

1. créer un environnement virtuel

```
python -m venv env  
env/Scripts/activate
```

2. Mettre à jour pip install

```
pip install --upgrade pip
```

3. installer le package django

```
pip install django==3.1.6
```

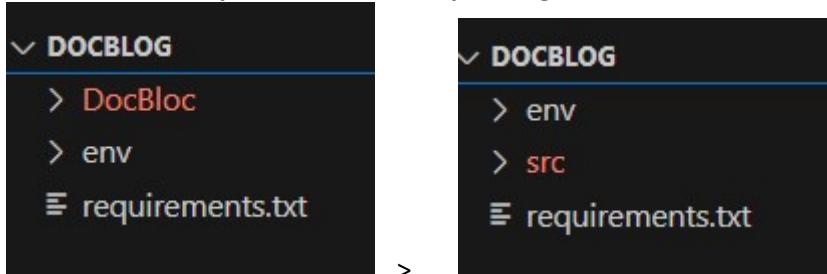
4. Vérifier que django soit bien installé

```
python -m django --version
```

5. Créer un projet Django

```
django-admin startproject DocBloc
```

6. renommer uniquement le dossier parent généré 'DocBloc' en 'src' (uniquement celui-ci !)



7. Pour appliquer toute les migrations exécuter la commande ci-dessous avant de lancer le serveur

> sert aussi créer aussi la BDD « db.sqlite3 », il faut ajouter ce fichier à .gitignore
python manage.py migrate

8. Pour lancer le serveur :

```
cd src  
python manage.py runserver
```

9. Pour afficher la page gérée par Django, Copier-coller le lien surisé en jaune dans un navigateur

```
Django version 3.1.6, using settings 'DocBloc.settings'  
Starting development server at http://127.0.0.1:8000/  
Quit the server with CTRL-BREAK.
```

Le site n'est pas accessible si le serveur n'est pas lancé.

Il n'est pas utile de relancer le serveur à chaque modification, le serveur charge toutes les MAJ automatiquement

Sauf si les modifications n'apparaissent pas sur le site ! comme une modification de la BDD

10. Pour voir sur quel fichier url pointe le projet Django

```
settings.py X
src > DocBloc > settings.py > ...
51
52     # pointe vers le fichier des url
53 ROOT_URLCONF = 'DocBloc.urls'
```

On peut voir que le projet django pointe vers le fichier urls présent dans le dossier DocBloc

11. Créer le chemin d'une vue

```
urls.py X
1. import the include() function: from django.urls import include
2. Add a URL to urlpatterns: path('blog/', include(
15     "nidal.urls",
16 ))
17
18 from django.contrib import admin
19 from django.urls import path
20 from django.views.defaults import server_error
21
22 urlpatterns = [
23     # nidal représente le chemin d'url
24     # admin.site.urls représente la vue
25     path("nidal/", admin.site.urls),
26     path('bonjour/comment/ca/va', server_error),
27 ]
```

Dans la liste urlpatterns créer une nouvelle fonction path(« chemin », vue à appeler)

12 Créer une nouvelle vue de test

- créer un fichier view

```
DocBlog C:\Users\nidal\Desktop\DocBlog
> env library root
src
  DocBloc
    __init__.py
    asgi.py
    settings.py
    urls.py
    views.py
    wsgi.py
1     """test view"""
2
3     from django.http import HttpResponse
4
5
6     def vue_de_test(request):
7         return HttpResponse("<h1>Vue du test</h1>")
```

- Retourner sur le fichier urls et créer le chemin test renvoyant vers la vue « vue_de_test »

```

1  """DocBloc URL Configuration..."""
16 from django.contrib import admin
17 from django.urls import path
18 from django.views.defaults import server_error
19 from .views import vue_de_test
20
21 urlpatterns = [
22     # nidal représente le chemin d'url
23     # admin.site.urls représente la vue
24     path("nidal/", admin.site.urls),
25     path('bonjour/comment/ca/va/', server_error),
26     path("test/", vue_de_test)
27 ]
28

```

13. fonction APPEND_SLASH=True :

Permet de rediriger automatiquement le vers le chemin se terminant avec un slash si le chemin entré ne se termine par un slash

14. Pour créer la vue de la page de page d'accueil (connecter une vue à une url)

- Créer le fichier view et la fonction index retournant un élément HttpResponse(html...)

```

1  """test view"""
2  from django.http import HttpResponseRedirect
3
4
5  def vue_de_test(request):
6
7      # request permet de récupérer la requête du navigateur
8
9  def index(request):
10
11      return HttpResponseRedirect("<h1>Bonjour, bienvenue sur mon site</h1>")
12

```

- Créer l'url de la page d'accueil à l'aide de «», name sert de légende et d'indication de cette url, c'est une bonne pratique

```

1  """DocBloc URL Configuration..."""
16 from django.contrib import admin
17 from django.urls import path
18 from django.views.defaults import server_error
19 from .views import vue_de_test, index
20
21 urlpatterns = [
22     # nidal représente le chemin d'url
23     # admin.site.urls représente la vue
24     # si on met juste "" dans path en argument cela crée le chemin de la page d'accueil
25     path("", index, name='index'),
26     path("admin/", admin.site.urls),
27 ]
28

```

Et le tour est joué, la page d'accueil est créé ! et affiche « Bonjour... »

15. Pour faire appel à des vue HTML

- Créer un dossier « Templates » dans le dossier projet django ici « DocBlog »

- A l'intérieur de ce dossier créer un fichier index.html

The screenshot shows the PyCharm interface with the project 'DocBlog' open. The left sidebar shows the file structure: DocBlog (env, library root, src, DocBloc, templates). Inside 'templates' is an 'index.html' file. The right pane displays the code for 'index.html':

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Mon site wew</title>
</head>
<body>
    <h1>Bonjour, Bienvenue sur mon site !</h1>
</body>
</html>
```

- Créer un fichier view.py en faisant appel à la fonction render dans le return de la fonction index (alt+enter pour l'auto complétion de l'import)

Le chemin intégré dans la fonction path correspond au chemin relatif présent dans le settings > DIRS []

The screenshot shows the PyCharm interface with the project 'DocBlog' open. The left sidebar shows the file structure: DocBlog (env, library root, src, DocBloc, templates, index.html), urls.py, views.py, settings.py, etc. The right pane displays the code for 'views.py':

```
"""test view"""
from django.shortcuts import render

# request permet de récupérer la requête du navigateur
def index(request):
    return render(request, "index.html")
```

- Ajouter à liste DIRS [] le chemin relatif au dossier « templates » auto calculé de la variable BASE_DIR

```
# Calcul automatiquement le chemin absolu de fichier avec deux parents en moins
BASE_DIR = Path(__file__).resolve().parent.parent
```

Attention os.path déconne sous pycharm

The screenshot shows the PyCharm interface with the project 'DocBlog' open. The left sidebar shows the file structure: DocBlog (env, library root, src, DocBloc, templates, index.html), urls.py, views.py, settings.py, etc. The right pane displays the code for 'settings.py':

```
...
# pointe vers le fichier des url
ROOT_URLCONF = "DocBloc.urls"

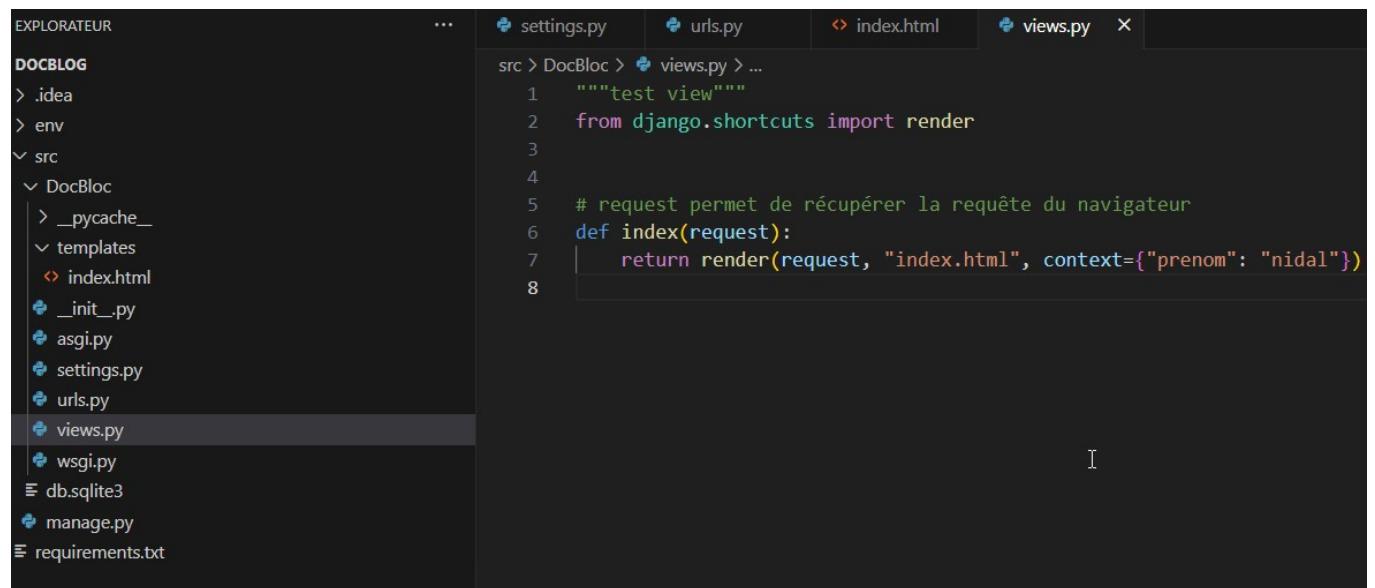
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [os.path.join(BASE_DIR, "DocBlog/templates")],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                ...
            ],
            "debug": DEBUG
        }
    },
]
```

A line of code in the 'DIRS' section of the TEMPLATES dictionary is highlighted in yellow: `"DIRS": [os.path.join(BASE_DIR, "DocBlog/templates")],`

