Créer un environnement virtuel nommé env 1
Activer l'environnement virtuel 1
Installer Django 2
Mettre les dépendances dans un fichier txt 2
check version 2
Créer un projet qui s'appelle merchex dans l'environnement virtuel: 2
Démarrer le serveur 2
de développement dans le dossier merchex dans l'env avec py manage.py runserver 2
Créer une application 3
Créer une migration 6
Appeler la migration 6
Entrer des données dans la BDD dans le shell de Django 6
Gabarit 8
CSS 12
Capture des données avec des modèles et des champs 14
CRUD (Create, Read, Update, Delete) 16
Erreur de migration et rectfication 20
Fusion de migration et résolution de conflit merge 21
Paramètre dans l'URL 22
Référencement URL 24
Form 25
Créer des objets de modèles avec ModelForm 28

Django est un MVT (Modèle-Vue-Template) :

- Modèle : stocke les données
- Vue : récupère les données
- Template : afficher les données

Vous devez toujours avoir votre environnement virtuel activé pendant le développement. Si vous redémarrez votre machine et revenez ensuite au projet, vous pouvez activer votre environnement virtuel en vous assurant d'abord que vous êtes dans le répertoire qui contient votre répertoire `env`

Créer un environnement virtuel nommé env

```
~/projects/django-web-app
→ python -m venv env
~/projects/django-web-app
→ ls
env
```

Activer l’environnement virtuel

projects/django-web-app
`env\Scripts\activate.bat`
(env) ~/projects/django-web-app

Le `(env)` au début de notre chemin nous indique que notre environnement virtuel est maintenant activé. Vous devez toujours avoir votre environnement virtuel activé pendant le développement.

Installer Django

```
(env) ~/projects/django-web-app

→ pip install django
...
Successfully installed django-3.1.7
```

Mettre les dépendances dans un fichier txt

→ pip freeze > requirements.txt

```
# ~/projects/django-web-app/requirements.txt

asgiref==3.3.1
Django==3.1.7
pytz==2021.1
sqlparse==0.4.1
```

check version

`django-admin --version`

Créer un projet qui s’appelle merchex dans l'environnement virtuel:

django-admin startproject merchex

Démarrer le serveur

de développement dans le dossier merchex dans l'env avec `py manage.py runserver`

```
(env) C:\Users\LOL\Desktop\django-web-app\merchex>py manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin,
 auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
April 27, 2024 - 09:30:44
Django version 5.0.4, using settings 'merchex.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

http://127.0.0.1:8000/

Pour arrêter le serveur : Ctrl + C

Les **migrations** représentent un moyen de configurer la base de données de notre application : python manage.py migrate

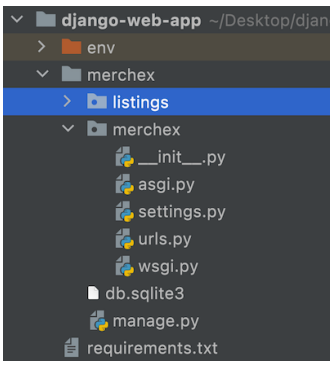
```
(env) ~/projects/django-web-app/merchex
→ python manage.py migrate
Operations to perform:
Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
Applying ...
```

```
(env) C:\Users\LOL\Desktop\django-web-app\merchex>ls
db.sqlite3  manage.py  merchex
```

Créer une application

appelé listings, qui est notre répertoire d'application avec plusieurs fichiers code de base de configuration :

```
(env) ~/projects/django-web-app/merchex
→ python manage.py startapp listings
```



Ajouter ‘listings’dans setting.py

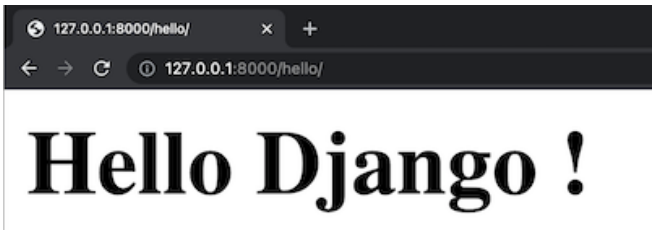
```
# ~/projects/django-web-app/merchex/merchex/settings.py

INSTALLED_APPS = [
'django.contrib.admin',
...
'django.contrib.staticfiles',

'listings',
]
```

Modifier le code dans les fichiers suisvants pour que ça ressemble à ceci :

