouter notre nouvelle app aux applications chargées par défaut :

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

4.9 les URLs et les Vues

Pour faire fonctionner notre application, nous avons besoin de configurer au moins une *route* (URL) qui va déclencher un comportement. Les routes se configurent dans les fichiers *urls.py*, tout d'abord celui de l'app principale, puis dans les fichiers *urls.py* des autres apps.

4.9.1 Les fichiers urls.py

Editons tout d'abord le fichier principal *GestionTaches/urls.py* et ajoutons la ligne suivante dans urlpatterns :

```
path('lesTaches/',include('lesTaches.urls'))
```

Vous aurez un petit *import* à ajouter :

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

Anciennenment, on utilisait la fonction url :

```
url(r'^lesTaches/',include('lesTaches.urls'))
```

qui est dépréciée et remplacée par *django.urls.path()* . La fonction *django.urls.re_path()* sert pour la gestion d'urls plus complexes à l'aide d'expressions régulières.

4.9.2 Fichier urls.py dans l'App lesTaches

Créons ensuite dans le répertoire *GestionTaches/lesTaches* le fichier *urls.py* de notre application :

```
from django.urls import path
from . import  views
urlpatterns=[
  path('home',views.home,name='home'),
]
```

4.10 Les views

Les Views ou Vues correspondent en gros aux « contrôleurs » dans une application MVC, c'est à dire qu'elles vont faire généralement le lien entre le modèle et les templates d'affichage en jouant le rôle de « chef d'orchestre » ou d'aiguillage.

Dans le répertoire *GestionTaches/lesTaches* il va falloir ajouter dans le fichier *views.py* la fonction *home()* suivante :

```
from django.http import HttpResponse
def home(request):
  return HttpResponse('bonjour à tous')
```

Relancer le serveur (si vous l'avez stoppé) et dans le navigateur, pointez sur l'URL : localhost:8000 (http ://localhost :8000) .

Le message "Bonjour à tous" devrait apparaître ...

Compliquons légèrement les choses pour pouvoir router des urls du type http://localhost:8000/lesTaches/home/toto.

Modifions tout d'abord le fichier *urls.py* de *lesTaches* de la façon suivante :

```
path('home/<name>',views.home,name='home'),
```

Ceci signifie que les urls du type http://localhost:8000/lesTaches/home/toto seront routées vers notre vue.

Modifiez maintenant la fonction home() avec le profil suivant (à vous de la compléter) :

```
from django.http import HttpResponse

def home(request,name):
  return HtppResponse(...)
```

du coup les urls du type localhost:8000/lesTaches/home/toto (http ://localhost :8000/lesTaches/home/toto) seront aussi routées vers notre vue.

Testez différentes urls en modifiant le paramètre fourni à path.

Comment faire en sorte que l'URL localhost:8000/lesTaches/home/ (http ://localhost :8000/lesTaches/home/) soit également routée ?

Réponse pour la view :

```
from django.http import HttpResponse


def home(request, name):
    return HttpResponse("Bonjour depuis Django  " + name)
```

4.11 Le modele de «lesTaches» : class Task

Une tâche sera définie par :

- un nom
- une description
- une date de création
- la date à laquelle la tache devra être terminée (due_date)
- la date à laquelle vous aimeriez vous mettre à travailler sur cette tache (schedule_date)

— un booleen closed qui indique si la tâche est terminée ou pas

Pour cela, dans le répertoire de votre application lesTaches, complétez le fichier *models.py* dans lequel vous définirez la classe Task héritant de la classe *models.Model* de django.db

```
from django.db import models

class Task(models.Model):
        name = models.CharField(max_length=250)
        description = models.TextField()
        created_date = models.DateField(auto_now_add=True)
        closed = models.BooleanField(default=false)


        def __str__(self):
        return self.name

# classe à compléter
```

4.11.1 prise en compte du modèle : migrations

Utilisons la commande manage.py pour prendre en compte ce nouveau modèle et observons au passage le SQL correspondant :

```
python manage.py makemigrations lesTaches
python manage.py sqlmigrate lesTaches 0001
python manage.py migrate
```

4.11.2 shell de django

Pour aller dans le shell de django :

```
python manage.py shell
```

Et effectuez-y les commandes suivantes :

```
>>> from lesTaches.models import Task # importer la class Task
>>> Task.objects.all() # lister la liste des taches créées
[]
```

pour l'instant il n'y a aucune tache enregistrée ...

Créons à présent une tâche :

```
>>> tache=Task()
>>> tache.name='une premiere tache'
>>> tache.description='la première mais pas la derniere'
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> #  on enregistre
>>> tache.save()
```

Vérifions qu'elle est bien enregistrée :

```
>>> tache=Task.objects.all()[0]
```

Puis :

- Vérifiez que les champs que vous avez renseignés sont bien présents.
- Créez une tache supplémentaire et testez d'autres filtres que *all()* : *filter*, *exclude*, *get* , *order_by*, etc.
- Ces filtres correspondent à des requêtes SQL automatiquement générées par l'ORM de Django.

Par exemple *Task.objects.all()* génère :

```
select * from Task ;
```

```
Task.objects.filter(created_date__lte='2022-01-01')[0]
```

génère la requête :

```
select * from Task WHERE created_date <= '2022-01-01';
```

4.12 Utilisation de l'interface d'administration

Indication : L'interface administration de django est une application au même titre que celle que vous êtes en train de créer

Créons d'abord un compte d'administrateur avec un mot de passe qui nous permettra de rentrer dans le mode administrateur. Pour cela utilisons la commande :

```
python manage.py createsuperuser
```

- Il vous sera demandé un nom de login une adresse electronique et un mot de passe qu'il faudra confirmer.
- vérifiez que *django.contrib.admin* est présent dans la partie *INSTALLED_APPS* de *settings.py* et vérifier que la ligne

```
from django.contrib import admin
```

est présente dans *GestionTaches/urls.py* et que *urlpatterns* contient bien :

```
urlpatterns = [
path('admin', adim.site.urls)
```

(suite sur la page suivante)

(suite de la page précédente)

```
../..
]
```

Vous pouvez maintenant avec l'URL localhost:8000/admin (http ://localhost :8000/admin) accéder à l'administration de la base de données.

Avertissement : Prennez garde, que la suppression de la base de données entrainera la suppression du compte admin créé.

4.12.1 Ajout du modèle de tâches à l'admin

Pour ajouter le modele Task à l'administration il faut modifier le fichier *admin.py* de l'application de la manière suivante :

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

Complétez ensuite le modèle de la classe Task si nécéssaire. Que faut-il faire pour que django prenne en compte nos modifications ? (migrations, ...)

L'ennui est que les taches ne présentent que leur nom dans l'admin. On souhaiterait voir figurer également la *due_date* et la *schedule_date*. Pour ce faire, on modifie le fichier *admin.py* en lui ajoutant une classe héritant de *admin.modelAdmin*

```
from django.contrib import admin
from lesTaches.models import Task

class TaskAdmin(admin.ModelAdmin):
    class TaskAdmin(admin.ModelAdmin):
        list_display=('name','description','closed')
        read_only=('created_date')

admin.site.register(Task,TaskAdmin)
```

4.12.2 Ajout des champs due_date et schedule_date au modèle

On édite *models.py* pour celà avec éventuellement en paramètre l'indication *null=True* pour avoir en toutes circonstances des valeurs par défaut pour certains champs.

et on refait :

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

Testez !

Le modèle doit maintenant être :

```
from django.db import models

from datetime import datetime, timedelta, date

from django.utils.html import format_html

class Task(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()
    created_date = models.DateField(auto_now_add=True)
    closed = models.BooleanField(default=False)
    due_date = models.DateField(null=True)
    schedule_date = models.DateField(default=datetime.
↪now()+timedelta(days=7))
```

4.12.3 Personnalisation de l'affichage dans l'admin

Modifier l'affichage du champ *due_date* en fonction de la date du jour :

- vert si la tache est à réaliser dans plus d'une semaine
- orange si la tache est à réaliser dans la semaine e
- rouge sinon

Pour cela, on ajoute la méthode suivante dans la class Task :

```
def colored_due_date(self):
        due_date = django_date(self.due_date,"d F Y")
        if self.due_date is None or self.due_date-timedelta(days=7) >␣
↪date.today():
                color = "green"
        elif self.due_date < date.today():
                color = "red"
        else:
                color = "orange"
return format_html("<span style=color:%s>%s</span>" % (color, due_
↪date))


colored_due_date.allow_tags = True
```

Ajoutez ensuite l'affichage de *colored_due_date* dans l'interface admin et vérifiez la coloration syntaxique des dates selon le délai restant (ou dépassé !) : Pour cela, dans admin.py :

```
class TaskAdmin(admin.ModelAdmin):
        list_display = ('name', 'description', 'created_date',
↪'colored_due_date')
```

(suite sur la page suivante)

(suite de la page précédente)

```
admin.site.register(Task, TaskAdmin)
```

4.13 Intérêt des templates

Pour nous aider à afficher les données de façon plus structurée, Django fournit un système de templating intégré au framework.