```
os.path.join(BASE_DIR, "DocBloc/templates/")
```

16. insérer du texte dans le html via la vue à l'aide du dictionnaire context

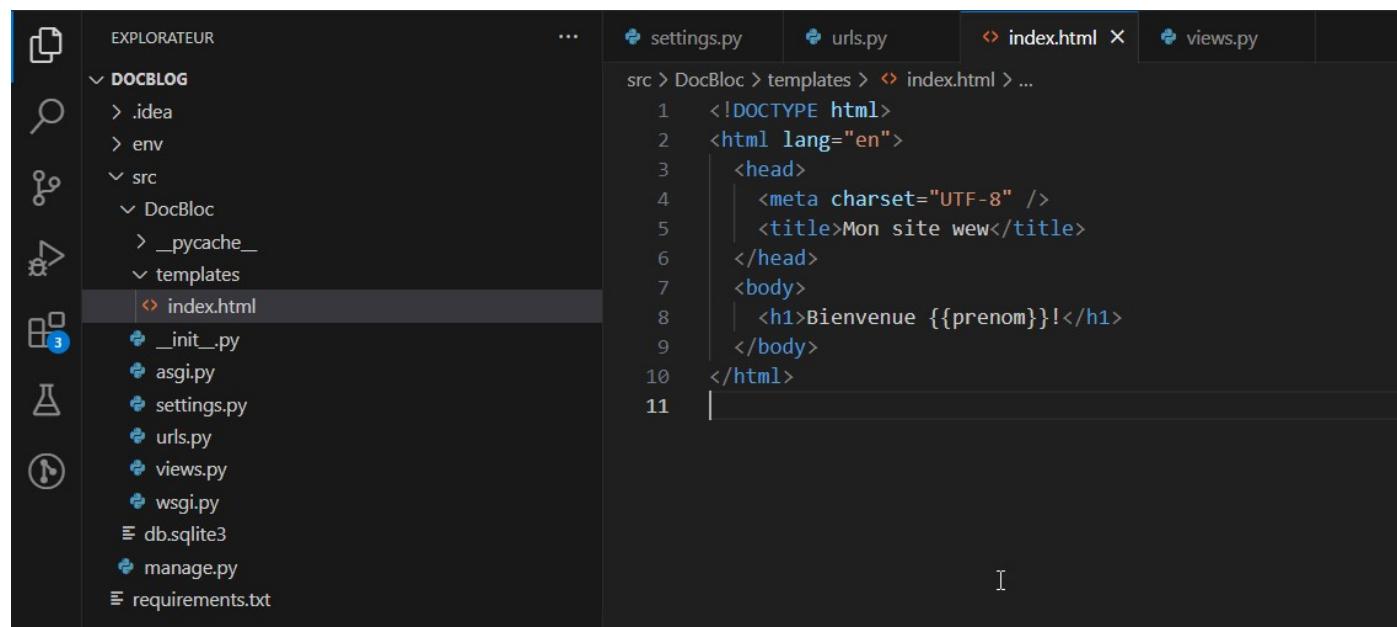
- Insérer le deuxième paramètre context avec la paire clé : valeur



```
EXPLORATEUR ... settings.py urls.py index.html views.py
DOCBLOG
> .idea
> env
src
  > DocBloc
    > __pycache__
    > templates
      > index.html
    > __init__.py
    > asgi.py
    > settings.py
    > urls.py
    > views.py
    > wsgi.py
  > db.sqlite3
  > manage.py
requirements.txt
```

```
src > DocBloc > views.py > ...
1  """test view"""
2  from django.shortcuts import render
3
4
5  # request permet de récupérer la requête du navigateur
6  def index(request):
7      return render(request, "index.html", context={"prenom": "nidal"})
8
```

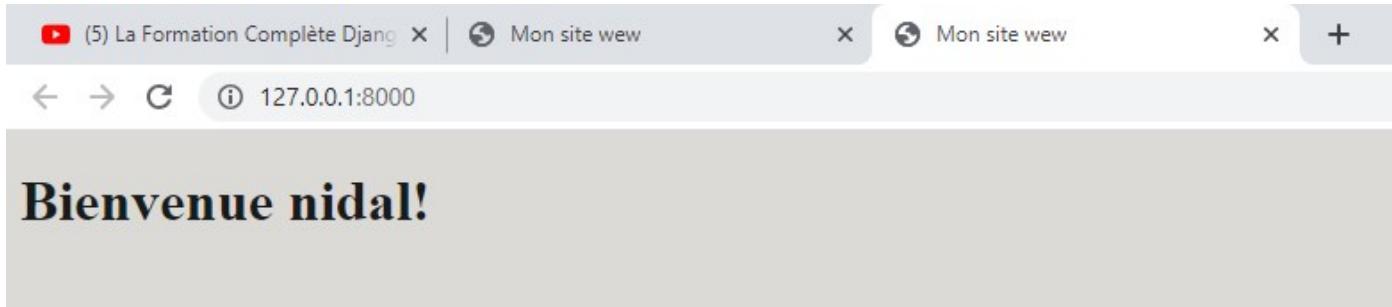
- Insérer la double accolade avec la clé du dictionnaire « context » que le souhaite voir apparaître dans le html



```
EXPLORATEUR ... settings.py urls.py index.html views.py
DOCBLOG
> .idea
> env
src
  > DocBloc
    > __pycache__
    > templates
      > index.html
    > __init__.py
    > asgi.py
    > settings.py
    > urls.py
    > views.py
    > wsgi.py
  > db.sqlite3
  > manage.py
requirements.txt
```

```
src > DocBloc > templates > index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="UTF-8" />
5          <title>Mon site wew</title>
6      </head>
7      <body>
8          <h1>Bienvenue {{prenom}}!</h1>
9      </body>
10     </html>
11
```

- Résultat :



- Possibilité de mettre en majuscule ! (avec « |upper ») | est l'opérateur pip

```

EXPLORATEUR
...
settings.py urls.py index.html X views.py
src > DocBloc > templates > index.html ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4  |<meta charset="UTF-8" />
5  |<title>Mon site wew</title>
6  </head>
7  <body>
8  |<h1>Bienvenue {{prenom|upper}}!</h1>
9  </body>
10 </html>
11

```

Nous pouvons changer la clé du dictionnaire « convert » tout comme sa valeur ! (ex avec date)

```

EXPLORATEUR
...
settings.py urls.py index.html X views.py
src > DocBloc > views.py ...
1  """test view"""
2  from datetime import datetime
3
4  from django.shortcuts import render
5
6  date = datetime.today()
7  print("la date d'aujourd'hui : ", date)
8  print(type(date))
9
10
11 # request permet de récupérer la requête du navigateur
12 def index(request):
13     return render(request, "index.html", context={"date": date})
14

```

Nous pouvons nous apercevoir que Django est assez complet pour afficher un objet complexe <class : datetime>

`{{date|date}}`

Permet d'afficher la date dans un format lisible

Les filters applicables sur le html :

- `|upper > majuscule`
- `|date > affiche la date dans un format lisible`

Pour formater la date jour-mois année par exemple (attention pas d'espace !)

```
EXPLORATEUR
...
src > DocBloc > templates > index.html ...
1  <!DOCTYPE html>
2  <html lang="en">
3  | <head>
4  | | <meta charset="UTF-8" />
5  | | <title>Mon site wew</title>
6  | </head>
7  | <body>
8  | | <h1>Bienvenue!</h1>
9  | | <h2>Aujourd'hui nous sommes le : {{date|date:"d F y"}}.</h2>
10 | </body>
11 </html>
12
```

Pour formater la date et l'heure :

```
EXPLORATEUR
...
src > DocBloc > templates > index.html ...
1  <!DOCTYPE html>
2  <html lang="en">
3  | <head>
4  | | <meta charset="UTF-8" />
5  | | <title>Mon site wew</title>
6  | </head>
7  | <body>
8  | | <h1>Bienvenue!</h1>
9  | | <h2>Aujourd'hui nous sommes le : {{date|date:"d F y H:m:s"}}.</h2>
10 | </body>
11 </html>
12
```

Pour mettre le site en Français (dont la date) :

```
EXPLORATEUR
...
src > DocBloc > settings.py ...
98     },
99     {
100         "NAME": "django.contrib.auth.password_validation.NumericPasswordValidator"
101     },
102 ]
103
104
105 # Internationalization
106 # https://docs.djangoproject.com/en/3.1/topics/i18n/
107
108 LANGUAGE_CODE = "fr-FR"
109
110 TIME_ZONE = "UTC"
111
112 USE_I18N = True
113
114 USE_L10N = True
115
```

Pour sélectionner une autre langue c'est possible !

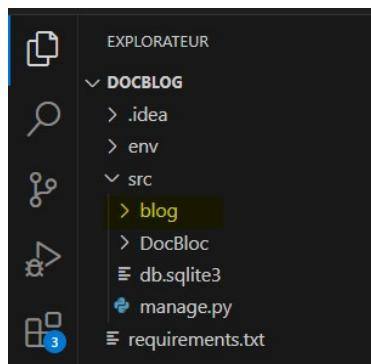
<http://www.i18nguy.com/unicode/language-identifiers.html>

Pour créer une nouvelle application :

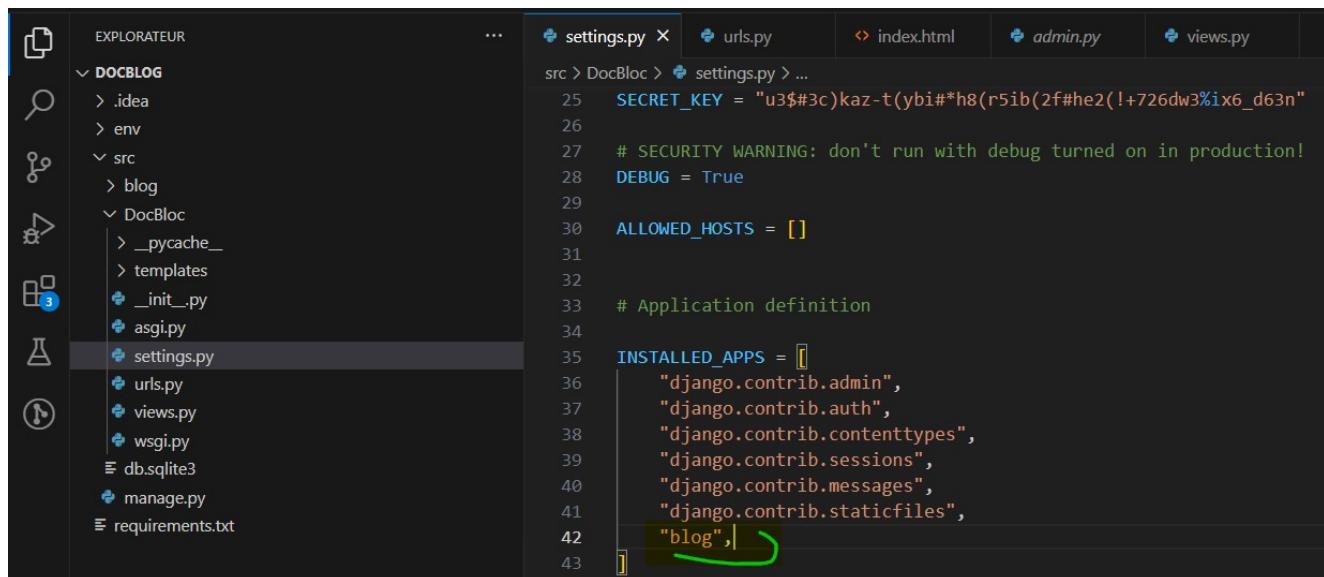
`python manage.py startapp nom_application`

● (env) PS C:\Users\nidal\Desktop\DocBlog\src> **python manage.py startapp blog**

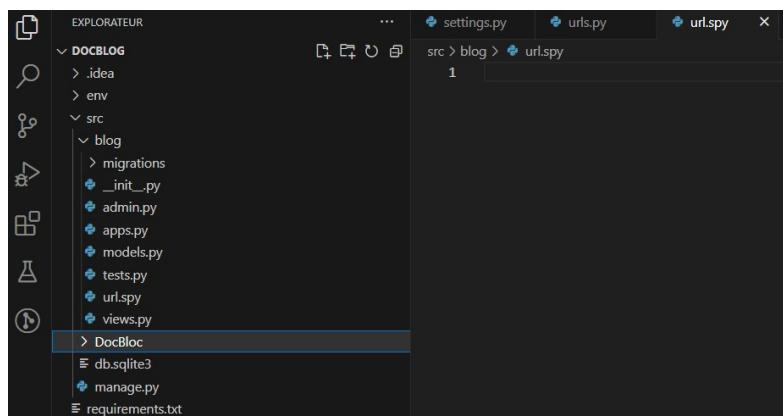
- Un nouveau dossier dans src apparaît avec le nom de l'application :



- Ajouter le nom de l'application créée dans le fichier settings.py en l'ajoutant à la fin de la liste, pour qu'elle soit la dernière à se charger, c'est une bonne pratique ! INSTALLED_APP []



- Créer un fichier url.py à la racine du dossier de l'application blog créé



- Créer une nouvelle vue dans l'application blog sur le fichier « views.py »