```
# ~/projects/django-web-app/merchex/listings/views.py

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

def hello(request):
    return HttpResponse('<h1>Hello Django!</h1>')
```

```python
# ~/projects/django-web-app/merchex/merchex/urls.py

from django.contrib import admin
from django.urls import path
from listings import views

urlpatterns = [
path('admin/', admin.site.urls),
path('hello/', views.hello)
]
```

Aller sur http://127.0.0.1:8000/hello/



Pour chaque entité pour laquelle nous voulons stocker des données, nous créons un modèle pour représenter cette entité. Commençons par un modèle de groupe !

Un **modèle** définit les caractéristiques que nous voulons stocker à propos d'une entité particulière (un titre, un genre, l'année, …) Ces caractéristiques sont également connues sous le nom de **champs**.

Un modèle est un peu comme une classe en Python, avec des méthodes. un modèle est également capable de stocker (ou de "**persister**") ses données dans une base de données pour une utilisation ultérieure. Cela contraste avec les classes et objets ordinaires, dont les données existent *temporairement*

Les « caractéristiques » des classes Python sont appelées *attributs*, mais lorsqu'un modèle enregistre un attribut dans la base de données, il s'agit d'un champ.

```python
# listings/models.py

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

Nous avons défini notre classe, l'avons nommée `Band` et l'avons fait hériter de models.Model, qui est la classe de base du modèle de Django.
Ensuite, nous ajoutons un attribut de classe à notre classe `name`. À cet attribut, nous attribuons un `CharField`, qui est l'abréviation de Character Field. Il s'agira d'un

champ qui stocke des données de type caractère/texte/chaîne, ce qui est le type de données approprié pour un nom.
Nous avons également fixé la longueur maximale du nom d'un `Band` à 100.

Si nous voulons stocker les groupes dans notre base de données, nous aurons besoin d'une nouvelle table, contenant une colonne pour chaque champ que nous avons ajouté à notre modèle de groupe, ainsi qu'une colonne id pour servir de **clé primaire** : un identifiant unique pour chaque ligne de la table.

La structure d'une base de données, en termes de tables et de colonnes, est appelée **schéma**.

Si nous construisions notre schéma de base de données manuellement, nous pourrions écrire une requête SQL ou utiliser une interface graphique de gestion de base de données, pour créer notre première table.

Mais dans Django, nous faisons les choses différemment. Nous utilisons une sous-commande de l'utilitaire de ligne de commande qui va générer des instructions pour construire la table. Et ensuite, nous utilisons une autre sous-commande pour exécuter ces instructions. Ces instructions sont appelées une *migration*.

Une **migration** est un ensemble d'instructions permettant de passer le schéma de votre base de données d'un état à un autre. Il est important de noter que ces instructions peuvent être exécutées automatiquement, comme un code.

Créer une migration

Lancer migration pour inclure dans la BDD, et instruction pour créer une table

```
(env) ~/projects/django-web-app/merchex
→ python manage.py makemigrations
python manage.py makemigrations
Migrations for 'listings':
  listings/migrations/0001_initial.py
    - Create model Band
```

Appeler la migration

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

Entrer des données dans la BDD dans le shell de Django

Le **shell de Django** est simplement un shell Python ordinaire qui exécute votre application Django

```
(env) ~/projects/django-web-app/merchex

→ python manage.py shell
```

```
>>> from listings.models import Band
>>> band = Band()  # nouvelle instance du modèle Band
>>> band.name = 'De La Soul'
>>> band
<Band: Band object (None)>  ⇒ Aucun objet Band car pas d'ID
>>> band.save() #sauvegarde des données
>>> band
<Band: Band object (1)> ⇒ Id = 1, objet Band sauvegardé

>>> band = Band()
>>> band.name = 'Cut Copy'
>>> band.save()
>>> band
<Band: Band object (2)>

>>> band = Band.objects.create(name='Foo Fighters') # autre moyen d'attribuer en une seule ligne
>>> band
<Band: Band object (3)>
>>> band.name
'Foo Fighters'


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

Appuyez sur `Ctrl` + `Z + Entré` pour quitter le shell. ou exit()

```
def hello(request):
    bands = Band.objects.all() #Appelle tout le Modèle Band dans une
liste
    # utiliser des guillemets triples (""") pour répartir notre chaîne
HTML sur plusieurs lignes ;
    # fait de cette chaîne une « f-string » (f""") afin que nous
puissions injecter nos noms de groupes dans la chaîne en utilisant{ ...
}comme placeholders.
    return HttpResponse(f"""
        <h1>Hello Django !</h1>
        <p>Mes groupes préférés sont :<p>
```