Les développeurs peuvent s'il le souhaitent utiliser d'autres systèmes de templates, comme Jinja2 un peu plus répandu, mais le système de templates de Django est assez complet et possède une syntaxe volontairement épurée.

4.14 Les bases de l'interface client

Pour commencer, présentons une simple liste des taches en utilisant un template défini dans une chaine de caractères :

dans le fichier *lesTaches/views.py*, ajoutons :

```
from lesTaches.models import Task # import de la class Task
from django.shortcuts import render # import de la methode render

def task_listing(request):
        from django.template import Template,Context
        objets=Task.objects.all().order_by('due_date')
        template=Template('{% for elem in objets %} {{elem}} <br />{
↪%endfor%}')
        print(str(template))
        context=Context({'objets':objets})
        print(str(template.render(context)))
        return HttpResponse(template.render(context))
```

Pour que vous puissiez visualiser votre liste il faut modifier le fichier *lesTaches/urls.py* en y ajoutant une route :

```
path('listing', views.task_listing,name="listing"),
```

Testez sur l'URL localhost:8000/lesTaches/listing (http ://localhost :8000/lesTaches/listing) si votre serveur est toujours actif.

Cette solution, si elle est operationnelle, est un bien limitée et nous allons donc l'améliorer en utilisant des templates dans des fichiers séparés.

Commençons par créer un répertoire *templates* dans notre application lesTaches :

```
cd lesTaches
mkdir  templates
```

Par défaut, Django recherche les templates dans les dossiers *templates* de chaque app de votre projet.

Maintenant, il ne nous reste plus qu'à écrire le fichier template *list.html* dans le répertoire *lesTaches/templates/*

Pour utiliser le fichier *list.html*, on modifiera la vue en utilisant la fonction *render()* comme suit :

```
from lesTaches.models import Task
from django.shortcuts import render

def task_listing(request):
        tasks = Task.objects.all().order_by('due_date')
        return render(request,template_name='list.html',context={'tasks
↪':tasks})
```

avec list.html :

```
{% block content %}
<h1>Liste des tâches</h1>
<ul>
{% for task in tasks %}
        <li>{{task}}</li>
{% endfor %}
</ul>
{% endblock %}
```

4.15 Héritage de templates

Pour uniformiser nos pages nous allons créer une page de base dont hériterons les différentes pages de notre app. Voici un exemple nommé *baseLayout.html*, que nous pouvons déposer dans le dossier *templates* de notre app, tandis que nos ressources statiques (css, js, images, etc.) sont à placer dans le dossier *static* de notre app.

On utilise dans notre template la commande *load static*, et on préfixe les adresses des ressources statiques du mot-clé *static* :

```
<!DOCTYPE html>
{% load static %}
<html lang="fr" >
<head>
        <meta charset="utf-8" >
        <title>
                {% block title %}{% endblock %}
```

(suite sur la page suivante)

(suite de la page précédente)

```
        </title>
        <link rel="stylesheet" href="{% static css/mon.css %}" >
</head>
<body>
        <header>
        <h2> Gestion des Taches </h2>
        </header>

        <section>
                {% block content %}
                {% endblock %}
        </section>

        <footer>
                <em>Copyright IUT O, 2049</em>
        </footer>
</body>
</html>
```

Django utilise un système de templates volontairement doté d'une syntaxe simplifiée (par rapport à d'autres systèmes de templates un peu plus sophistiqués commes Jinja2 en Python ou Twig en PHP). La page est structurée de façon classique et nous avons utilisé la syntaxe des templates django pour créer deux blocs vides (titre et content) Ces deux blocs seront mis en place et remplis par les templates héritant de modeleBase.html comme par exemple list.html

Rappelons que dans le fichier views.py nous avons défini la fonction :

```
def task_listing2(request):
        objets = Task.objects.all().order_by('-due_date')
        # - (inverse l'ordre )
        return render(request,template_name='list.html', context={
↪'taches':objets})
```

Nous voyons que l'on passe à la fonction render() un dictionnaire et un template (list.html) Ce sont les données de ce dictionnaire que l'on va pouvoir utiliser dans ce template list.html. Nous commençont par déclarer que list.html hérite de notre template baselayout.html :

```
{% extends 'baseLayout.html' %}
{% block title %} Liste {% endblock %}
{% block content %}
<h3>Tâches en cours ou terminées</h3>
<ul>
{% for  tache in taches %}
        <li>
        <h3>
        {% if tache.closed %}
                <span class="closed">{{tache.name}}</span>
```

(suite sur la page suivante)

(suite de la page précédente)

```
        {% else %}
                {{tache.name}}
        {% endif %}
        </h3>
        <h4>{{tache.description }} </h4>
        <p>
        Date de rendu: {{tache.colored_due_date}}
        </p>
        <p>
        Date de debut: {{tache.schedule_date}}
        </p>
        </li>
{% endfor %}
</ul>
{% endblock %}
</html>
```

Vous pouvez constater que :

— l'on a fait hériter notre template de *baseLayout.html*
— nous avons donné un contenu au bloc titre
— nous avons complété le bloc content.
— les objets définis dans la fonction *task_listing2* se retouvent ici dans le mots clé taches qui était justement l'index des objets dans le dictionnaire.

On parcourt l'ensemble des taches par une boucle for et pour chaque tache on effectue le traitement souhaité.

4.15.1 Template générique

Les templates sont cherchés par défaut dans les dossiers templates de chacune des app qui composent notre projet mais peuvent aussi être placées dans le dossier templates situé à la racine de notre projet.

Ce réglage permet de définir un template de base pour l'ensemble de notre projet dont peuvent hériter les templates des différentes apps. Pour cela nous devons autoriser l'usage d'un dossier *templates* à la racine du projet. Modifiez pour cela la clef DIRS dans la rubrique TEMPLATES du fichier settings.py :

```
TEMPLATES = [
{
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        # 'DIRS': [],
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
        'context_processors': [
                'django.template.context_processors.debug',
```

(suite sur la page suivante)

(suite de la page précédente)

```
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
        ],
        },
},
]
```

4.15.2 Ressources statiques

De même, les ressources statiques sont cherchées par défaut dans les dossiers *static* des apps. On peut vouloir ajouter des dossiers de ressources statiques globales pour le projet. On peut placer ces ressources dans un dossier *static* à la racine du projet en ajoutant de la même façon dans *settings.py* :

```
STATICFILES_DIRS = [
        os.path.join(BASE_DIR, "static"),
]
```

Nous pourrons ensuite déposer des ressources statiques communes (JS, CSS, Images, favicons, etc.) dans le dossier *static* placé à la racine du projet.

4.16 Twitter Bootstrap

La librairie Bootstrap est la plus célèbre des libs CSS, mise à disposition par Twitter, elle permet de réaliser des sites élégants et responsives assez facilement.

La documentation se trouve là :

Bootstrap Doc (https ://getbootstrap.com/docs/5.1/getting-started/introduction/)

et les exemples :

Bootstrap examples (https ://getbootstrap.com/docs/5.1/examples/)

Il sera souvent le plus rapide de partir d'un ou plusieurs de ces exemples et de les modifier à notre guise. Nous allons ainsi construire un template de base pour notre application Django.

4.17 Template de base Bootstrap

Nous allons mettre en place un template *base.html* global au projet dans Gestion-Taches/templates utilisant la librairie CSS Bootstrap.

4.17.1 installation de dépendances statiques

Installons tout d'abord dans le dossier *static* du projet quelques librairies avec la commande *yarn* ou *npm* :

```
yarn add bootstrap jquery popper.js
```

Ce qui créera dans *static* un dossier *node_modules* contenant les librairies demandées.