```

EXPLORATEUR
...
src > blog > views.py ...
1   from django.shortcuts import render
2   from django.http import HttpResponseRedirect
3
4
5
6
7   # Create your views here.
8
9
10 def index(request):
11     return HttpResponseRedirect("<h1>Le Blog</h1>")
12
13

```

- Dans le fichier « urls.py » présent dans le dossier de l’application, créer une liste urlpattern = [] tout comme le fichier urls.py présent dans le dossier du projet django

```

EXPLORATEUR
...
src > blog > urls.py ...
1   from django.contrib import admin
2   from django.urls import path
3   from .views import index
4
5   urlpatterns = [
6       # nidal représente le chemin d'url
7       # admin.site.urls représente la vue
8       # si on met juste "" dans path en argument cela crée le chemin de la page d'accueil
9       # name contient une sorte de commentaire de l'url, si celle ci est dans une app, faire
10      # figurer le nom de l'app > name="blog-index"
11
12      # l'url sera http://127.0.0.1:8000/blog/
13      # path("", index, name="blog-index"),
14
15      # l'url sera http://127.0.0.1:8000/blog/page1/
16      # path("page1/", index, name="blog-index"),
17
18      # path("admin/", admin.site.urls),
19
20 ]

```

- Retourner dans le fichier urls.py du projet django parent puis venir inclure le fichier urls.py de l’app « blog » pour que ce dernier soit pris en compte (car toutes urls doivent être présentes dans le fichier parents urls.py car le projet django renvoi les urls présent dans la variable ROOT_URLCONF du fichier settings.py)

```

EXPLORATEUR
...
src > DocBloc > urls.py ...
1   Add a URL to urlpatterns:  path(), views.home, name='home')
2
3 Class-based views
4   1. Add an import:  from other_app.views import Home
5   2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
6
7 Including another URLconf
8   1. Import the include() function: from django.urls import include, path
9   2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
10
11 from django.contrib import admin
12 from django.urls import path, include
13 from .views import index
14
15 urlpatterns = [
16     # nidal représente le chemin d'url
17     # admin.site.urls représente la vue
18     # si on met juste "" dans path en argument cela crée le chemin de la page d'accueil
19     path("blog/", include("blog.urls")),
20     path("", index, name="index"),
21     path("admin/", admin.site.urls),
22
23 ]

```

- Pour supprimer le conflit de nommage entre les template du dossier parent django et les templates des application :

On crée dans chaque dossier template, un dossier enfant du nom du dossier parent de templates et on rajoute le dossier que l’on vient d’ajouter entre le dossier templates et « index.html » dans la vue et le tour est joué

➤ Pour le projet parent Django nommé DocBlog

The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar displays the project structure under 'DOCBLOG'. A green circle highlights the 'templates\DocBlog' folder. The main editor window shows the 'views.py' file with the following code:

```

1  """test view"""
2
3  from datetime import datetime
4
5  from django.shortcuts import render
6
7
8  # request permet de récupérer la requête du navigateur
9  def index(request):
10     return render(request, "DocBlog/index.html", context={"date": datetime.today()})
11

```

➤ Pour le projet enfant (l'application blog)

The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar displays the project structure under 'DOCBLOG'. A green circle highlights the 'templates\blog' folder. The main editor window shows the 'views.py' file with the following code:

```

1  from django.shortcuts import render
2  from django.http import HttpResponseRedirect
3
4
5  # Create your views here.
6
7
8  def index(request):
9      return render(request, "blog/index.html")
10

```

Si l'on nomme toujours les dossier « templates » même dans les applications ceci permet à python d'aller également chercher les « templates » présent dans les application enfant donc très pratique mais attention au conflit de nommage que l'on peut éviter grâce à la création de sous dossier dans le dossier templates. Tout ceci est possible grâce à la variable APP_DIRS est à true.

The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar displays the project structure under 'DOCBLOG'. A green circle highlights the 'templates\blog' folder. The main editor window shows the 'settings.py' file with the following code:

```

src > DocBloc > settings.py ...
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72

```

The code includes a section for 'TEMPLATES' configuration:

```

TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [os.path.join(BASE_DIR, "docBloc/templates/")],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    },
]

```

- Il est possible de récupérer dans une variable une partie de l'url entré par l'utilisateur dans la barre d'adresse si la formule adéquate est présente dans le fichier « urls.py »

```

EXPLORATEUR
...
views.py    urls.py    index.html
src > blog > urls.py > ...
1   from django.contrib import admin
2   from django.urls import path
3   from .views import index, article
4
5   urlpatterns = [
6       # nidal représente le chemin d'url
7       # admin.site.urls représente la vue
8       # si on met juste "" dans path en argument cela crée le chemin de la page d'accueil
9       # name contient une sorte de commentaire de l'url, si celle ci est dans une app, faire
10      # figurer le nom de l'app > name="blog-index"
11      # l'url sera http://127.0.0.1:8000/blog/
12      path("", index, name="blog-index"),
13      # l'url sera http://127.0.0.1:8000/blog/article_0X
14      # si l'on récupérer un non nombre ou un chaîne de caractère on utilise <> dans le chemin
15      # <int> ou <str>
16      path("article-<str:numero_article>/", article, name="blog-article"),
17      # l'url sera http://127.0.0.1:8000/blog/page1/
18      path("page1/", index, name="blog-index"),
19      # path("admin/", admin.site.urls),
20  ]
21

```

Lorsque l'utilisateur entrera l'adresse :

<http://127.0.0.1:8000/blog/article->

Tout ce que l'utilisateur tapera après « article-» sera stocké dans la variable string « numero_article »

Il est possible de la stocker dans une variable int mais si l'utilisateur entre une chaîne de caractère : python affichera une erreur dans la console

Et exploitable dans la vue :

```

EXPLORATEUR
...
views.py    urls.py    index.html
src > blog > views.py > ...
1   from django.shortcuts import render
2   from django.http import HttpResponseRedirect
3
4   # Create your views here.
5
6
7
8   def index(request):
9       return render(request, "blog/index.html")
10
11  def article(request, numero_article):
12      print(numero_article)
13      return render(request, "blog/article_01.html")
14
15
16

```

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL

Performing system checks...

System check identified no issues (0 silenced).

July 07, 2023 - 23:52:19

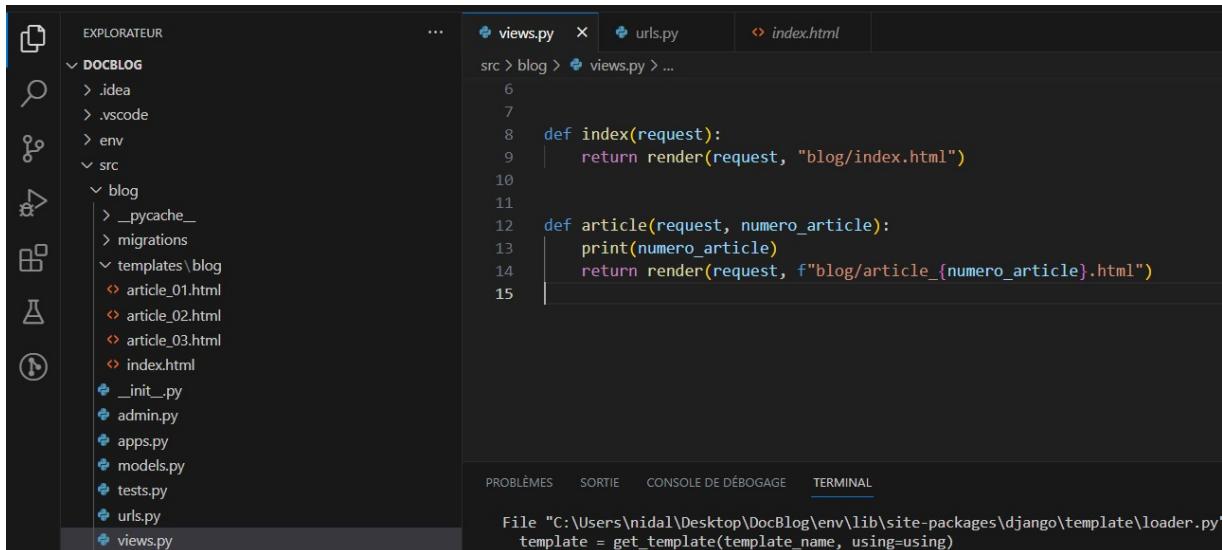
Django version 3.1.6, using settings 'DocBloc.settings'

Starting development server at <http://127.0.0.1:8000/>

Quit the server with CTRL-BREAK.

[07/Jul/2023 23:54:18] "GET /blog/article-18/ HTTP/1.1" 200 236

A l'aide d'une f-string il est possible d'afficher dynamique un template via la valeur entré par l'utilisateur



The screenshot shows the VS Code interface with the following file structure in the Explorer sidebar:

- DOCLOG
- .idea
- .vscode
- env
- src
 - blog
 - _pycache_
 - migrations
 - templates\blog
 - article_01.html
 - article_02.html
 - article_03.html
 - index.html
 - __init__.py
 - admin.py
 - apps.py
 - models.py
 - tests.py
 - urls.py
 - views.py

The main editor window displays the following Python code in `views.py`:

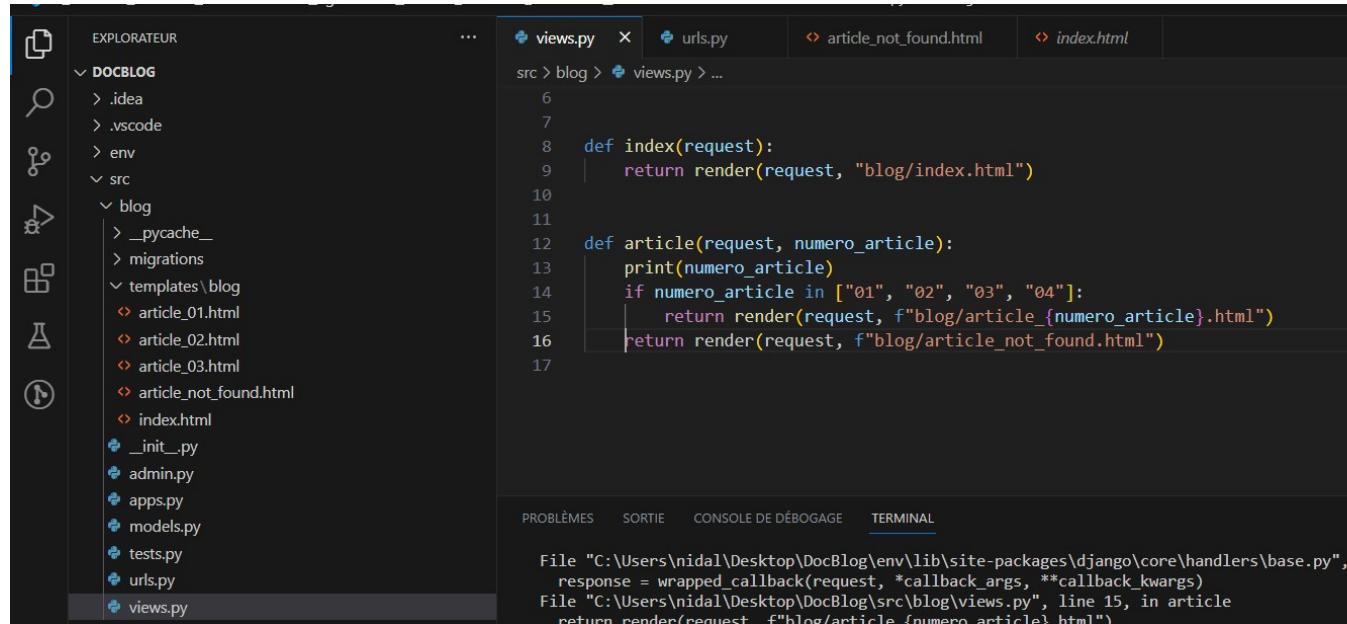
```
def index(request):
    return render(request, "blog/index.html")