```
        <ul>
            <li>{bands[0].name}</li>
            <li>{bands[1].name}</li>
            <li>{bands[2].name}</li>
        </ul>
""")
```

En programmation, un **design pattern** (patron de conception) est un style ou une technique qui représente la meilleure pratique et conduit à de bons résultats. À l'inverse, un **anti-pattern** représente une mauvaise pratique, quelque chose que nous devons éviter.

Gabarit

On a un anti-pattern ci-dessous. La vue commence déjà à avoir l'air un peu chargée et la quantité de HTML ici ne fera que s'accroître au fur et à mesure de la construction de notre application

```
def hello(request):
    bands = Band.objects.all()
    return HttpResponse(f"""
        <html>
            <head><title>Merchex</title><head>
            <body>
                <h1>Hello Django !</h1>
                <p>Mes groupes préférés sont :<p>
                <ul>
                    <li>{bands[0].name}</li>
                    <li>{bands[1].name}</li>
                    <li>{bands[2].name}</li>
                </ul>
            </body>
        </html>
    """)
```

Pour alléger, le code, on utilise des gabarits (templates)
Créer le fichier html dans les sous-répertoires de listings. Nous mettons toujours un sous-répertoire dans le répertoire des gabarits (templates) qui porte le même nom que l'application (« listings »).

```
# listings/templates/listings/hello.html

<html>
    <head><title>Merchex</title></head>
    <body>
        <h1>Hello Django !</h1>
        <p>Mes groupes préférés sont :</p>
        <!-- TODO : liste des groupes -->
    </body>
</html>
```

générer le gabarit :

```
# listings/views.py

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

def hello(request):
    bands = Band.objects.all()
    return render(request, 'listings/hello.html')
```

Chaque fois que vous voyez des doubles accolades contenant un nom de variable, la valeur de cette variable sera insérée. Elles sont appelées **variables de gabarits**.

```
# listings/views.py

...
return render(request, 'listings/hello.html',
        {'first_band': bands[0]})
```

Revenons à notre vue et ajoutons un troisième argument à notre appel de la méthode `render`. Cet argument doit être un `dict` Python. Ce dictionnaire est appelé **dictionnaire contextuel**. Chaque clé du dictionnaire devient une variable que nous pouvons utiliser dans notre modèle

```
# listings/templates/listings/hello.html

...
<p>Mes groupes préférés sont :</p>
    <ul>
        <li>{{ first_band.name }}</li>
    </ul>
...
```


```python
# merchex/listings/views.py

    ...
    return render(request,
    'bands/hello.html',
    {'bands': bands})
```

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.

```
# listings/templates/listings/hello.html

...
<p>Mes groupes préférés sont :</p>
    <ul>
        <li>{{ bands.0.name }}</li>
        <li>{{ bands.1.name }}</li>
        <li>{{ bands.2.name }}</li>
    </ul>
```

Avec des boucles for :

```
# listings/templates/listings/hello.html

<p>Mes groupes préférés sont :</p>
<ul>
    {% for band in bands %}
    <li>{{ band.name }}</li>
    {% endfor %}
</ul>
```

On peut aussi mettre en majuscule ou minuscule avec “lower” et “upper”, en appliquant un filtre grâce à la barre verticale |

```
<li>{{ band.name|upper }}</li>
```

Pour afficher la taille :

```
<p>J'ai {{ bands|length }} groupes préférés.</p>
```

condition dans un gabarit :

```
<p>
            J'ai
```

```
            {% if bands|length < 5 %}
                peu de
            {% elif bands|length < 10 %}
                quelques
            {% else %}
                beaucoup de
            {% endif %}
                groupes préférés.
        </p>
```

Hello Django !

J'ai 3 groupes préférés.

J'ai peu de groupes préférés.

Mes groupes préférés sont :

- DE LA SOUL
- CUT COPY
- FOO FIGHTERS

Pour éviter que le <html>, <head>, <title>, <body> revient trop souvent, nous allons écrire dans un nouveau fichier gabarit à l'adresse « listings/templates/listings/base.html »

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

Le contenu de la balise block, s'appelle content

on enlève les balises html de "hello.html", afin que ça ressemble à ça

```
{% extends 'listings/base.html' %}

{% block content %}

<h1>Hello Django !</h1>
...
</ul>

{% endblock %}
```

La balise de gabarits `extends` en haut indique à Django que nous voulons que ce gabarit **hérite** de notre gabarit de base.
Django va prendre tout ce qui se trouve dans le bloc nommé `content` dans notre gabarit de page et l'injecter dans le bloc du même nom dans notre gabarit de base.

CSS

Nous plaçons les fichiers statiques à un endroit spécifique de notre application. Créez un dossier dans listings/static/listings/ et à l'intérieur, créez un nouveau fichier appelé `styles.css` .