4.17.2 template générique Bootstrap

Mettons maintenant en place un template assez générique, *base.html* dans le dossier *templates* du projet. Nous pouvons nous inspirer par exemple du starter template de Bootstrap (https ://getbootstrap.com/docs/5.1/examples/starter-template/)

```
<!doctype html>

{% load static %}

<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>
        {% block title %}{% endblock %}
    </title>
    {% block metas %}
    <meta name="viewport" content="width=device-width, initial-scale=1,
↪ shrink-to-fit=no">
    {% endblock %}
    {% block styles %}
    <link rel="stylesheet" href="{% static 'node_modules/bootstrap/
↪dist/css/bootstrap.min.css' %}">
    {% endblock %}
</head>

<body>
    {% block nav %}
        <nav class="navbar navbar-expand-md navbar-dark bg-dark mb-4">
            <h1> Gestion des Tâches</h1>
        </nav>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {% endblock %}
    <main role="main" class="container top-pushed">
        <div class="jumbotron">
            {% block content %}
            {% endblock %}
        </div>
    </main>
    {% block js %}
        <script src="{% static 'node_modules/jquery/dist/jquery.min.js
↪' %}"></script>
        <script src="{% static 'node_modules/bootstrap/dist/js/
↪bootstrap.min.js' %}"></script>
        <script src="{% static 'node_modules/popper.js/dist/popper.min.
↪js' %}"></script>
    {% endblock %}
</body>

</html>
```

ou si on installe « à la main » les librairies dans static :

```
    <!doctype html>

{% load static %}

<html lang="fr">

<head>
    <meta charset="utf-8">
    <title>
        {% block title %}{% endblock %}
    </title>
    {% block metas %}
    <meta name="viewport" content="width=device-width, initial-scale=1,
↪ shrink-to-fit=no">
    {% endblock %}
    {% block styles %}
        <link rel="stylesheet" href="{% static 'bootstrap/css/
↪bootstrap.min.css' %}">
    {% endblock %}
</head>

<body>
    {% block nav %}
    <nav class="navbar navbar-expand-md navbar-dark bg-dark mb-4">
        <h1> Gestion des Tâches</h1>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    </nav>
    {% endblock %}
    <main role="main" class="container top-pushed">
        <div class="jumbotron">
            {% block content %}
            {% endblock %}
        </div>
    </main>
    {% block js %}
        <script src="{% static 'js/jquery-3.3.1.min.js' %}"></script>
        <script src="{% static 'bootstrap/js/bootstrap.min.js' %}"></
↪script>
    {% endblock %}
</body>
</html>
```

Ce template va nous servir de base pour tous les templates de notre site. Nous y avons incorporé des zones variables comme *styles*, *nav*, *js* ou *content* qui pourront être redéfinies dans les templates héritiers. Nous pouvons placer les templates héritiers plus spécialisés dans les dossiers templates des différentes apps. Pour cela il faut ajouter le réglage suivant dans *settings.py*, à la suite de :

```
STATIC_URL = '/static/'
```

ajoutez :

```
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),
]
```

Maintenant, il ne nous reste plus qu'à écrire le fichier template *list2.html* dans le répertoire *lesTaches/templates/*

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

{% load static %}

{% block title %}
    Les Tâches
{% endblock %}

{% block styles %}
    {{ block.super }}
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/
↪v5.3.1/css/all.css"
      integrity="sha384-
↪mzrmE5qonljUremFsqc01SB46JvROS7bZs3IO2EmfFsd15uHvIt+Y8vEf7N7fWAU"
```

(suite sur la page suivante)

(suite de la page précédente)

```
      crossorigin="anonymous">
{% endblock %}

{% block content %}
<h1>Liste des tâches</h1>

{% for tache in objects %}
<div class="row">
    <div class="col">{{tache.name}}</div>
    <div class="col">{{tache.created_date}}</div>
    <div class="col">{{tache.colored_due_date}}</div>
    <div class="col">
            {% if tache.closed %}
                <i class="fas fa-lock"></i>
            {% else %}
                <i class="fas fa-lock-open"></i>
            {% endif %}
    </div>

</div>
{% endfor %}

{% endblock %}
```

et modifier dans views.py la méthode task_listing, appelant ce template en utilisant la fonction *render()* de django :

```
def task_listing(request):
    objects = Task.objects.all().order_by('due_date')
    return render(request, template_name='list2.html', context={
↪'objects': objects} )
```

La création de formulaires avec Django est assez simple et pleine de fonctionnalités mais la construction de formulaires parfaitement adaptés peut s'avérer quelque peu délicate !

Vous pouvez générer des formulaires à partir de vos modèles ou bien les créer directement depuis des classes. On peut également utiliser des formulaires générés automatiquement mais avec un style plus élaboré comme *Crispy Forms*.

4.18 Création de formulaires en Django

4.18.1 Création d'une nouvelle app dans notre projet

Passons tout de suite à la pratique, et mettons cela en place. Pour plus de propreté, nous allons créer une nouvelle app dans notre projet, que nous appellerons "myform".

Reprenons le projet GestionTaches après avoir initialisé votre environnement virtuel et créé l'application myform

```
./manage.py startapp myform
```

Comme nous l'avons déjà fait avec lesTaches, ajoutons cette application au fichier *settings.py* :

```
INSTALLED_APPS = (
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.sites',
'django.contrib.messages',
'django.contrib.staticfiles',
'django.contrib.admin',
'django.contrib.admindocs',
'lesTaches',
'myform',
)
```

4.18.2 Modèles de l'app myform

Créons à présent le modèle de notre application, on ouvre pour cela le fichier *models.py* contenu dans le dossier de notre app "myform" et on l'édite comme ceci :

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


class Contact(models.Model):
    name = models.CharField(max_length=200)
    firstname = models.CharField(max_length=200)
    email = models.EmailField(max_length=200)
    message = models.CharField(max_length=1000)
```

On crée donc une classe "Contact" qui contient un nom, un prénom, un email et un message, puis :

```
# Générons la en base de données:
./manage.py makemigrations myform
# vérifions le contenu de la table avec
./manage.py sqlmigrate myform 0001
# et réalisons la migration
./manage.py migrate
```

Et voilà, notre modèle est présent en base de données, passons à la création de notre contrôleur, afin de générer notre formulaire. On ouvre pour cela le fichier *views.py* présent dans notre application, et on l'édite :

```
from django.shortcuts import render
from django.forms import ModelForm
from myform.models import Contact

class ContactForm(ModelForm):
class Meta:
    model = Contact
    fields = ('name', 'firstname', 'email', 'message')

def contact(request):

contact_form = ContactForm()
return render(request,'contact.html', {'contact_form' : contact_form})
```

On importe notre modèle, un "helper", la méthode *render()*, ainsi que la classe *ModelForm*. On définit alors notre classe qui va permettre de générer notre formulaire en la faisant dériver de ModelForm et en lui spécifiant le modèle à inclure *Contact*. Il ne nous reste plus qu'à déclarer une nouvelle variable qui sera une instance de la nouvelle classe créée, et à la passer en tant que donnée à notre template.

4.18.3 Routes pour l'app myform

Maintenant il va falloir créer les urls correspondantes :

Tout d'abord dans GestionTaches ajoutons dans *urls.py* la ligne suivante :

```
path('contacts/',include('myform.urls')),
```

et créons le fichier *urls.py* dans le répertoire *myform* :

```
from django.urls import path
from . import  views
urlpatterns=[
    path('',views.contact,name='contact'),
]
```

Créons maintenant notre template qui se nommera *contact.html* dans *myform/templates/* :

```
{{ contact_form.as_p }}
```

Voilà ! Pas plus !

Nous affichons ici directement notre objet *contact_form* et on lui associe l'option d'affichage *as_p* qui signifie que le formulaire sera affiché en utilisant la balise <*p*>.

4.18.4 Test du premier formulaire

Nous pouvons à présent apprécier le résultat ! Lancez votre serveur :

```
./manage.py runserver
```

Et dirigez-vous à l'adresse : http://localhost:8000/contacts/ Vous devriez obtenir un formulaire (moche) avec les libellés de la classe Contact.

Notre formulaire html, utilisant des balises "<p>", a été automatiquement généré ! Pour l'instant il reste encore assez rudimentaire, mais nous allons voir que nous allons pouvoir très vite tout personnaliser et avoir un bien meilleur rendu ! D'autres options d'affichage telles que *as_ul* et *as_table* auraient pu être utilisées.