def article(request, numero_article):
    print(numero_article)
    return render(request, f"blog/article_{numero_article}.html")
```

The terminal below shows a command being run:

```
File "C:\Users\nidal\Desktop\DocBlog\env\lib\site-packages\django\template\loader.py"
template = get_template(template_name, using=using)
```

➤ Gérer les erreurs avec une condition if sur numero_article:



The screenshot shows the VS Code interface with the same file structure as the previous screenshot.

The main editor window displays the following Python code in `views.py`:

```
def index(request):
    return render(request, "blog/index.html")

def article(request, numero_article):
    print(numero_article)
    if numero_article in ["01", "02", "03", "04"]:
        return render(request, f"blog/article_{numero_article}.html")
    return render(request, f"blog/article_not_found.html")
```

The terminal below shows an error message:

```
File "C:\Users\nidal\Desktop\DocBlog\env\lib\site-packages\django\core\handlers\base.py",
response = wrapped_callback(request, *callback_args, **callback_kwargs)
File "C:\Users\nidal\Desktop\DocBlog\src\blog\views.py", line 15, in article
return render(request, f"blog/article_{numero_article}.html")
```

Créer une feuille de style css dans une application django dédié à un templates

- Crée un dossier « static » dans le répertoire de l'application
- Dans celui créer un dossier « css »
 - Puis enfin créer un fichier « style.css »
 - Mettre {%- load static %} en début de html
 - Générer le lien de manière automatique avec {%- static 'path dans static' %} dans html

```

EXPLORATEUR
...
src > blog > templates > blog > index.html > ...
1   <!-- mettre load static en début de html pour être en mesure
2   |   d'utiliser lien généré par django allant vers static -->
3   {%- load static %}
4   <!DOCTYPE html>
5   <html lang="en">
6       <head>
7           <meta charset="UTF-8" />
8           <meta name="viewport" content="width=device-width, initial-scale=1.0" />
9           <title>Le Blog</title>
10          <!-- Django permet de générer des lien grace à {% static 'path dans static' %}-->
11          <link rel="stylesheet" href="{% static 'css/style.css' %}" />
12      </head>
13      <body>
14          <h1>Bienvenue sur la vue du Blog</h1>
15      </body>
16  </html>
17

```

Par défaut, django va chercher uniquement les dossier static présent dans une application ici 'blog'

Pour être en mesure de faire appel à un fichier static présent dans le dossier parent il sera nécessaire d'ajouter la variable STATICFILES_DIRS dans le fichier setting :

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

```

EXPLORATEUR
...
src > DocBlog > settings.py > ...
114     TIME_ZONE = 'UTC'
115     USE_I18N = True
116
117     USE_L10N = True
118
119     USE_TZ = True
120
121
122     # Static files (CSS, JavaScript, Images)
123     # https://docs.djangoproject.com/en/3.1/howto/static-files/
124
125     STATIC_URL = "/static/"
126     STATICFILES_DIRS = [os.path.join(BASE_DIR, "DocBlog/static/")]
127
128

```

Fin cours Docstring

Début cours OC

Pour créer un nouveau model dans une application django ici dans l'application « listings » :

- Aller dans le fichier models.py présent dans le dossier de l'application « listings »
- Créer une classe qui hérite models.Model, un classe django, puis lui attribuer un attribut de classe
- Création d'une classe « Band » ayant pour attribut « name »

The screenshot shows a code editor with the following structure:

- Left sidebar: EXPLORATEUR, showing the project structure:
 - DJANGO-WEB-APP
 - .idea
 - env
 - merchex
 - listings
 - _pycache_
 - migrations
 - __init__.py
 - admin.py
 - apps.py
 - models.py
 - tests.py
 - views.py
 - merchex
 - db.sqlite3
 - manage.py
 - .gitignore
 - requirements.txt
 - Top tabs: views.py, models.py (selected), 0001_initial.py, .gitignore
 - Code area (models.py):

```
from django.db import models

class Band(models.Model):
    name = models.fields.CharField(max_length=100)
```

- Charfield est un champ qui stocke les données de type string, ce qu'il faut pour 'name'

Les type de donnée différent que possède django pour la BDD :

- CharField : str (l'argument max_length est obligatoire)
- IntegerField : int (doit avoir un max et un min)
- DateField : date
- BooleanField : bool (défaut= true permet d'avoir une valeur par défaut)
- UrlField : pour les liens (comprends un test qui vérifie s'il s'agit bien d'un lien
 - (null=True, blank=True) permet d'accepter les valeurs nulles

Exemple avec plusieurs champs :

```
from django.core.validators import MaxValueValidator, MinValueValidator
...

class Band(models.Model):
    name = models.fields.CharField(max_length=100)
    genre = models.fields.CharField(max_length=50)
    biography = models.fields.CharField(max_length=1000)
    year_formed = models.fields.IntegerField(
        validators=[MinValueValidator(1900), MaxValueValidator(2021)])
    active = models.fields.BooleanField(default=True)
    official_homepage = models.fields.URLField(null=True, blank=True)
```

Définir une liste de valeur pour un attribut (ici genre) :

- Dans le fichier « models.py » ajoutons une classe imbriqué que l'on nommera Genre pour pouvoir avoir une liste déroulante pour l'attribut genre, notons que la classe Genre hérite de la classe models.TextChoices, c'est ce qui permet d'avoir l'option liste déroulante.

- Déterminons les clés que l'on souhaite voir apparaître sur le front et les valeurs stockées dans la BDD ex (HIP_HOP sera sélectionnable et HH sera stocké dans la BDD)
- Enfin pour la variable 'genre' nous attribuons le 1er paramètre `choices=Genre.choices` et le 2nd `max_length=5` qui permet à django d'aller piocher les éléments sélectionnables dans la classe Genre et de déterminer une longueur maximum à 5 caractères pour ce qui sera stocké dans la BDD.

The screenshot shows a code editor with a file tree on the left and the `models.py` file open on the right. The file tree shows a project structure with a `DJANGO-WEB-APP` folder containing `.idea`, `env`, and a `merchex` folder. Inside `merchex` are `listings`, `migrations`, `static\listings`, `templates\listings`, `__init__.py`, `admin.py`, `apps.py`, and the current file `models.py`. The `models.py` file contains Python code defining a `Band` model with various fields like `name`, `genre`, `biography`, `year_formed`, `active`, and `official_homepage`.

```

EXPLORATEUR ...
DJANGO-WEB-APP ...
merchex > listings > models.py ...
models.py
1 from django.db import models
2 from django.core.validators import MaxValueValidator, MinValueValidator
3
4
5 class Band(models.Model):
6     class Genre(models.TextChoices):
7         """sert pour la création d'un liste à dérouler ou l'on peut choisir uniquement un genre"""
8
9         HIP_HOP = "HH"
10        SYNTH_POP = "SP"
11        ALTERNATIVE_ROCK = "AR"
12        RAP = "RP"
13
14        name = models.fields.CharField(max_length=100)
15        genre = models.fields.CharField(choices=Genre.choices, max_length=5)
16        biography = models.fields.CharField(max_length=1000)
17        year_formed = models.fields.IntegerField(
18            validators=[MinValueValidator(1900), MaxValueValidator(2021)])
19
20        active = models.fields.BooleanField(default=True)
21        official_homepage = models.fields.URLField(null=True, blank=True)
22

```

Ajoutons maintenant les nouveaux attributs (à refaire en cas de changement sur la classe model pour MAJ la BDD)

- Il n'est pas nécessaire avec django de créer une fonction constructeur, django le fait pour nous
- Migration signifie : mise à jour de la BDD d'un état à un autre
- Entrer la CLI : `python manage.py makemigrations` (cette CLI va analyser le fichier `models.py` pour y déceler toute modification et déterminer le type de migration à générer : cad écrire l'instruction de création de la BDD « `Band` »)

```

1 # shell
2
3 (env) ~/projects/django-web-app/merchex
4 → python manage.py makemigrations
5 python manage.py makemigrations
6 Migrations for 'listings':
7   listings/migrations/0001_initial.py
8     - Create model Band

```

- Une fois que nous avons notre migration créée (instruction) nous devons exécuter ces instructions dans la BDD avec la CLI : `python manage.py migrate` (à refaire en cas de changement sur la classe model pour MAJ la BDD)

```

1 (env) ~/projects/django-web-app/merchex
2 → python manage.py migrate
3 Operations to perform:
4 Apply all migrations: admin, auth, contenttypes, listings, sessions
5 Running migrations:
6 Applying listings.0001_initial... OK

```

- Ouvrir le shell django : `python manage.py shell`

```
1 (env) ~/projects/django-web-app/merchex
2 → python manage.py shell
3 >>>
```

À l'invite du shell (`>>>`), tapez le code suivant pour importer notre modèle de groupe :

```
1 >>> from listings.models import Band
```

Appuyez sur `Entrée` pour exécuter cette ligne.

Ensuite, nous allons créer une nouvelle instance du modèle `Band` :

```
1 >>> band = Band()
2 >>> band.name = 'De La Soul'
```

Jetez un coup d'oeil à l'état actuel de l'objet en tapant simplement `band` :

```
1 >>> band
2 <Band: Band object (None)>
```

Le shell nous dit que nous avons un objet `band`, mais l'id est `None`, il n'a pas encore d'id.

Maintenant, sauvegardons cet objet dans la base de données :

```
1 >>> band.save()
```

... et ensuite regardez à nouveau l'état de l'objet :

```
1 >>> band
2 <Band: Band object (1)>
```

... maintenant l'id est `1`.

Noter que chaque objet créé possède un indice, et l'objet sera callable via cette indice.

Essayons maintenant une autre méthode. Voici une alternative en une seule ligne qui fait la même chose :

```
1 >>> band = Band.objects.create(name='Foo Fighters')
```

➤ `band=Band.objects.create(name='foo fighters')`

```
1 >>> Band.objects.count()
2 3
3 >>> Band.objects.all()
4 <QuerySet [<Band: Band object (1)>, <Band: Band object (2)>, <Band: Band object (3)>]>
```



Pour l'instant, vous pouvez considérer qu'un `QuerySet` ressemble beaucoup à une `list` Python.

➤ `Band.objects.count()`
➤ `Band.objects.all()`

- Insérer les données créée dans la BDD sur la vue hello : (bands contient la liste des objects dans la table Band, et il est possible d'accéder à l'ensemble de ces derniers via leur indice)

The screenshot shows the file structure of a Django application named 'merchex'. The 'views.py' file is selected in the sidebar. The code in 'views.py' defines a 'hello' view function that returns an HttpResponse containing an HTML template. The template uses an f-string to access the 'bands' list from the context.