```
* { background: red }
```

dans « listings/templates/listings/base.html », ajoutons une balise `<link>` , sous le titre, pour charger notre feuille de style. Maintenant, pour que la balise `static` fonctionne, nous devons d'abord la « charger » dans ce modèle. Nous le faisons en ajoutant une balise `load` au tout début du fichier, comme ceci :

```
{% load static %}

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

On devrait avoir ça :

Si le css ne s'applique pas, il faut arrêter le serveur et le redémarrer

Django est un framework monolithique. Il fournit des moteurs pour toutes les parties de l'application : le moteur de gabarits pour la présentation, l'ORM pour la persistance, etc. Et si vous vouliez changer une de ces parties ? Il est peut-être préférable de construire votre architecture MVC/T à partir des éléments de votre choix, comme Flask, SQLAlchemy, etc.

Deuxièmement, MVC/T n'est qu'un style d'architecture parmi d'autres. Il est très adapté aux applications CRUD simples. Mais pour les solutions d'entreprise, vous pouvez étudier des alternatives comme Clean Architecture.

Pour l'instant, comprenez simplement que MVC a ses limites, et vous pourriez un jour constater que vous atteignez l'une de ces limites !

Capture des données avec des modèles et des champs

```python
from django.db import models
from django.core.validators import MaxValueValidator, MinValueValidator

class Band(models.Model):
    class Genre(models.TextChoices):
        HIP_HOP = 'HH'
        SYNTH_POP = 'SP'
        ALTERNATIVE_ROCK = 'AR'

    name = models.fields.CharField(max_length=100)
    genre = models.fields.CharField(choices=Genre.choices,max_length=5)
    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)
```

- `genre` et `biography`: tout comme `name` , ces champs contiendront des données de type caractère ou chaîne de caractères, nous utiliserons donc `CharField`. l'option `max_length` est obligatoire
- `year_formed`: une année est essentiellement un nombre entier, donc j'ai opté pour un `IntegerField` ici. Django a aussi un `DateField` , mais il comprendrait aussi un mois et un jour, qui ne seraient pas utilisés dans ce cas.
- `active`: il s'agit d'une réponse « oui » ou « non », donc un `BooleanField` avec des valeurs `True` ou `False` est parfait ici.
- `official_homepage` : pour la page officielle, nous aurions pu utiliser un autre `CharField` ici, mais Django a une meilleure proposition : `URLField` , qui n'autorisera que les URL valides dans ce champ ; nous verrons comment cela fonctionne plus tard.

l'option `null=True` pour autoriser les valeurs NULL.
lorsque nous créerons un formulaire pour créer ou modifier des objets `Band` , le fait de définir `blank=True` ici nous permettra de soumettre ce formulaire avec une zone de texte vide pour ce champ

Nous avons créé la classe `Genre`, définissant les choix qui peuvent être utilisés pour le champ `genre`. `Genre` est une **classe imbriquée** : c'est une classe définie dans une autre classe. le champ `genre` soit limité à une liste de choix que nous spécifions

Pour enregistrer les nouveaux attributs dans la table Band, ne pas oublier de faire dans le shell de django :

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

bands		
nom	**genre**	**annee_de_creation**
De La Soul	Hip hop	1988
Cut Copy	Synth-pop	200
Foo Fighters	Rock alternatif	1

Si on ne fait pas la migration, on aura :

```
(env) ~/projects/django-web-app/merchex
→ python manage.py shell
>>> from listings.models import Band
>>> band = Band()
>>> band.save()
...
django.db.utils.OperationalError: table listings_band has no column named genre
```

Mais même, après avoir fait `python manage.py makemigrations` , on a encore des problèmes :

```
(env) ~/projects/django-web-app/merchex
→ python manage.py makemigrations
You are trying to add a non-nullable field 'biography' to band without a default; we can't do that
(the database needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
 2) Quit, and let me add a default in models.py
Select an option: 1
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do e.g. timezone.now
Type 'exit' to exit this prompt
>>> ''
```

Les nouvelles colonnes ont besoin que des valeurs soient affectés. Le shel nous propose alors de définir ces valeurs

id	name	biography	genre	year_formed	active	official_homepage
1	De La Soul	?	?	?	True	NULL