Pour plus d'informations sur les forms, voir la doc officielle : https://docs.djangoproject.com/fr/3.1/topics/forms/

4.18.5 Création d'un formulaire à la main (sans Model)

Mais avant cela, nous allons nous intéresser à la création de formulaires sans utiliser de modèle, directement avec le module forms de Django ! Pour ceux qui souhaitent approfondir la création de formulaires à partir de modèles, voir la documentation officielle. Créons maintenant notre formulaire *ContactForm2* sans utiliser de *ModelForm*, en modifiant comme suit le *views.py* de l'application "myform" :

```
from django.shortcuts import render
from django.forms import ModelForm
from models import Contact

# Avec ModelForm
class ContactForm(ModelForm):
    class Meta:
        model = Contact
        fields = ('name', 'firstname', 'email', 'message')

from django import forms

# Sans ModelForm
class ContactForm2(forms.Form):
    name = forms.CharField(max_length=200)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    firstname = forms.CharField(max_length=200)
    email = forms.EmailField(max_length=200)
    message = forms.CharField(max_length=1000)

def contact(request):
    contact_form = ContactForm()
    contact_form2 = ContactForm2()
    return render(request,'contact.html', {'myform/contact_form' :␣
↪contact_form, 'contact_form2' : contact_form2})
```

On inclut directement le module "forms" de django et on crée une classe *ContactForm2* dérivant de Form et non de ModelForm cette fois-ci. On crée ainsi notre classe, ce qui ressemble sensiblement à la déclaration de notre modèle. Puis on crée un objet *contact_form2* dans notre fonction et on le passe en paramètre à notre template. Maintenant éditez votre fichier *contact.html* de cette façon :

```
{{contact_form.as_p  }}
<br/>
{{ contact_form2.as_p }}
```

On obtient donc deux formulaires identiques mais créés de deux façons différentes !

4.18.6 Personnalisation des formulaires et vérifications

Intéressons-nous maintenant à la personnalisation de ces formulaires avant de comprendre comment les utiliser et gérer leur validation. Les formulaires de type *ModelForm* ainsi que ceux de type *Form* peuvent contenir de nombreuses options de personnalisation.

On peut tout de suite remarquer certaines options telles que

[*required*, *label*, *initial*, *widget*, *help_text*, *error_messages*, *validator*, *localize* ou] *required*. Ce dernier va spécifier si le champ du formulaire peut être ignoré ou non, par défaut tous les champs sont requis.

Cela implique que si l'utilisateur valide le formulaire avec des champs à vide (» » ou None), une erreur sera levée. Pour illustrer cela, voici ce que donnent les tests suivants après avoir ouvert une console python en mode django et avoir importé le module forms :

Ouvrons un shell et testons :

```
>>>from django inport forms
>>> f=forms.CharField()
>>> f.clean('foo')
'foo'
>>> f.clean('')
''
ValidationError: ['This field is required.'] ...
>>> f = forms.CharField(required=False)
>>> f.clean(None)
```

(suite sur la page suivante)

(suite de la page précédente)

```
''
ValidationError: ['This field is required.'] ...
```

Nous simulons ici la vérification de formulaire avec la méthode *clean()*. Comme vous pouvez le constater, une erreur est bien levée si nous passons une chaîne vide ou *None* à notre champ. *label* va permettre de réécrire la façon dont le label du champ est affiché, par défaut, cela prend le nom de la variable et lui ajoute une majuscule à la première lettre et remplace tous les underscores par des espaces.

Exemple :

```
>>>class CommentForm(forms.Form):
...     name = forms.CharField(label='Your name')
...     url = forms.URLField(label='Your Web site', required=False)
...     comment = forms.CharField()
...
>>> f = CommentForm()
>>> print(f)
<tr><th>
<label for="id_name">Your name:</label>
</th>
<td><input type="text" name="name" id="id\_name" />
</td></tr>
<tr><th>
<label for="id_url">Your Web site:</label>
</th><td>
<input type="text" name="url" id="id_url" /></td></tr>
<tr><th><label for="id_comment">Comment:</label>
</th><td>
<input type="text" name="comment" id=">>>
f = forms.CharField() /></td></tr>
```

Vous voyez également qu'ici on affiche notre *form* sans utiliser d'option d'affichage, il est donc affiché par défaut en utilisant une table. Pour plus de commodité, on aurait pu rajouter l'option *auto_id* à *False*, ce qui aurait pour effet de désactiver la création d'une balise label, et génèrerait ceci :

```
>>>f = CommentForm(auto_id=False)
>>> print(f)
<tr><th>Your name:</th>
<td><input type="text" name="name" /></td></tr>
<tr><th>Your Web site:</th><td>
<input type="text" name="url" />
</td></tr>
<tr><th>Comment:</th><td>
<input type="text" name="comment" />
</td></tr>
```

initial va permettre de donner des valeurs par défaut à vos champs, exemple :

```
>>>class CommentForm(forms.Form):
...     name = forms.CharField(initial='Your name')
...     url = forms.URLField(initial='http://')
...     comment = forms.CharField()
>>> f = CommentForm(auto_id=False)
>>> print f
<tr><th>Name:</th><td>
<input type="text" name="name" value="Your name" />
</td></tr>
<tr><th>Url:</th><td>
<input type="text" name="url" value="http://" /></td></tr>
<tr><th>Comment:</th><td>
<input type="text" name="comment" />
</td></tr>
```

Nous nous arrêterons à ces trois options pour le moment et verrons les autres dans des cas plus poussés. Par contre il est très intéressant de se pencher sur l'option *widget* car elle va vous permettre de réaliser très facilement des champs spécifiques :

```
django import forms
class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

Ou bien :

```
from django.forms.fields import DateField, ChoiceField,␣
↪MultipleChoiceField
from django.forms.widgets import RadioSelect, CheckboxSelectMultiple
from django.forms.extras.widgets import SelectDateWidget

BIRTH_YEAR_CHOICES = ('1999', '2000', '2001')
GENDER_CHOICES = (('m', 'Male'), ('f', 'Female'))
FAVORITE_COLORS_CHOICES = (('blue', 'Blue'),
                           ('green', 'Green'),
                           ('black', 'Black'))

class SimpleForm(forms.Form):
    birth_year = DateField(widget=SelectDateWidget(years=BIRTH_YEAR_
↪CHOICES))
    gender = ChoiceField(widget=RadioSelect, choices=GENDER_CHOICES)
f   avorite_colors = forms.MultipleChoiceField(required=False,
    widget=CheckboxSelectMultiple, choices=FAVORITE_COLORS_CHOICES)
```

Un petit dernier …

```
        class CommentForm(forms.Form):
            name = forms.CharField(
                        widget=forms.TextInput(attrs={'class':'special'}
↪))
            url = forms.URLField()
            comment = forms.CharField(
                        widget=forms.TextInput(attrs={'size':'40'}))


Ce qui donnera :
```

```
f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td>
<input type="text" name="name" class="special"/></td></tr>
<tr><th>Url:</th><td>
<input type="text" name="url"/></td></tr>
<tr><th>Comment:</th><td>
<input type="text" name="comment" size="40"/></td></tr>
```

N'hésitez pas à parcourir les différents types de widgets dans la doc de django et à les essayer pour découvrir ce qu'ils produisent.

4.19 ModelForms et templates

4.19.1 ModelForms

A noter que vous pouvez également utiliser les widgets avec des Form venant de Modèles :

```
from django.forms import ModelForm, Textarea

class AuthorForm(ModelForm):
class Meta:
    model = Author
    fields = ('name', 'title', 'birth_date')
    widgets = {
        'name': Textarea(attrs={'cols': 80, 'rows': 20}),
    }
```

Modifier les formulaires de l'application myform en utilisant les informations ci-dessus. Tester par exemple en modifiant la vue de myform comme ci-dessous :

```
from django.shortcuts import render
from django.forms import ModelForm, Textarea
from myform.models import Contact
```

(suite sur la page suivante)

(suite de la page précédente)

```
from django import forms

class ContactForm(ModelForm):

class Meta:
    model = Contact
    fields =('name,'firstaname,'email','message')
        widgets = {