```

EXPLORATEUR
...
merchex > listings > views.py > ...
    1  from django.shortcuts import render
    2  from django.http import HttpResponseRedirect
    3  from listings.models import Band
    4
    5
    6  def hello(request):
    7      bands = Band.objects.all()
    8      return HttpResponseRedirect(
    9          f"""
    10         <h1>Hello Django !</h1>
    11         <p> Mes groupes préférés sont : </p>
    12         <ul>
    13             <li>{bands[0].name}</li>
    14             <li>{bands[1].name}</li>
    15             <li>{bands[0].name}</li>
    16         </ul>
    17     """
)

```

Faire appel à un templates sur la vue :

- Créer le fichier « hello.html » dans listings/templates/listings

The screenshot shows the file structure of the 'listings' app. A new file 'hello.html' is created in the 'templates/listings' directory. The code in 'hello.html' is a simple HTML document with a title and a body section containing placeholder text for a list of groups.

```

EXPLORATEUR
...
merchex > listings > templates > listings > ...
    1  <!DOCTYPE html>
    2  <html lang="en">
    3      <head>
    4          <meta charset="UTF-8" />
    5          <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    6          <title>Document</title>
    7      </head>
    8      <body>
    9          <h1>Hello Django !</h1>
    10         <p>Mes groupes préférés sont :</p>
    11         <!-- TODO : liste des groupes -->
    12     </body>
    13 </html>
    14

```

- Dans la vue utiliser la fonction render(request, (chemin relatif de la vue), dict)

The screenshot shows the 'views.py' file again. The 'hello' view now uses the 'render' function instead of 'HttpResponse'. It passes the 'request' object, the template path 'listings/hello.html', and a context dictionary containing 'first_band' set to the first band in the 'bands' list.

```

EXPLORATEUR
...
merchex > listings > views.py > ...
    1  from django.shortcuts import render
    2  from django.http import HttpResponseRedirect
    3  from listings.models import Band, Listing
    4
    5
    6  def hello(request):
    7      bands = Band.objects.all()
    8      titles = Listing.objects.all()
    9      return render(request, "listings/hello.html", {"first_band": bands[0]})
    10
    11
    12  def about(request):
    13      return HttpResponseRedirect("<h1>À propos</h1> <p>Nous adorons merch !</p>")
    14
    15
    16  def listings(request):

```

- Dans la fonction render il est possible d'ajouter un dict en 3^{ème} argument, dans le fichier html nous pourrons faire appel à la valeur grâce à sa clé « first_band »

Ce dictionnaire est appelé **dictionnaire contextuel**. Chaque clé du dictionnaire devient une variable que nous pouvons utiliser dans notre modèle, comme ceci :

```
1 # listings/templates/listings/hello.html
2
3 ...
4 <p>Mes groupes préférés sont :</p>
5   <ul>
6     <li>{{ first_band.name }}</li>
7   </ul>
8 ...
```

html

- Il est possible de passer toute une table à la fois en context plutôt que de passer valeur par valeur

The screenshot shows a code editor with two tabs: `views.py` and `hello.html`. The `views.py` tab contains Python code for a `hello` view that renders a list of bands. The `hello.html` tab contains the corresponding HTML template with a list item for each band.

```
EXPLORATEUR
DJANGO-WEB-APP
> idea
> env
merchex
  > listings
    > __pycache__
    > migrations
    templates\listings
      hello.html
      __init__.py
      admin.py

... views.py U X hello.html U
merchex > listings > views.py > ...
1 from django.shortcuts import render
2 from django.http import HttpResponse
3 from listings.models import Band, Listing
4
5
6 def hello(request):
7     bands = Band.objects.all()
8     titles = Listing.objects.all()
9     return render(request, "listings/hello.html", {"bands": bands})
10
11
```

- Puis html il est possible de faire appel à tous les objets de list bands ainsi qu'à leur attribut

```
1 # listings/templates/listings/hello.html
2
3 ...
4 <p>Mes groupes préférés sont :</p>
5   <ul>
6     <li>{{ bands.0.name }}</li>
7     <li>{{ bands.1.name }}</li>
8     <li>{{ bands.2.name }}</li>
9   </ul>
10 ...
```



Dans le code du gabarit, pour accéder à un élément d'une liste, on utilise `bands.0`, au lieu de `bands[0]` comme on le fait dans le code Python.

- Il est possible d'utiliser une boucle FOR dans le html directement, voici l'équivalent. La boucle for permet d'éviter un bug en cas de suppression de donnée

```
<ul>
  {% for band in bands %}
    <li>{{band.name}}</li>
  {% endfor %}
</ul>
```

- Avec filtre upper

```
<ul>
    {% for band in bands %}
        <li>{{band.name|upper}}</li>
    {% endfor %}
</ul>
```

- Le filtre length permet d'afficher le nombre de valeur contenue dans le dict

```
1 # exemple de code
2
3 <p>J'ai {{ bands|length }} groupes préférés.</p>
```

- If else

```
1 <p>
2     J'ai..
3     {% if bands|length < 5 %}
4         peu de
5     {% elif bands|length < 10 %}
6         quelques
7     {% else %}
8         beaucoup de
9     {% endif %}
10    groupes préférés.
11 </p>
```

Les instructions `if` , `elif` et `else` sont également semblables à la façon dont nous les écrivons en Python, mais là encore, nous omettons les deux-points de fin (`:`).

Pour créer du code html qui ne se répète pas :

1. créer un fichier « base.html » avec une balise de gabarit django

`{% block content %}{% endblock %}` sert à insérer du code dynamiquement.

- `content` est le nom de la variable
- `block` est le type de variable et prend en argument le nom de la variable

```
1 # listings/templates/listings/base.html
2
3 <html>
4     <head><title>Merchex</title></head>
5     <body>
6
7         {% block content %}{% endblock %}
8
9     </body>
10 </html>
```

2. créer une page html uniquement avec le contenu du body et sans le reste contenu dans `base.html` (ce qui équivaut à supprimer ce qui se répète)

```
1 # listings/templates/listings/hello.html
2
3 <h1>Hello Django !</h1>
4
5 <p>Mes groupes préférés sont :</p>
6
7 <ul>
8     {% for band in bands %}
9         <li>
10            {{ band.name }}
11        </li>
12    {% endfor %}
13 </ul>
```

3. Ajouter les balises de gabarit django en haut et en bas (comme nous le ferions pour un décorateur)

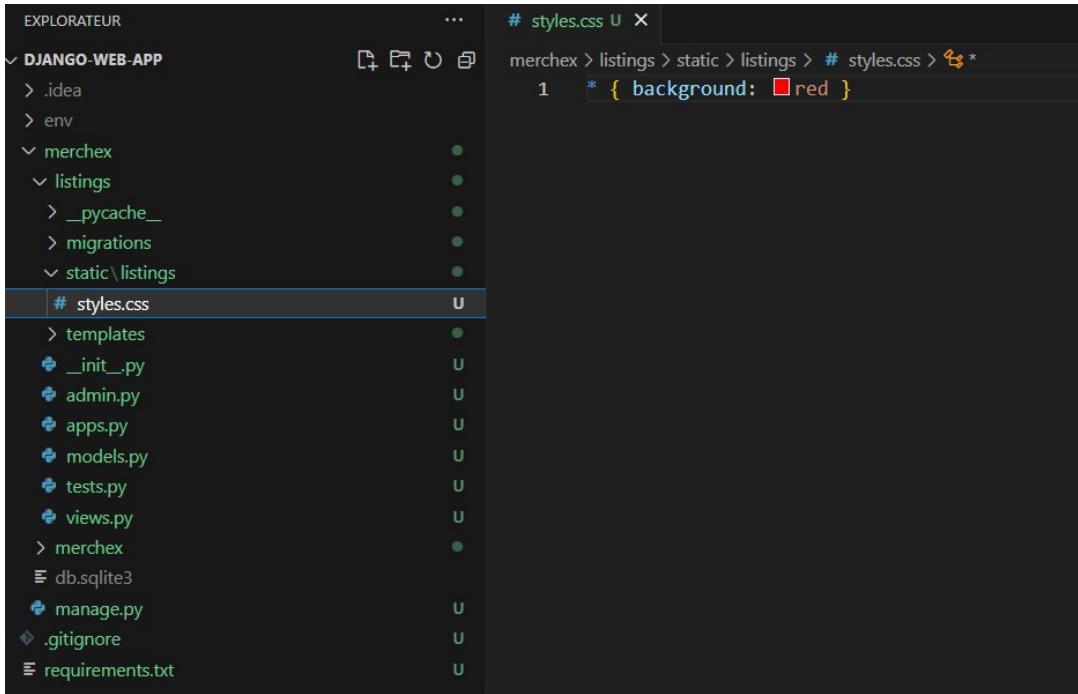
La balise de gabarit django extends indique à django que nous souhaitons importer du code html avec partir de base.html

- La balise extends prend en argument un string entre guillemets

```
1 {% extends 'listings/base.html' %} 
2
3 {% block content %}
4
5 <h1>Hello Django !</h1>
6 ...
7 </ul>
8
9 {% endblock %}
```

Les fichiers static dans django sont les fichiers qui ne peuvent pas être changé dynamiquement (.js, .css par exemple)

- Créer un dossier listings/static/listings/ à l'intérieur duquel nous placerons nos fichiers static ici « styles.css »



- Aller dans le fichier html ou l'on souhaite appliquer ce styles.css et intégrer en début de fichier css :
 - { % load static % }

```

1  { % load static %}
2
3  <html>
4
5  ...

```

- Et dans la balise link ce chemin relatif composé d'une autre balise de gabarit django { % static 'listings/styles.css' % }

```

1  <html>
2    <head>
3      <title>Merchex</title>
4      <link rel="stylesheet" href="{ % static 'listings/styles.css' %}" />

```

Site d'administration de BDD django :

1. créer un superUSER

Entrer la CLI suivante :

```
python manage.py createsuperuser
(l'adresse mail est facultative)
```

2. Mettre le fichier « admin.py » de l'application pour être en mesure de mettre à jour le modèle dans l'interface graphique de django

`admin.site.register(Band)` > permet l'enregistrement du modèle Band sur l'interface graphique d'administration django.