2	Cut Copy	?	?	?	True	NULL
3	Foo Fighters	?	?	?	True	NULL

Une fois toutes les valeurs rentrées, on peut exécuter python manage.py migrate

```
(env) ~/projects/django-web-app/merchex
→ python manage.py shell
>>> from listings.models import Band
>>> band = Band()
>>> band.save()
...
django.db.utils.IntegrityError: NOT NULL constraint failed: listings_band.year_formed
```

On a une `IntegrityError` sur le champ `year_formed` . C'est une bonne chose ! Nous n'avons pas spécifié `null=True` sur ce champ, donc il y a une erreur sur les valeurs NULL.

Nous pouvons observer que `year_formed` est actuellement défini à `None`

```
>>> band.delete()
(1, {'listings.Band': 1})
```

CRUD (Create, Read, Update, Delete)

Avec Django, on peut gérer un de nos modèles dans le site d'administration. Pour cela, il faut créer un superuser

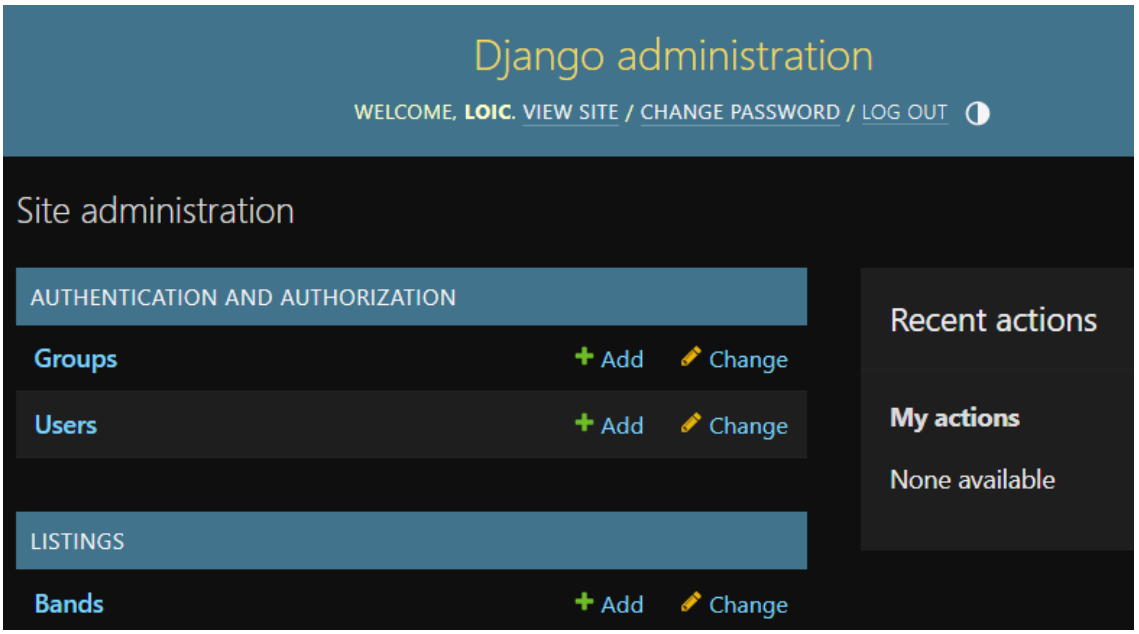
```
(env) C:\Users\LOL\Desktop\django-web-app\merchex>python manage.py createsuperuser
Username (leave blank to use 'lol'): loic
Email address: cheongloic@gmail.com
```

pwd : azerty01

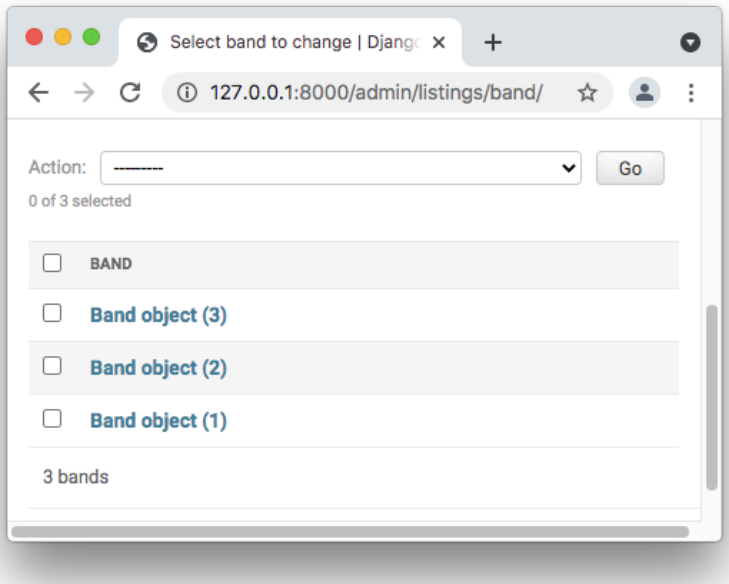
Modifier le fichier admin.py pour que ça ressemble à ça :

```python
from django.contrib import admin

# Register your models here.

from listings.models import Band

admin.site.register(Band)
```