        'message': Textarea(attrs={'cols':60,'rows':10}),

    }

class ContactForm2(forms.Form):
    name = forms.CharField(max_length=200,initial="votre nom",label=
↪"nom")
    firstname = forms.CharField(max_length=200,initial="votre prenom",
↪label="prenom")
    email = forms.EmailField(max_length=200,label="mel")
    message = forms.CharField(widget=forms.Textarea(attrs={'cols':60,
↪'rows':10}))


def contact(request):

    contact_form = ContactForm()
    #return render(request,'myform/conctact.html',{'contact_form
↪':contact_form})

    contact_form2 = ContactForm2()
    return render(request,'myform/conctact.html',{'contact_form
↪':contact_form,'contact_form2':contact_form2})
```

Essayez d'autres possibilités. Enfin, nous allons nous intéresser à la validation des formulaires, comment gérer des messages pour la page suivante et comment travailler avec ceux-ci. Nous allons reprendre le fichier *views.py* de notre application "myform" et remplacer son contenu par celui-ci :

```
from django.shortcuts import render, redirect
from django.forms import ModelForm, Textarea
from myform.models import Contact
from django import forms
from django.urls import reverse
from django.http import HttpResponse
from django.contrib import messages
```

(suite sur la page suivante)

(suite de la page précédente)

```
class ContactForm(ModelForm):
    def __init__(self, *args, **kwargs):
        super(ContactForm, self).__init__(*args, **kwargs)
        self.fields['name'].label = "Nom "
        self.fields['firstname'].label = "Prenom"
        self.fields['email'].label = "mél"
    class Meta:
        model = Contact
        fields = ('name', 'firstname', 'email','message')
        widgets = {'message': Textarea(attrs={'cols': 60, 'rows': 10}),
↪}

    def contact(request):
    # on instancie un formulaire
    form = ContactForm()
    # on teste si on est bien en validation de formulaire (POST)
    if request.method == "POST":
        # Si oui on récupère les données postées
        form = ContactForm(request.POST)
        # on vérifie la validité du formulaire
        if form.is_valid():
            new_contact = form.save()
            # on prépare un nouveau message
            messages.success(request,'Nouveau contact '+new_contact.
↪name+' '+new_contact.email)
            #return redirect(reverse('detail', args=[new_contact.pk] ))
            context = {'pers': new_contact}
            return render(request,'detail.html', context)
    # Si méthode GET, on présente le formulaire
    context = {'form': form}
    return render(request,'contact.html', context)
```

Avec une nouvelle vue *detail* :

```
def detail(request, cid):
    contact = Contact.objects.get(pk=cid)
    context = {'pers': contact}
    return render(request,'detail.html', context)
```

et un template detail.html :

```
{% extends "base.html" %}
    {% load static %}
    {% block title %}
    Contacts
    {% endblock %}
    {% block header %}
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {% if messages %}
        {% for message in messages %}
            <h3>{{message}}</h3>
        {% endfor %}
    {% endif %}
    {% endblock %}

    {% block content %}
    <ul>
        <li>{{pers.firstname}}</li>
        <li>{{pers.email}}</li>
        <li>{{pers.message}}</li>
    </ul>
    {% endblock %}
```

Et nous allons définir une nouvelle url pour la page « detail ».

Pour ce faire on modifiera le fichier *urls.py* de *myform*. Pour tester, il nous faudra également modifier le template contact.html :

```
{% extends "base.html" %}
    {% load static %}
    {% block title %}
    Nouveau Contact
    {% endblock %}

    {% block navtitle %}
        Nouveau Contact
    {% endblock %}

    {% block content%}
        <form action="/contacts/" method="POST">
        {% csrf_token %}
        {{ form }}
        <button type="submit">
            Sauvegarder
        </button>
        </form>
    {% endblock %}
```

4.20 CrispyForms

Pour bénéficier de formulaires plus jolis, nous allons installer et mettre en marche le module *crispy-forms*. Pour cela ajoutez le à votre venv :

```
pip install django-crispy-forms
```

puis ajoutez crispy-forms aux apps installées dans settings.py :

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

Puis lisez la documentation des crispy-forms pour adapter vos formulaires et les rendre plus attrayants ! Pour contact.html par exemple :

```
{% extends "base.html" %}
    {% load static %}
    {% load crispy_forms_tags %}
    {% block title %}
    Nouveau Contact
    {% endblock %}

    {% block navtitle %}
        Nouveau Contact
    {% endblock %}

    {% block content%}
        <form action="/contacts/" method="POST">
        {% csrf_token %}
        {{ form | crispy }}
        <button type="submit" class="btn btn-success">
            Sauvegarder
        </button>
        </form>
    {% endblock %}
```

On peut aussi personnaliser davantage les différents widgets des formulaires avec des attributs spécialisés en utilisant la librairie widget-tweaks (https ://pypi.org/project/django-widget-tweaks/)

4.21 App déployée sur Gitpod

Le code correspondant se trouve sur Github (https ://github.com/roza/django-starter-bsootstrap)

Le projet est disponible, prêt à être utilisé et complété sur gitpod :

(https ://gitpod.io/#https ://github.com/roza/django-starter-bootstrap)

4.22 Compléments sur les modèles Django

Nous allons à présent explorer quelques modèles de relations classiques de Django. Vous trouverez leur documentation (https ://docs.djangoproject.com/fr/3.2/topics/db/models/). Les plus utilisés sont de type *ManyToOne* avec des *ForeignKey* ou les relations *ManyToMany*.

4.22.1 Les Foreign Keys

Si on a par exemple de `Task` et de `Users` et une ou plusieurs tâches assignées à un utilisateur, on crée une ForeignKey *utilisateur* vers le modèle de `User`.

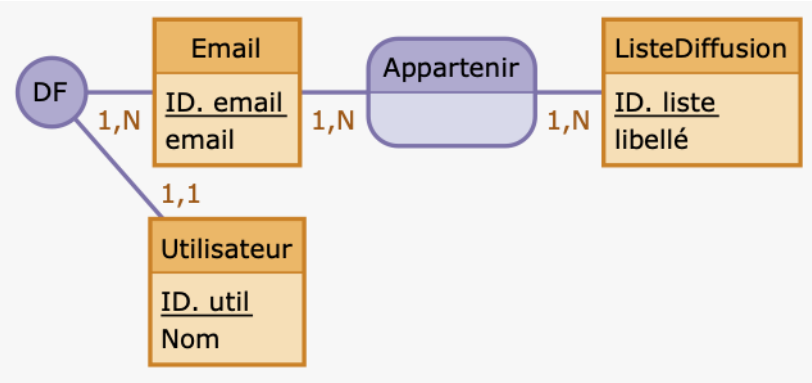
```
class User(models.Model):
    username = models.CharField(max_length=150)

    def __str__(self):
        return self.username


class Task(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()
    creation_date = models.DateField(auto_now_add=True)
    # ../..
    closed = models.BooleanField(default=False)
    utilisateur = models.ForeignKey(User, on_delete=models.CASCADE,␣
↪related_name="tasks")

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

Indication : text

Notez l'attribut `related_name=tasks` qui permettra de retrouver aisément toutes les tâches assignées à un utilisateur *user* avec l'appel : `user.tasks.all()` :

4.22.2 ManyToMany

Si on considère un modèle avec des Utilisateurs, des Emails et des Listes de Diffusion, on peut considérer qu'un utilisateur peut avoir plusieurs adresses mail, qu'une liste de diffusion contient différents emails et qu'un email peut apparaitre dans plusieurs listes de diffusion :



On peut alors déclarer ces entités :

```
"""models.py"""

# utilisateurs
class Utilisateur(models.Model):
    nom = models.CharField(max_length=30)

    def __str__(self):
        return "%s %s" % (self.prenom, self.nom)

# emails
class Email(models.Model):
    mail = models.CharField(max_length=30)
    user = models.ForeignKey(Utilisateur, on_delete=models.CASCADE,
↪null=True, related_name='emails')

    def __str__(self):
        return self.mail

# listes de diffusion
class ListeDiffusion(models.Model):
    listeName = models.CharField(max_length=30)
    email = models.ManyToManyField(Email, related_name='listes')

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

puis les manipuler :

```
from .models import Utilisateur, Email, ListeDiffusion
```

(suite sur la page suivante)

(suite de la page précédente)

```
# On crée 2 utilisateurs
user1 = Utilisateur(nom='user1')
user2 = Utilisateur(nom='user2')

# On crée les mails associés
email1 = Email(mail='user1@baba.fr', user=user1)
email2 = Email(mail='user2@bibi.fr', user=user2)