```
1 # listings/admin.py
2
3 from django.contrib import admin
4
5 from listings.models import Band
6
7 admin.site.register(Band)
```

3. Lancer la commande `python manage.py runserver`

4. Aller sur <http://127.0.0.1:8000/admin/>

Maintenant, voici la partie cool. Cliquez sur le lien « + Add » pour « Bands »...



Le lien « + Add ».

... et vous verrez apparaître un formulaire généré automatiquement pour ajouter un nouveau groupe à votre base de données ! Ce formulaire comporte des types de saisie appropriés pour chaque champ (comme une liste déroulante pour le champ `genre`) et inclut une validation pour garantir que les données soumises sont conformes aux contraintes que vous avez définies dans votre modèle.

Essayez maintenant de cliquer sur « Save » avec un formulaire vide, et vous verrez que cela déclenche des erreurs de validation (en anglais) pour tous les champs pour lesquels nous n'avons pas défini `blank=True` :

A screenshot of a 'Create New' form titled 'Add band'. It has a red error message box containing 'Please correct the errors below.' Below it, the 'Name:' field has a red error message 'This field is required.' The 'Genre:' dropdown menu also has a red error message 'This field is required.' at the bottom.

The 'Name:' and 'Genre:' fields both have validation errors indicated by red text and a red border around the input area.

Erreurs de validation.

Pour modifier la représentation des objets d'un modèle il suffit d'ajouter la fonction `__str__` dans le modèle

Ouvrez le fichier models.py et ajoutez :

```
1 class Band(models.Model):
2     ...
3     def __str__(self):
4         return f'{self.name}'
```

Select band to change

Action: ----- Go 0 of 4 selected

- BAND
- Band object (4)
- Band object (3)
- Band object (2)
- Band object (1)

4 bands

Action: ----- Go 0 of 4 selected

- BAND
-
- foo fighters
- Cut Copy
- De La Soul

4 bands

>>>

Nous voyons maintenant le « name » du groupe à la place de « Band object (4) »

Pour afficher plusieurs champs dans le READ sur l'admin de django (name, year, genre) :

➤ Modifier le fichier admin.py de l'application

EXPLORATEUR

- ✓ DJANGO-WEB-APP
 - > .idea
 - > env
 - ✓ merchex
 - ✓ listings
 - > __pycache__
 - > migrations
 - > static
 - > templates
 - ✓ __init__.py
 - ✓ admin.py
 - ✓ apps.py
 - ✓ models.py
 - ✓ tests.py
 - ✓ views.py
 - > merchex
 - ✗ db.sqlite3

```
models.py U      admin.py U      tests.py U
merchex > listings > admin.py > ...
1   from django.contrib import admin
2
3   from listings.models import Band
4
5
6   class BandAdmin(admin.ModelAdmin): # nous insérons ces deux lignes..
7       list_display = (
8           "name",
9           "year_formed",
10          "genre",
11      ) # liste les champs que nous voulons sur l'affichage de la liste
12
13
14  admin.site.register(
15      Band, BandAdmin
16  ) # nous modifions cette ligne, en ajoutant un deuxième argument
```

Si deux classes à importer alors écrire une fonction admin par classe

Select band to change

Action: ----- Go 0 of 4 selected

NAME	YEAR FORMED	GENRE
<input type="checkbox"/>	2000	Hip Hop
<input type="checkbox"/> foo fighters	2000	Hip Hop
<input type="checkbox"/> Cut Copy	2000	Hip Hop
<input type="checkbox"/> De La Soul	2000	Hip Hop

4 bands

Cardinalité : donne des informations sur le nombre minimum et maximum d'élément associé à chaque table de la relation

Cardinalités

- 1 à 1
- 1 à plusieurs
- Plusieurs à plusieurs

- Dans une relation 1 à 1 chaque élément de la table A ne peut être lié qu'à un seul élément de la table B

1 à 1

A
1.
2.
3.
4.

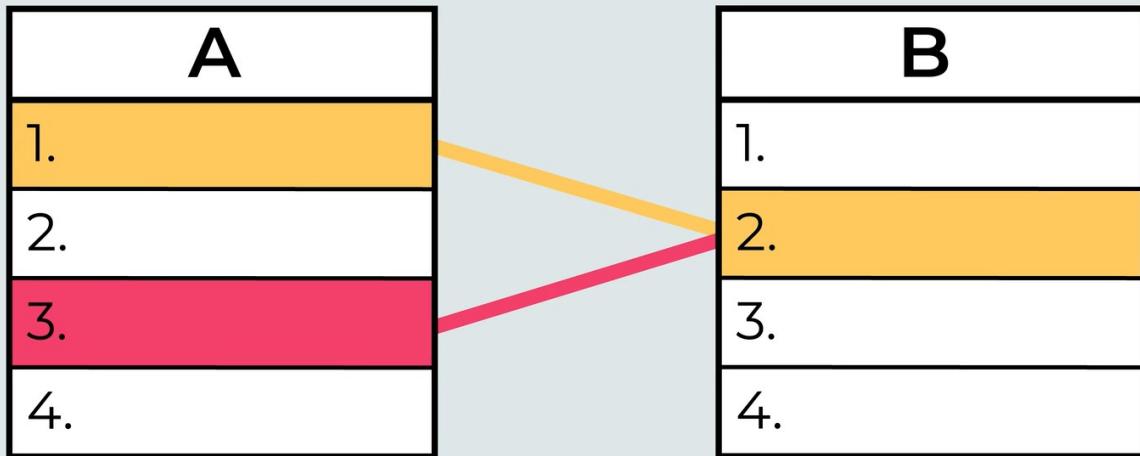
B
1.
2.
3.
4.



- Dans une relation 1 à plusieurs l'objet de la table A est lié qu'à un seul objet de la table B mais la table B peut être lié à plusieurs éléments de la table A (relation la plus courante)

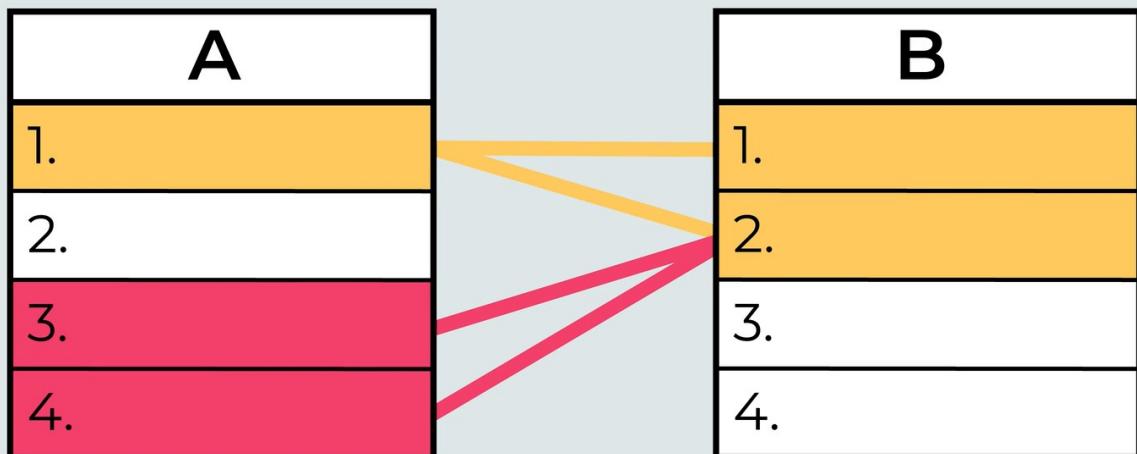
Ex B=Chanteur et A=Chanson

1 à plusieurs



- Dans une relation plusieurs à plusieurs

Plusieurs à plusieurs



Foreign key : (band_id) qui lie la table Listing à la table Band

La clé étrangère est un id faisant référence à un élément d'une autre table

Ex : band_id est une colonne de clé étrangère renvoyant vers la table band

Band		Listing	
	id		id
	name		title
	1	De La Soul	1
	2	Foo Fighters	2
	3	Cut Copy	3

Listing		
	id	title
		band_id
	1	T-shirt Cut Copy
	2	Affiche tournée De La Soul
	3	Affiche du single des Foo Fighters

Ajouter une clé étrangère à une classe modèle :

```

1 class Listing(models.Model):
2
3     ...
4     band = models.ForeignKey(Band, null=True, on_delete=models.SET_NULL)

```

Nous passons également trois arguments à `ForeignKey` :

- le modèle auquel on veut se rattacher : `Band` ;
- `null=True` : parce que nous voulons permettre la création d'annonces même si elles ne sont pas directement liées à un groupe ;
- `on_delete=models.SET_NULL` : c'est ici que nous décidons de la stratégie à suivre lorsque les objets `Band` sont supprimés. Il existe de multiples options pour cela, comme par exemple :
 - définir le champ `band` comme nul en utilisant `models.SET_NULL`,
 - définir le champ `band` à sa valeur par défaut en utilisant `models.SET_DEFAULT`,
 - supprimer l'objet `Listing` en utilisant `models.CASCADE`,
 - et d'autres paramètres plus complexes que vous pouvez trouver décrits dans la [documentation de Django](#).

Nous ne voulons pas supprimer l'objet `Listing` si un `Band` est supprimé, nous utiliserons donc `SET_NULL`.

Pour annuler une migration :

Méthode 1 (si l'application est toujours en local, non déployée)

1. Afficher la liste de toutes les migrations

```
python manage.py showmigrations
```

2. Identifier la migration qui doit être annulé (par son nom ou bien son positionnement, en dernier)

```
[X] 0002_remove_content_type_name
listings
[X] 0001_initial
[X] 0002_listing
[X] 0003_band_active_band_biography_band_genre_and_more
[X] 0004_listing_description_listing_sold_listing_type_and_more
[X] 0005_alter_listing_year_formed
[X] 0006_listing_band
[X] 0007_band_like_new X
```

3. Pour supprimer « 0007_band_like_new » récupérer le nom de la migration précédente et le nom de l'application

Migration précédente : `0006_listing_band`

Nom app : `listings`

4. Exécuter la CLI : `python manage.py migrate listings 0006_listing_band`

Si on exécute : `python manage.py showmigrations listings`

A ce stade le problème est résolu !

On peut voir que la migration `0007_band_like_new` n'est plus active

```
listings
[X] 0001_initial
[X] 0002_listing
[X] 0003_band_active_band_biography_band_genre_and_more
[X] 0004_listing_description_listing_sold_listing_type_and_more
[X] 0005_alter_listing_year_formed
[X] 0006_listing_band
[ ] 0007_band_like_new
```

5. Il est maintenant possible de supprimer la migration

```
rm listings/migrations/0006_band_like_new.py
```

Méthode 2 (si l'application est déployée)

1. Modifier le fichier « `models.py` » en appliquant les modifications souhaitées

```
1 class Band(models.Model):
2     ...
3     # like_new = models.BooleanField(default=False)    <-- SUPPRIMER CETTE LIGNE
```

2. Réaliser les migrations ! (`makemigration` et `migrate`)