Aller sur http://127.0.0.1:8000/admin/ , et rentrer les identifiants



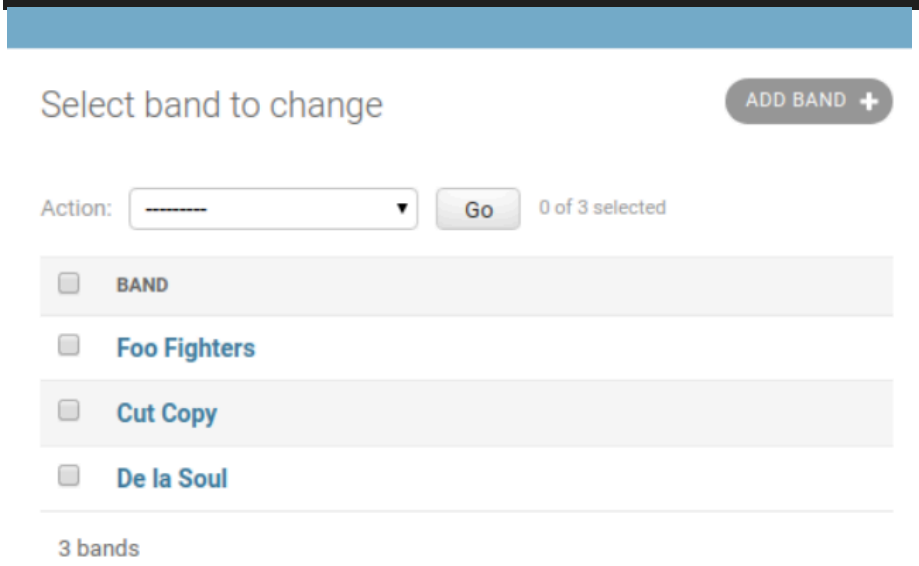
http://127.0.0.1:8000/admin/listings/band/



Ne serait-il pas préférable qu'au lieu d'afficher « Band object (id number) », nous puissions afficher quelque chose de plus significatif ? Pourquoi pas le nom du

groupe ? Pour ce faire, nous pouvons éditer la représentation de la chaîne de caractères du modèle `Band` en modifiant sa méthode intégrée`__str__`.

```
class Band(models.Model):
    class Genre(models.TextChoices):
        …
    def __str__(self):
        return f'{self.name}'
```



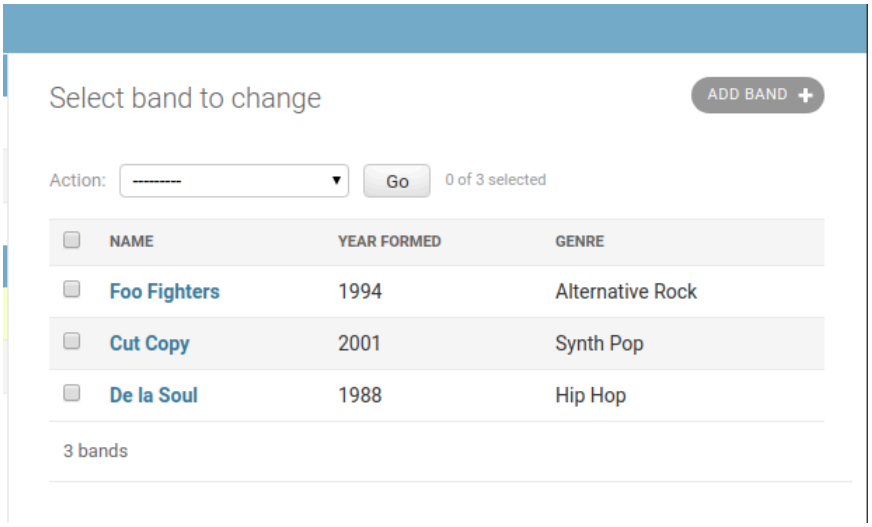
En changeant encore admin.py :

```
from django.contrib import admin
from listings.models import Band

class BandAdmin(admin.ModelAdmin):  # nous insérons ces deux lignes..
    list_display = ('name', 'year_formed', 'genre') # liste les champs
que nous voulons sur l'affichage de la liste

admin.site.register(Band, BandAdmin) # nous modifions cette ligne, en
ajoutant un deuxième argument
```