# On crée 4 listes de diffusion
liste1 = ListeDiffusion(listeName='liste1')
liste2 = ListeDiffusion(listeName='liste2')
liste3 = ListeDiffusion(listeName='liste3')
liste4 = ListeDiffusion(listeName='liste4')

# on sauvegarde tout le monde
user1.save()
user2.save()
email1.save()
email2.save()
liste1.save()
liste2.save()
liste3.save()
liste4.save()

# on ajoute les emails à des listes
liste1.email.add(email1)
liste1.email.add(email2)
liste2.email.add(email1)
liste3.email.add(email1)
liste3.email.add(email2)
liste4.add(email2)

# Pour obtenir toutes les listes auxquelles l'email1 est abonné :
email1.listes.all()
# Et celles auxquelles l'email2 est abonné :
email2.listes.all()
```

On peut ensuite faire une petite vue du style :

```
#views.py
def email_detail(request, pk):
    email = Email.objects.get(pk=pk)
    user = email.user
    listes_abonnees = email.listes.all()
```

(suite sur la page suivante)

(suite de la page précédente)

```
    template = 'email_detail.html'
    context = {'user': user, 'email': email.mail, 'listes': listes_
↪abonnees }
    return render(request, template, context)
```

puis un template :

```
# template email_detail.html
<p>
Email : {{email}}
</p>
<p>
User : {{user}}
</p>
Listes de diffusion :
<ul>
{% for liste in listes %}
    <li>{{ liste }}</li>
{% endfor %}
</ul>
```

4.22.3 documentation

Les docs détaillées sont là :

- ManyToOne = ForeignKey (https ://docs.djangoproject.com/fr/3.2/topics/db/examples/many_to_one/)
- ManyToMany (https ://docs.djangoproject.com/fr/3.2/topics/db/examples/many_to_many/)
- OneToOne (https ://docs.djangoproject.com/fr/3.2/topics/db/examples/one_to_one/)

4.23 Vues génériques et compléments sur les migrations avec Django

Nous allons à présent explorer les vues génériques de Django qui sont aussi appelées *Vues fondées sur des classes*. Vous trouverez leur documentation sur le site de django (https ://docs.djangoproject.com/fr/3.2/topics/class-based-views/). L'usage des vues génériques permet d'utiliser des vues associées à des classes en lieu et place des fonctions. Cela permet aussi de mieux organiser le code et de davantage le réutiliser en faisant parfois usage de *mixins* (héritage multiple) Nous approfondirons au passage notre compréhension des migrations Django.

4.23.1 Initialisation du projet

Vous allez créer un petit projet de gestion de musiques pour illuster tout ça !

```
django-admin startproject GestionMusiques
cd GestionMusiques
./manage.py startapp musiques
```

N'oubliez pas d'ajouter l'application **musiques** nouvellement créée dans la liste des apps installées, ainsi que de modifier le fichier *urls.py* :

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

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

4.23.2 Création d'un modèle simple

On veut pouvoir stocker des entités modélisant des morceaux de musique. On va donc définir notre modèle dans le fichier *models.py* de l'app **musiques**. Le modèle `Morceau` ne requiert que des champs de type texte, rendez vous sur la documentation officielle pour de plus amples détails sur ce qu'il est possible de faire.

```
class Morceau(models.Model):
    titre = models.CharField(max_lenght=64)
    artiste = models.CharField(max_length=64)

    def __str__(self):
        return '{self.titre} ({self.artiste})'.format(self=self)
```

Mettez à jour le schéma de votre base de donnée avec le couple *makemigrations/migrate*. On reviendra sur ces deux commandes un peu plus tard.

```
./manage.py makemigrations
./manage.py migrate
```

Enfin, peuplez votre base de données avec quelques morceaux de bon goût (à partir de la console Django lancée avec `python manage.py shell`)

```
from musiques.models import Morceau

Morceau.objects.create(titre='Fruit de la passion', artiste='Francky␣
↪Vincent')
Morceau.objects.create(titre='Space Oddity', artiste='David Bowie')
```

4.23.3 Afficher un morceau

Ajout d'un test

Ecrivons maintenant des tests afin de nous assurer que certaines routes existent dans notre app pour afficher les objets de la base de données. Aucune vue Django n'a encore été créée, il est donc évident que les tests vont échouer : c'est tout le principe du *Test Driven Development* (TDD). Les tests pour une fonctionnalité doivent être écrits avant l'implémentation de la fonctionnalité.

Plus précisément, nous allons tester d'une part le fait qu'une url nommée existe pour afficher un objet, d'autre part que le traitement de la requête s'effectue sans erreur (code http 200).

```
from django.urls import reverse
from django.test import TestCase
from django.urls.exceptions import NoReverseMatch

from musiques.models import Morceau


class MorceauTestCase(TestCase):
    def setUp(self):
        Morceau.objects.create(titre='musique1', artiste='artise1')
        Morceau.objects.create(titre='musique2', artiste='artise2')
        Morceau.objects.create(titre='musique3', artiste='artise3')

    def test_morceau_url_name(self):
        try:
            url = reverse('musiques:morceau-detail', args=[1])
        except NoReverseMatch:
            assert False

    def test_morceau_url(self):
        morceau = Morceau.objects.get(titre='musique1')
        url = reverse('musiques:morceau-detail', args=[morceau.pk])
        response = self.client.get(url)
        assert response.status_code == 200
```

La commande `./manage.py test` devrait vous indiquer à ce stade que vos deux tests ne passent pas.

4.23.4 Création d'une url

La première étape consiste à définir une nouvelle url dans notre app :

Direction le fichier *musiques/urls.py* :

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

app_name = 'musiques' # Encapsule les urls de ce module dans le␣
↪namespace musique
urlpatterns = [
    path('<int:pk>', morceau_detail, name='morceau-detail')
]
```

La vue `morceau_detail` n'existe pas, vous allez donc en créer une qui ne fait rien !

```
def morceau_detail(request, pk):
    pass
```

La commande `./manage.py test` devrait à présent vous informer qu'un test ne passe pas (au lieu des 2).

Et vous pouvez compléter :

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

def morceau_detail(request,pk):
        return HttpResponse('OK')
```

… pour passer enfin les 2 tests ! !

4.23.5 Création de la vue

Vous allez implémenter la vue qui permettra d'afficher une instance du modèle `Morceau`. Pour cela, vous allez utiliser une *vue générique basée sur une classe*.

Django fournit différentes classes génériques pour représenter des vues, comme `DetailView` ou `ListView`.

C'est la classe `DetailView` qui va vous intéresser ici puisqu'elle permet d'associer une vue à un model. D'autre classes existent pour répondre à d'autres problématiques, vous en trouverez la liste sur la documentation officielle (https ://docs.djangoproject.com/fr/3.2/ref/class-based-views/).

```
from django.views.generic import DetailView

from .models import Morceau
```

(suite sur la page suivante)

(suite de la page précédente)

```
class MorceauDetailView(DetailView):
    model = Morceau
```

La vue Django est créée, à présent il faut déterminer comment l'afficher. C'est le template qui va s'en charger. Par défaut, `DetailView` va chercher un template nommé selon le nom du model renseigné. Ici ce template s'appelle *morceau_detail.html* et doit être situé dans le dossier *templates/musiques* de l'app **musiques**. Créez à présent ce fichier et remplissez le. L'objet attendu s'appelle *object* ou *morceau*

La dernière étape consiste à modifier le fichier *musiques/urls.py*. La méthode `morceau_detail` n'existe plus, il faut alors indiquer que c'est la classe `MorceauDetailView` qui va se charger de la requête.