```
python manage.py makemigrations
```

```
python manage.py migrate
```

La base de données est maintenant revenue à son état antérieur et si quelqu'un a exécuté la migration indésirable sur sa machine, une fois qu'il aura exécuté cette migration, sa machine le fera aussi !

Fusionner les migrations :

Si une personne A créé une migration sur une branche `FEATURE-add-hometown-to-band` et qu'ensuite il merge son travail sur le main

ET

Si une personne B créé une migration sur une branche `FEATURE-add-record-company-to-band` et qu'ensuite il merge son travail sur le main

Il sera alors nécessaire d'appliquer la CLI suivante pour fusionner les deux travail précédent pour ensuite pouvoir appliquer les migrations

Sur la main où l'on a mergé l'ensemble du travail :

```
python manage.py makemigrations --merge
```



Cette technique ne fonctionne que si les migrations n'affectent pas le même champ sur le même modèle. Si c'est le cas, la meilleure chose à faire est de supprimer les migrations en conflit et d'en créer de nouvelles à la place !

Format Nom de template :

Ex pour le model Band

- Band_list.html (Pour template global)

The screenshot shows a code editor with a dark theme. At the top, there are three tabs: 'urls.py' (marked with a green checkmark), 'band_list.html' (marked with a red X), and 'band_detail.html' (marked with a green checkmark). Below the tabs, the file path is shown as 'merchex > listings > templates > listings > band_list.html'. The code itself is a Django template:

```
1  {% extends 'listings/base.html' %}  
2  
3  {% block content %}  
4  
5  <h1>Groupes</h1>  
6  
7  <ul>  
8  |  {% for band in bands %}  
9  |  |  <a href="{% url 'band-detail' band.id %}">{{ band.name }}</a>  
10 |  |  {% endfor %}  
11 |  </ul>  
12  
13  {% endblock %}  
14
```

La balise de gabarit django url permet de générer des liens solides vers les détails :

- 'band-detail' est l'argument name présent dans url
- band.id vient de l'argument context passé à render

Ou

- band_detail.html (pour template détail)

```

merchex > listings > templates > listings > band_detail.html.html > ...
1  {% extends 'listings/base.html' %} {% block content %}
2
3  <h2>{{ band.name }}</h2>
4
5  <ul>
6      <!-- .get_genre_display permet d'afficher sur le site la clé
7          et non la valeur abrégé stocké dans la BDD en de liste de valeur --&gt;
8      &lt;li&gt;Genre : {{ band.get_genre_display }}&lt;/li&gt;
9      &lt;li&gt;Année de formation : {{ band.year_formed }}&lt;/li&gt;
10     &lt;li&gt;Actif : {{ band.active|yesno }}&lt;/li&gt;
11     {% if band.official_homepage %}
12         &lt;li&gt;&lt;a href="{{ band.official_homepage }}&gt;{{ band.official_homepage }}&lt;/a&gt;&lt;/li&gt;
13     {% endif %}
14 &lt;/ul&gt;
15
16 &lt;p&gt;{{ band.biography }}&lt;/p&gt;
17
18 &lt;a href="{% url 'band-list' %}"&gt;Retour à liste des groupes&lt;/a&gt;
19
20  [% endblock %]
</pre>

```

« .get_genre_display » permet d'afficher sur le site la clé et non la valeur abrégé stocké dans la BDD en de liste de valeur

```

1 # listings/templates/listings/band_detail.html
2
3     <li>Genre : {{ band.get_genre_display }}</li>

```

Utilisons également le filtre de gabarit yesno sur le champ active, pour convertir « True » en « Yes ». Et nous ferons de la page d'accueil officielle un lien cliquable.

Nous devons aussi nous rappeler que official_homepage peut être vide (c'est un champ nullable), donc nous devons d'abord vérifier qu'il existe avec une instruction if.

Format nom vue : (même nom que le template)

Pour vue global > liste toutes les entités par un champs comme le nom (**READ**)

```
merchex > listings > views.py > ...
1  from django.shortcuts import render
2  from django.http import HttpResponseRedirect
3  from listings.models import Band, Listing
4
5
6  def band_list(request):
7      bands = Band.objects.all()
8
9      return render(request, "listings/band_list.html", {"bands": bands})
10
```

Pour vue détail > affiche tous les champs d'une entité (**READ**)

```
def band_detail(request, band_id): # notez le paramètre id supplémentaire
    # nous insérons cette ligne pour obtenir le Band avec cet id
    band = Band.objects.get(id=band_id)
    return render(
        request, "listings/band_detail.html", {"band": band}
    ) # nous mettons à jour cette ligne pour passer le groupe au gabarit
```

Notons que sur la vue détail, nous récupérons band_id capturé dans l'entrée utilisateur.
band = Band.objects.get(id=band_id) #

Format url (modèle Band) :

```
urls.py  u x

merchex > merchex > urls.py > ...
o   1. Add an import: from my_app import views
9     2. Add a URL to urlpatterns: path('', views.home, name='home')
10 Class-based views
11   1. Add an import: from other_app.views import Home
12     2. Add a URL to urlpatterns: path('', Home.as_view(), name='home')
13 Including another URLconf
14   1. Import the include() function: from django.urls import include, path
15     2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
16 """
17 from django.contrib import admin
18 from django.urls import path
19 from listings import views
20
21 urlpatterns = [
22     path("admin/", admin.site.urls),
23     path("bands/", views.band_list, name="band-list"),
24     # ajouter ce motif sous notre autre motif de groupes
25     path("bands/<int:band_id>", views.band_detail, name="band-detail"),
26     path("about-us/", views.about),
27     path("listings/", views.listings),
28     path("contact-us/", views.contact),
29 ]
```

path("bands/", views.band_list) > pour vue globale

path("bands/<int:band_id>", views.band_detail) > pour vue détail

Notons que :

- pour la vue détail, nous capturons l'entier venant après « bands/ » pour le réutiliser dans la vue
« band_id » joue le rôle d'une variable input
- L'argument name pourra être passé en argument aux balises de gabarit django pour générer des liens solides

Create : depuis la partie utilisateur sur le site web directement (Envoyez des données du navigateur au serveur avec un formulaire)

1. Dans le répertoire de l'application : créer un fichier « forms.py » qui contiendra tous les formulaires permettant de créer une entité dans la BDD (ex ici avec le formulaire de contact)

The screenshot shows a web browser window with the title 'Merchex'. The address bar displays '127.0.0.1:8000/contact-us/'. The page content is as follows:

Groupes Liste

Contactez-nous

Nous sommes là pour vous aider

Nom :

Email :

Message :

Voici à quoi doit ressembler le template « contact-us »

Voici la class écrite dans forms.py permettant de récupérer les coordonnées d'une personne :

```
1 # listings/forms.py
2
3 from django import forms
4 class ContactUsForm(forms.Form):
5     name = forms.CharField(required=False)
6     email = forms.EmailField()
7     message = forms.CharField(max_length=1000)
```

L'argument « required=False » permet de noter qu'un champ est facultatif (nous permettons ici à l'utilisateur de ne pas entrer son nom et de rester anonyme)

2. Dans views.py de l'app, importer ContactUsForm qui contiendra les données du formulaire entré par un utilisateur et send_email, puis injecter ces données dans le template grâce à la variable de context

```
views.py U X  email.html U  # styles.css U  contact.html U  settings.py U  listing_detail.html U  listing_list.html U  model

merchex > listings > views.py > ...
1  from django.shortcuts import render
2  from django.http import HttpResponseRedirect
3  from django.core.mail import send_mail
4  from django.shortcuts import redirect
5
6  from listings.models import Band, Listing
7  from listings.forms import ContactUsForm
8
9
10 def contact(request):
11     if request.method == "POST":
12         # créer une instance de notre formulaire et le remplir avec
13         # les données POST
14         form = ContactUsForm(request.POST)
15
16         if form.is_valid():
17             """si le formulaire est valide nous envoyons un email !"""
18
19             send_mail(
20                 subject=f'Message from {form.cleaned_data["name"]} or "anonyme" via MerchEx Contact Us form',
21                 message=form.cleaned_data["message"],
22                 from_email=form.cleaned_data["email"],
23                 recipient_list=[ "admin@merchex.xyz" ],
24             )
25             """ redirige vers la page "email envoyé" pour éviter les doublon de POST """
26             return HttpResponseRedirect("email-sent")
27             # créer l'url, le template et la vue de email-sent !
28
29     # si le formulaire n'est pas valide, nous laissons l'exécution continuer jusqu'au return
30     # ci-dessous et afficher à nouveau le formulaire (avec des erreurs).
31
32 else:
33     # ceci doit être une requête GET, donc créer un formulaire vide
34     form = ContactUsForm()
35
36 return render(request, "listings/contact.html", {"form": form})
37
```

Nous importons la fonction `send_mail` de Django au début. Ensuite, nous insérons l'instruction `if` imbriquée, commençant par : `if form.is_valid():`.

Si tous les champs de notre formulaire contiennent des données valides, alors

`form.is_valid()` renvoie `True` et nous appelons `send_mail` pour envoyer notre e-mail.



`form.cleaned_data` est un `dict` contenant les données du formulaire après qu'elles ont subi le processus de validation. Lorsque nous sommes prêts à faire quelque chose avec les données de notre formulaire, nous pouvons accéder à chacun des champs via `form.cleaned_data['name_of_field']`, mais nous devons d'abord appeler `form.is_valid()`.

L'envoi d'un véritable e-mail implique la configuration d'un serveur SMTP, que nous n'avons malheureusement pas le temps de couvrir ici ! Mais nous pouvons utiliser le serveur de messagerie fictif de Django pour tester notre formulaire. Ceci affichera tous les e-mails envoyés par Django dans le terminal. Ajoutez cette ligne au tout début du fichier settings.py :

```
1 # merchex/settings.py
2
3 EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

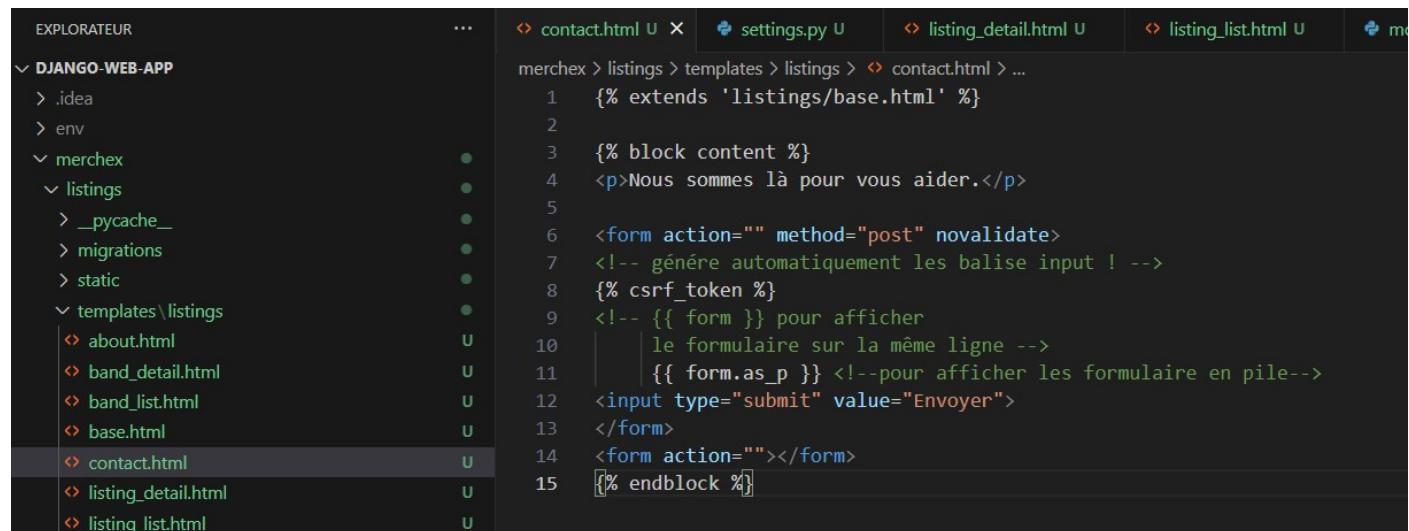
python

Pour implémenter la redirection, nous allons utiliser la fonction `redirect`.