La classe appelée `BandAdmin`, héritant de `admin.ModelAdmin`. Les classes ModelAdmin permettent de configurer la manière dont les objets du modèle sont affichés dans l'administration

un attribut de classe appelé `list_display` et nous l'avons défini comme le n-uplet (tuple) `('name', 'year_formed', 'genre')`. Cela signifie que nous pouvons voir tous ces champs lorsque nous visualisons les groupes dans l'administration.

On obtient ça




```
class Listing(models.Model):
    class ListingType(models.TextChoices):
        RECORDS = 'R'
        CLOTHING = 'C'
        POSTERS = 'P'
        MISC = 'M'

    title = models.fields.CharField(max_length=100)
    description = models.fields.CharField(max_length=1000)
    sold = models.fields.BooleanField(default=False)
    year_formed = models.fields.IntegerField(
        null=True,
        validators=[MinValueValidator(1900), MaxValueValidator(2021)]
    )
    type = models.fields.CharField(choices=ListingType.choices,
max_length=5)
    band = models.ForeignKey(Band, null=True,
on_delete=models.SET_NULL)
```

La table Listing a une clé étrangère du modèle Band . *Une `ForeignKey` est utilisée pour lier les modèles entre eux dans une relation many-to-one.*

- `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`, Listing ne sera pas supprimé si Band est supprimé

- 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.

Erreur de migration et rectfication

Nous voulons ajouter un champ `like_new` au modèle `Listing` mais « Zut ! », nous l'avons ajouté au modèle `Band` à la place et nous avons exécuté notre migration. Tout ça en local, les migrations n'ont pas encore été partagées sur git

```
class Band(models.Model):
  ...
  like_new = models.fields.BooleanField(default=False)
```

Nous avons fait la migration py manage.py makemigraitons
Pour lister toutes les migrations : `python manage.py showmigrations`

```
(env) ~/projects/django-web-app/merchex
→ python manage.py showmigrations
admin
[X] 0001_initial
[X] 0002_logentry_remove_auto_add
[X] 0003_logentry_add_action_flag_choices
auth
[X] 0001_initial
[X] 0002_alter_permission_name_max_length
[X] 0003_alter_user_email_max_length
[X] 0004_alter_user_username_opts
[X] 0005_alter_user_last_login_null
[X] 0006_require_contenttypes_0002
[X] 0007_alter_validators_add_error_messages
[X] 0008_alter_user_username_max_length
[X] 0009_alter_user_last_name_max_length
[X] 0010_alter_group_name_max_length
[X] 0011_update_proxy_permissions
[X] 0012_alter_user_first_name_max_length
contenttypes
[X] 0001_initial
[X] 0002_remove_content_type_name
listings
[X] 0001_initial
[X] 0002_listing
[X] 0003_auto_20210329_2350
[X] 0004_auto_20210524_2030
[X] 0005_listing_band ← Il s'agit de la migration précédente
[X] 0006_band_like_new ← C'est notre migration indésirable
sessions
[X] 0001_initial
```

Identifiez ensuite la migration qui doit être annulée, dans notre cas il s'agit de `0006_band_like_new`. Ensuite, récupérez le nom de la migration précédente vers celle-ci : `0005_listing_band`. Vous devrez également noter le nom de l'application, qui est ici `listings`

Pour changer la migration :
`(env) ~/projects/django-web-app/merchex`

```
→ python manage.py migrate listings 0005_listing_band
Operations to perform:
 Target specific migration: 0005_listing_band, from listings
Running migrations:
 Rendering model states... DONE
 Unapplying listings.0006_band_like_new... OK
```

Vérifier de nouveau la liste des migrations

```
→ python manage.py showmigrations listings  # ne liste que les migrations de l'application listings
listings
[X] 0001_initial
[X] 0002_listing
[X] 0003_auto_20210329_2350
[X] 0004_auto_20210524_2030
[X] 0005_listing_band
[ ] 0006_band_like_new
```