```
from .views import MorceauDetailView

app_name = 'musiques'
urlpatterns = [
      path('<int:pk>', MorceauDetailView.as_view(), name='morceau-
↪detail')
]
```

Notez la différence, la fonction `path` attend une fonction en deuxième paramètre, `MorceauDetailView` étant une classe, on doit utiliser la méthode de classe `as_view` pour que `path` fonctionne correctement.

A présent, la commande `./manage.py test` devrait indiquer que tous les tests passent.

Complétons le template *morceau_detail.html* :

```
<ul>
<li>{{object.titre}}</li>
<li>{{object.artiste}}</li>
</ul>
<a href="{% url 'musiques:morceau_list' %}">
    Revenir a la liste
</a>
```

Notez l'usage de `{% url 'musiques:morceau_list' %}` pour faire référence à la route « morceau_list » que nous allons ajouter à présent à notre app *musiques* :

4.23.6 ListView

Ajoutez à vos views :

```
from django.views.generic import ListView

class MorceauList(ListView):
    model = Morceau
```

Puis à urls.py :

```
from django.urls import path

from .views import MorceauDetailView, MorceauList
app_name = 'musiques'

urlpatterns = [
    path('<int:pk>', MorceauDetailView.as_view(), name='morceau_detail
↪'),
    path('', MorceauList.as_view(), name='morceau_list'),
]
```

Avec le petit template correspondant morceau_list.html à placer dans musiques/templates/musiques :

```
<h2>Morceaux</h2>
<ul>
    {% for morceau in morceau_list %}
    <li>
        <h3>Morceau n°{{morceau.pk}}</h3>
        {{ morceau.titre }} par {{ morceau.artiste }}
    </li>
    {% endfor %}
</ul>
```

Testez sur http://localhost:8000/musiques !

4.23.7 Complexification du modèle

Le modèle `Morceau` est très pauvre, beaucoup d'informations manquent. Par exemple, on ne connait pas la date de sortie d'un morceau donné. Heureusement, grâce au système de *migrations*, le schéma de notre base de données n'est pas figé, on va pouvoir le modifier.

4.23.8 Ajout d'un champ nullable

On va dans un premier temps compléter notre modèle en lui ajoutant un champ *date_sortie*. Modifiez votre fichier *musiques/models.py* en ajoutant ce champ date :

```
date_sortie = models.DateField(null=True)
```

Il est nécessaire de spécifier `null=True` car lorsque le moteur de migration ajoutera le champ dans la table, il ne saura pas quelle valeur attribuer à la colonne nouvellement créée. Il mettra donc une valeur nulle par défaut. Il est également possible de spécifier `blank=True`. Qu'est-ce que cela signifie ? Recherchez dans la doc de Django.

Ensuite, la commande `./manage.py makemigrations musiques` va scanner le fichier *musiques/models.py* pour détecter les changements entre la version courante du modèle et la nouvelle (celle avec le champ date). A la suite de cette commande un fichier est créé dans le répertoire *musiques/migrations*. Celui-ci est un fichier Python indiquant la marche à suivre pour passer de la version 1 du schéma de base de données à la version 2. Pour appliquer ces changements, on utilise la commande `./manage.py migrate` qui va lire ce fichier et appliquer les modifications nécessaires. Utilisez `./manage.py sqlmigrate musiques 0002` pour avoir un aperçu du code SQL appliquant la modification.

4.23.9 Déplacement d'un champ

Vous avez certainement déjà entendu parler du problème de la redondance des données dans une base de données relationnelle. Ce problème apparait lorsqu'une même donnée figure plusieurs fois dans la base de donnée sous des formes différentes. Si on enregistre deux morceaux de musique avec le même artiste, l'information contenue dans la colonne *artiste* devient redondante : on aimerait pouvoir lier un morceau avec un artiste, qui ne serait créé qu'une seule fois.

On va donc créer une nouvelle classe représentant un artiste dans le fichier *musiques/models.py*

```
class Artiste(models.Model):
    nom = models.CharField(max_length=64)
```

On va ensuite générer un fichier de migration avec la commande *makemigrations* puis appliquer cette migration avec *migrate* :

```
./manage.py makemigrations
./manage.py migrate
```

Cela aura pour conséquence de générer une table Artiste vide.

Vous allez à présent peupler cette nouvelle table à l'aide d'une *migration de données*, c'est à dire une migration n'impactant pas le schéma de la base, mais uniquement ses données.

Créez une migration vide avec `./manage.py makemigrations --empty musiques`, ce qui aura pour effet de créer le fichier :

musiques/migrations/0004_auto_date_num.py

Ouvrez ce fichier et constatez qu'une liste vide est affectée à la variable `operations` : cela signifie que cette migration, en l'état, ne fait rien. Vous allez indiquez à Django comment il doit migrer les données de votre base, grâce à la méthode `migrations.RunPython`. Cette méthode doit recevoir une fonction définissant le comportement de la migration. Il est souvent sécurisant de spécifier le paramètre `reverse_code` qui permettra à Django d'annuler la migration en cas de problème et de revenir au schéma antérieur.

```
from django.db import migrations

def migrer_artiste(apps, schema):
    # On récupère les modèles
    Morceau = apps.get_model('musiques', 'Morceau')
    Artiste = apps.get_model('musiques', 'Artiste')

    # On récupère les artistes déjà enregsitrés dans la table Morceau
    # Voir la documentation :
    # - https://docs.djangoproject.com/fr/3.1/ref/models/querysets/#all
    # - https://docs.djangoproject.com/fr/3.1/ref/models/querysets/
↪#values
    # - https://docs.djangoproject.com/fr/3.1/ref/models/querysets/
↪#distinct
    artistes_connus = [fields['artiste']
                       for fields in Morceau.objects.all().values(
↪'artiste').distinct()]
    # On peuple la table Artiste
    for artiste in artistes_connus:
        Artiste.objects.create(nom=artiste)

def annuler_migrer_artiste(apps, schema):
    Artiste = apps.get_model('musiques', 'Artiste')
    Artiste.objects.all().delete()

class Migration(migrations.Migration):

    dependencies = [
        ('musiques', '0003_artiste'),
    ]

    operations = [
        migrations.RunPython(migrer_artiste,
                             reverse_code=annuler_migrer_artiste)
    ]
```

Prenez l'habitude de définir la fonction de retour en arrière (`reverse_code`) quand c'est possible, ça ne coûte rien et si jamais une erreur a été faite lors de la migration, elle pourra être annulée par un retour en arrière.

A présent, appliquez la migration et vérifiez que la table Artiste n'est pas vide. Retournez à un état antérieur avec la commande `./manage.py migrate musiques 0003` et observez, d'une

part que ça marche car on a spécifié l'opération de retour en arrière et d'autre part que la table Artiste est vide. Réappliquez les migrations avec `./manage.py migrate musiques` .

4.23.10 Export/Import de données

Notre base doit contenir quelques données. Pour les exporter simplement au format *json* et disposer rapidement d'un petit jeu de données :

```
./manage.py dumpdata --format=json musiques > initial_musiques_data.
↪json
```

On pourra ainsi en disposer pour recréer les données avec :

```
./manage.py loaddata initial_musiques_data.json
```

4.23.11 Gitpod

Le code correspondant se trouve sur Gitlab (https ://gitlab.com/roza/gestmusic)

Le projet est disponible, prêt à être utilisé et complété sur gitpod :

(https ://gitpod.io/#https ://gitlab.com/roza/gestmusic)

4.23.12 Travail supplémentaire

Il reste encore plusieurs migrations à faire :

1. Ajouter un champ nullable de type `ForeignKey` dans la table Morceau
2. Effectuer une migration de données pour lier la clef étrangère à la bonne entrée de la table Artiste
3. Supprimer le champ artiste de la table Morceau et rendre la nouvelle clef étrangère non nullable

A vous d'écrire ces trois migrations !

Complétez aussi votre application en proposant un CRUD des Morceaux en utilisant des Vues basées sur des classes (comme des CreateView ou UpdateView) et avec un beau CSS. Ajoutez les tests correspondants (fonctionnels ou autres).

4.23.13 Code complet

Le code complet se trouve sur Github (https ://github.com/roza/django-musics) Il inclut des templates Bootswatch et des icones fournies par fontawesome.

Le projet est disponible, prêt à être utilisé et complété sur gitpod :

(https ://git-pod.io/#https ://github.com/roza/django-musics)

4.24 Modèle et admin de Django

Reprenons une nouvelle app *ToDo* dont le modèle est ci-dessous, pour pouvoir la gérer dans l'admin.

4.24.1 Modèle ToDo

```
#models.py
class ToDo(models.Model):
    name = models.CharField(max_length=250)
    begin_date = models.DateField(auto_now_add=True)
    end_date = models.DateField()
    done = models.BooleanField(default=False)

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

4.24.2 Migrations

4.24.3 Pour que Django prenne en compte les modifications de structure de la BD

```
./manage.py makemigrations
./manage.py sqlmigrate planning 0001
./manage.py migrate
```

— la commande *makemigrations* prépare les migrations
— la commande *sqlmigrate* décrit en sql la migration qui va se faire en base de données
— La commande *migrate* l'effectue

4.24.4 Et pour voir le nouveau modèle dans l'admin de Django

Complétons à présent le fichier *admin.py*

```
# admin.py
from django.contrib import admin
from planning.models import ToDo


class ToDoAdmin(admin.ModelAdmin):
    list_display=(
    'name', 'begin_date', 'end_date','done',
)

admin.site.register(ToDo,ToDoAdmin)
```

4.24.5 Créons un superuser dans Django pour accéder à l'admin

```
./manage.py createsuperuser
```

Puis visitez l'admin : Localhost:8000/admin/ (http ://localhost :8000/admin/) ous cette identité. Crééz quelques ToDo.