`redirect` est un raccourci pratique. Nous pouvons lui fournir un modèle d'URL avec des arguments ou directement un chemin d'URL.

Lorsque le formulaire est valide et que le courriel a été envoyé, cette redirection guidera le navigateur de la page du formulaire vers une page de confirmation. C'est une page qui a un motif URL avec le nom `email-sent`. Bien entendu, cette page n'a pas encore été créée. Vous devriez maintenant être en mesure de créer vous-même un modèle d'URL, une vue et un gabarit pour cette page de confirmation, je vous laisse donc vous en charger !

3. Mettre à jour le template « contact.html »



```
EXPLORATEUR ... ◊ contact.html U X ⚡ settings.py U ◊ listing_detail.html U ◊ listing_list.html U ⚡ m
DJANGO-WEB-APP
> .idea
> env
merchex
  listings
    __pycache__
    migrations
    static
    templates\listings
      about.html
      band_detail.html
      band_list.html
      base.html
      contact.html
      listing_detail.html
      listing_list.html
```

```
merchex > listings > templates > listings > contact.html > ...
1   {% extends 'listings/base.html' %}
2
3   {% block content %}
4     <p>Nous sommes là pour vous aider.</p>
5
6     <form action="" method="post" novalidate>
7       <!-- génère automatiquement les balise input ! -->
8       {% csrf_token %}
9       <!-- {{ form }} pour afficher
10      | le formulaire sur la même ligne -->
11      | {{ form.as_p }} <!--pour afficher les formulaire en pile-->
12      <input type="submit" value="Envoyer">
13    </form>
14
15  {% endblock %}
```

```
<!-- génère automatiquement les balise input ! -->
{% csrf_token %}
{{ form }}
```

The screenshot shows the Chrome DevTools Elements tab. At the top, there are input fields for 'Nom', 'Email', and 'Message', followed by a 'Soumettre' button. Below this, the browser's address bar shows the URL <http://127.0.0.1:8000/contact-us/>. The main content area displays the HTML code for a contact form. The code includes a CSRF token input field (`<input type="hidden" name="csrfmiddlewaretoken" value="dqsg..."/>`), a label for the name input (`<label for="id_name">Nom :</label>`), and an input field for the name (`<input type="text" name="name" id="id_name">`). The 'Elements' tab is selected, and the status bar at the bottom says 'DevTools.'

DevTools.

Nous pouvons voir que Django a automatiquement généré un `<label>` et un `<input>` pour chacun de nos champs de formulaire : `name`, `email` et `message`. C'est génial : cela signifie que chaque fois que nous voulons ajouter un nouveau champ à ce formulaire, nous l'ajoutons simplement à la classe `ContactUsForm`, et Django s'occupera du HTML pour nous.

Une petite modification s'impose : les champs seraient plus beaux s'ils étaient affichés en pile plutôt que de gauche à droite :

```
html
1 # listings/templates/listings/contact.html
2
3 ...
4 {{ form.as_p }}
5 ...
```

- L'attribut `action` désigne « l'URL où nous allons envoyer les données du formulaire ». Si vous donnez à cet attribut la valeur d'une chaîne vide, ou si vous l'omettez complètement, il renverra à l'URL de la page où nous nous trouvons déjà, c'est-à-dire ["http://127.0.0.1:8000/contact-us/"](http://127.0.0.1:8000/contact-us/). Cela signifie que nous allons gérer les données du formulaire dans notre vue `contact`.
- La valeur de `method` est `post` : ce qui signifie que les données seront envoyées comme une requête HTTP POST. C'est un peu différent des requêtes GET que nous avons utilisées jusqu'à présent, car en plus d'une « méthode » et d'un « chemin », elle comprend également un « corps » de requête qui contient les données du formulaire.
- `novalidate` désactive la validation de formulaire de votre navigateur. Il s'agit d'une fonctionnalité utile de la plupart des navigateurs, et nous la réactiverons plus tard, mais nous devons d'abord vérifier que notre formulaire fonctionne correctement sans elle.

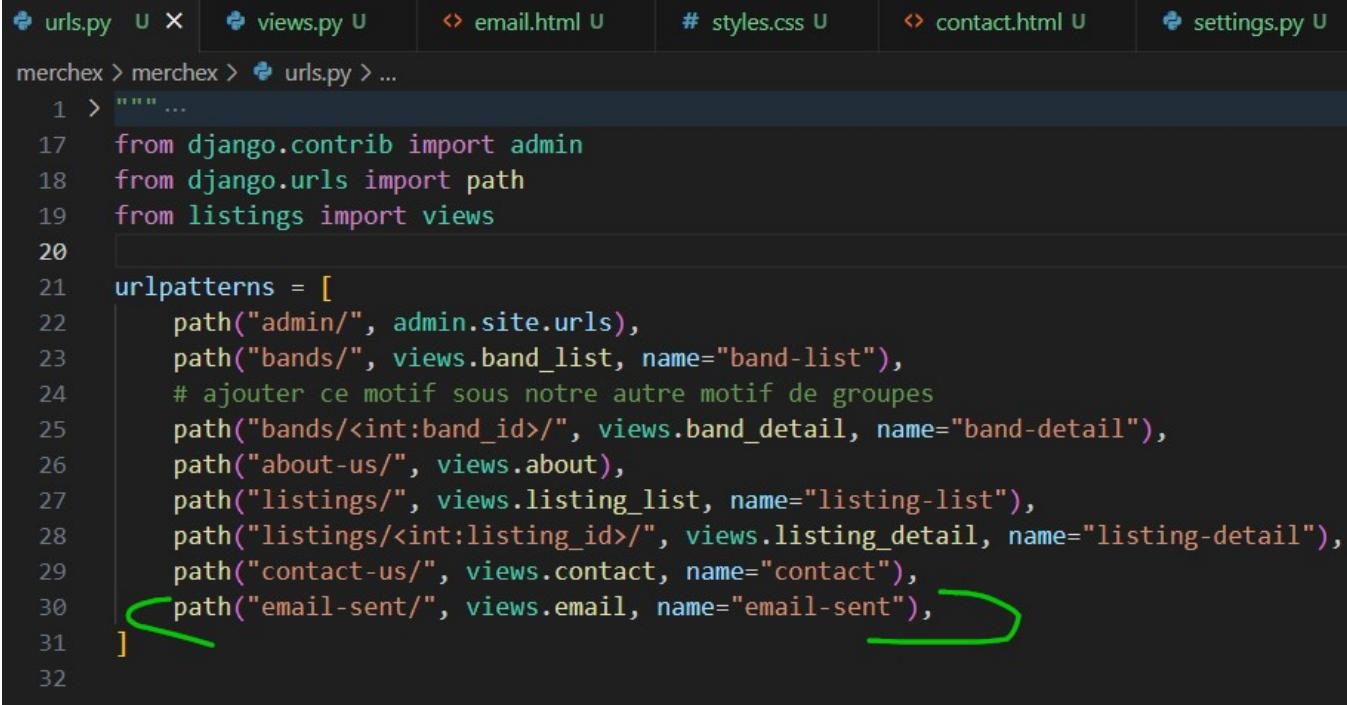
4. Ajoutons l'attribut « `name='contact'` » à l'url, pour être en mesure de call ce lien de manière solide

Ajoutons aussi cette page à notre `urlpatterns` en mettant à jour `merchex/urls.py`.

```
1 urlpatterns = [
2     ...
3     path('contact-us/', views.contact, name='contact'),
4 ]
```

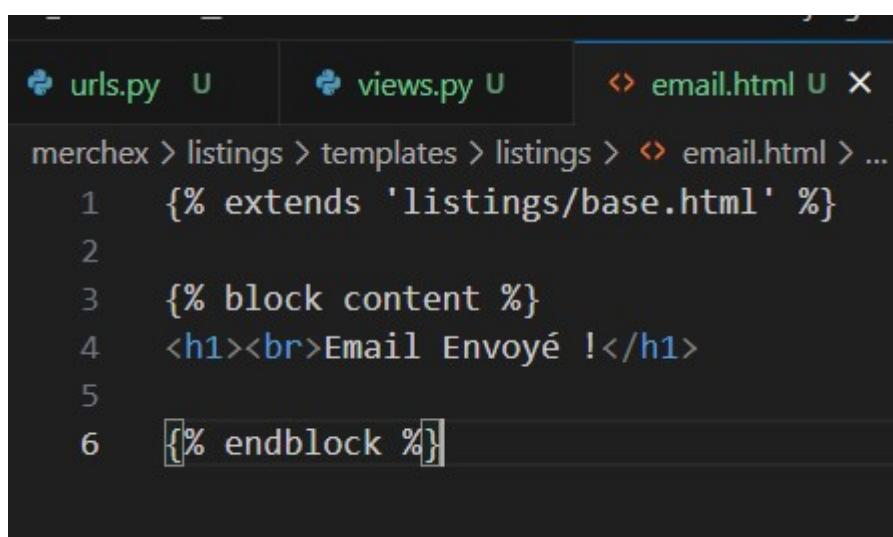
Dernière étape configurer l'url, la vue et le template de la page de redirection « email envoyé » :

➤ L'url



```
urls.py  u X views.py  u email.html  u # styles.css  u contact.html  u settings.py  u
merchex > merchex > urls.py > ...
1 > """
17 from django.contrib import admin
18 from django.urls import path
19 from listings import views
20
21 urlpatterns = [
22     path("admin/", admin.site.urls),
23     path("bands/", views.band_list, name="band-list"),
24     # ajouter ce motif sous notre autre motif de groupes
25     path("bands/<int:band_id>/", views.band_detail, name="band-detail"),
26     path("about-us/", views.about),
27     path("listings/", views.listing_list, name="listing-list"),
28     path("listings/<int:listing_id>/", views.listing_detail, name="listing-detail"),
29     path("contact-us/", views.contact, name="contact"),
30     path("email-sent/", views.email, name="email-sent"),
31 ]
32
```

➤ Le template



```
urls.py  u views.py  u email.html  u
merchex > listings > templates > listings > email.html > ...
1  {% extends 'listings/base.html' %} 
2
3  {% block content %}
4      <h1><br>Email Envoyé !</h1>
5
6  {% endblock %}
```

➤ La vue

```
def email(request):
    return render(request, "listings/email.html")
```