4.24.6 Améliorons un peu le rendu de l'admin des ToDo

```
@admin.register(ToDo)
class ToDoAdmin(admin.ModelAdmin):
        fieldsets = [
                ('ToDo Details', {'fields': ['name', 'done']}),
                ('ToDo Dates', {'fields': ['begin_date', 'end_date']}),
        ]

        readonly_fields = ('begin_date',)

        list_display = ('name', 'begin_date', 'end_date','done',)
        list_editable = ('done',)
        list_display_links = ('name', 'date_reviewed',)
        list_filter = ('done',)
        search_fields = ['name',]
```

Testez. Remarquez le décorateur qui sert à enregistrer directement le modèle ToDo dans l'admin et aussi les améliorations dans l'affichage des ToDo dans l'admin. Commitez.

4.24.7 Enrichissement du Modèle ToDo

Ajoutons par exemple le fait de mettre automatiquement *end_date* du Todo à *now* dès que le booléen *done* passe à *True*. Il faut pour cela surcharger la méthode *save()* de *Model* (avec ses args et kwargs, pourquoi à votre avis ?) et aussi ne pas oublier de faire appel à la méthode save() de la superclasse ...

```
#models.py
from django.db import models
from django.utils.timezone import now

class ToDo(models.Model):
    name = models.CharField(max_length=250)
    begin_date = models.DateField(auto_now_add=True)
    end_date = models.DateField()
    done = models.BooleanField(default=False)

    def __str__(self):
        return self.name

    def save(self, *args, **kwargs):
        if (self.done and self.end_date is None):
                self.end_date = now()

        super(ToDo, self).save(*args, **kwargs)
```

Réalisons les migrations.

```
./manage.py makemigrations
./manage.py migrate
```

Testez dans l'admin. Commitez.

4.25 Tester une application Python Django - TDD

Nous allons à présent nous attaquer à une problématique fondamentale dans toute application qu'elle soit Web, mobile ou autres : Les tests.

4.25.1 TDD

TDD veut dire *Test Driven Development* c'est à dire *Développement dirigé par les tests* C'est une démarche mise en avant en *Méthodologie Agile* Elle consiste en général en l'application des points suivants :

- écrire un test
- vérifier qu'il échoue (car le code qu'il teste n'existe pas)
- commiter votre code
- écrire juste le code suffisant pour passer le test
- vérifier que le test passe
- faire un commit
- procéder à un refactoring du code, c'est-à-dire l'améliorer en gardant les mêmes fonctionnalités.
- vérifier que le test passe toujours
- commiter
- etc.

4.25.2 Intérêt de la démarche :

Les avantages principaux de cette démarche sont :

- Préciser au mieux les spécifications du code et l'API envisagée
- Ceci oblige à faire des choix de conception qui restent parfois trop dans le flou au début du développement
- Plus tard, disposer d'une large base de tests est une riche pour une application car elle permet de vérifier à tout moment que les tests installés ne sont pas mis en défaut par de nouveaux développements ou des refactoring de code

Tous les langages de programmation disposent de Frameworks de tests. Par exemple Java offre *JUnit*, PHP quand à lui propose *PHPUnit*. Python propose *unittest* et permet aussi d'utiliser facilement la librairie selenium pour réaliser des tests fonctionnels.

4.25.3 Tests fonctionnels

Ces tests visent à vérifier le comportement global de notre application, pour mimer la navigation d'un client web sur le site.

Prérequis

Installer selenium via pip dans votre virtualenv :

```
pip install selenium
```

Eventuellement, installer selenium-standalone, qui offre plus de possibilités pour utiliser des browsers différents.

```
yarn add selenium-standalone@latest -g
selenium-standalone install
selenium-standalone start
```

Pour vérifier les paquets node globalement installés

```
npm list -g --depth=0
```

Pour vérifier les paquets node localement installés

```
npm list  --depth=0
```

Puis écrivons notre premier test fonctionnel dans le dossier Tests :

```
from selenium import webdriver
import time

browser = webdriver.Chrome(executable_path='./chromedriver')
time.sleep(3)
browser.get('http://localhost:8000')

assert 'Django' in browser.title
browser.quit()
```

Pour tester (dans le terminal où venv est actif) :

```
python functional-test1.py
```

ou avec *manage.py* de django :

```
./manage.py test functional-test1.py
```

Danger : python est python3X (celui du venv que vous avez créé) Vous devrez remettre temporairement DEBUG = FALSE pour que ce test passe. Pourquoi ?

Pour l'instant notre test échoue si notre serveur est arrêté ou si il est en mode DEBUG. Commitons.

Relançons le serveur ...

```
./manage.py runserver
```

Puis relançons le test et constatons qu'il passe. Commitons lorsque c'est le cas.

4.25.4 Configuration complémentaire de la debug toolbar

Ajout de debug-toolbar

On peut ajouter django-debug-toolbar à notre venv pour voir cet outil à l'oeuvre. (les autres réglages nécessaires seront faits dans settings.py)

```
pip install django-debug-toolbar
```

Corrigez, les requirements :

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

Commitez.

Complétons les settings de notre projet

Ajoutons la *debug_toolbar* aux INSTALLED_APPS Puis les panels que l'on souhaite voir afficher, le middleware nécessaire, et les IPs autorisées. On a déjà configuré la langue, le timezone, le format des dates, etc.

```
DEBUG = True

 # ../..

if DEBUG:
    INTERNAL_IPS = ('127.0.0.1', 'localhost',)

    MIDDLEWARE += (
                'debug_toolbar.middleware.DebugToolbarMiddleware',
                )

    INSTALLED_APPS += (
                'debug_toolbar',
                )


    DEBUG_TOOLBAR_PANELS = [
                'debug_toolbar.panels.versions.VersionsPanel',
                'debug_toolbar.panels.timer.TimerPanel',
                'debug_toolbar.panels.settings.SettingsPanel',
                'debug_toolbar.panels.headers.HeadersPanel',
                'debug_toolbar.panels.request.RequestPanel',
                'debug_toolbar.panels.sql.SQLPanel',
                'debug_toolbar.panels.staticfiles.StaticFilesPanel',
                'debug_toolbar.panels.templates.TemplatesPanel',
                'debug_toolbar.panels.cache.CachePanel',
```

(suite sur la page suivante)

(suite de la page précédente)

```
                'debug_toolbar.panels.signals.SignalsPanel',
                'debug_toolbar.panels.logging.LoggingPanel',
                'debug_toolbar.panels.redirects.RedirectsPanel',
                'debug_toolbar.panels.profiling.ProfilingPanel',
    ]

    DEBUG_TOOLBAR_CONFIG = {
                'INTERCEPT_REDIRECTS': False,
        }



# ../..

LANGUAGE_CODE = 'fr-fr'
TIME_ZONE = 'Europe/Paris'
```

On complète urls.py

Dans le fichier général *urls.py*, on intègre la debug_toolbar :

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

urlpatterns = [
        path('admin/', admin.site.urls),
        path('lesTaches', include('lesTaches.urls'))
]
# Pour la debug_toolbar:
from django.conf import settings

if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        path('__debug__/', include(debug_toolbar.urls)),
                  ] + urlpatterns
```

La doc se trouve là (https ://django-debug-toolbar.readthedocs.io/en/latest/index.html)

et les panels complémentaires que l'on peut installer sont documentés ici. (https ://django-debug-toolbar.readthedocs.io/en/latest/panels.html)

Attention : beaucoup de ces panels complémentaires sont obsolètes et non maintenus !

Enfin le dépôt Github (https ://github.com/jazzband/django-debug-toolbar) de l'extension python contient la version la plus à jour du code.