La migration `listings.0006_band_like_new` n'est plus appliquée

Pour supprimer la migration définitivement :

```
(env) ~/projects/django-web-app/merchex
→ rm listings/migrations/0006_band_like_new.py  # nous pouvons ainsi la supprimer, mais ne le faisons pas pour le moment !
```

Si les modifications non souhaitées ont été appliquées sur une autre machine, vous devrez créer une nouvelle migration pour annuler les modifications.
Il suffit de changer le fichier models.py, puis faire makemigrations et migrate

Fusion de migration et résolution de conflit merge

Nous ajoutons deux champs `hometown` et `record_company`, à `Band`.Chaque champ a été ajouté séparément sur deux branches différentes, et les migrations ont été effectués sur ces branches
Les branches ont été fusionnés au branche master, on regarde la liste des migrations, nous avons à présent deux migrations préfixées avec `0006`

```
(env) ~/projects/django-web-app/merchex (master)
→ pm showmigrations listings
listings
 [X] 0001_initial
 [X] 0002_listing
 [X] 0003_auto_20210329_2350
 [X] 0004_auto_20210524_2030
 [X] 0005_listing_band
 [ ] 0006_band_hometown
```

[ ] 0006_band_record_company

si nous essayons de les migrer, nous obtenons un message d'erreur.

```
(env) ~/projects/django-web-app/merchex (master)
→ python manage.py migrate
CommandError: Conflicting migrations detected; multiple leaf nodes in the migration graph:
)6_band_record_company, 0006_band_hometown in listings).
To fix them run 'python manage.py makemigrations --merge'
```

nous pouvons fusionner ces migrations pour qu'elles fonctionnent correctement, en utilisant le flag `--merge` avec `makemigrations`

```
→ python manage.py makemigrations --merge
Merging listings
 Branch 0006_band_hometown
   - Add field hometown to band
 Branch 0006_band_record_company
   - Add field record_company to band
Merging will only work if the operations printed above do not conflict
with each other (working on different fields or models)
Do you want to merge these migration branches? [y/N] y
Created new merge migration ~/projects/django-web-app/
merchex/listings/migrations/0007_merge_20210701_1911.py
```

Après migration :

```
(env) ~/projects/django-web-app/merchex (master)
→ python manage.py migrate
Operations to perform:
Apply all migrations: admin, auth, contenttypes, listings, sessions
Running migrations:
Applying listings.0006_band_record_company... OK
Applying listings.0006_band_hometown... OK
Applying listings.0007_merge_20210701_1911... OK
```

/!\ 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 !

Paramètre dans l'URL

views.py

```
def band_detail(request, id):  # notez le paramètre id supplémentaire
    return render(request,
                  'listings/band_detail.html',
                  {'id': id}) # nous passons l'id au modèle
```

urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('bands/', views.band_list),
    path('bands/<int:id>/', views.band_detail),
    path('about-us/', views.about),  # ajoutez cette ligne
```

```
]
```

band_detail.html

```
{% extends 'listings/base.html' %}

{% block content %}
 <h2>L'identifiant est {{ id }}</h2>
{% endblock %}
```

L'id est 2

Maintenant changeons band_detail dans views.py

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

Changeons band_detail.html

```
{% extends 'listings/base.html' %}
{% block content %}
<h2>{{ band.name }}</h2>
<ul>
    <li>Genre : {{ band.genre }}</li>
    <li>Année de formation : {{ band.year_formed }}</li>
    <li>Actif : {{ band.active }}</li>
    <li>{{ band.official_homepage }}</li>
</ul>
 <p>{{ band.biography }}</p>
{% endblock %}
```

De La Soul

- Genre : HH
- Année de création : 1988
- Actif : True
- https://wearedelasoul.com/
- De La Soul est un groupe de hip-hop américain, originaire de Long Island dans l'État de New York. Ils sont connus pour l'influence qu'ils ont eu sur l'essor du jazz rap grâce à leurs samples aussi étranges que divers et à leurs textes parfois surréalistes.

Pour remplacer le 'HH' (qui est une clé) du genre par Hip Hop, convertir True en Yes
La page HTML peut être null donc on met une condition if

```
<li>Actif : {{ band.active|yesno }}</li>
    {% if band.official_homepage %}
    <li><a href="{{ band.official_homepage }}">{{
band.official_homepage }}</a></li>
    {% endif %}
```

Si on rentre un id qui n'est pas dans la base de données, la page renvoie une erreur DosNotExist

Référencement URL

Dans urls.py

```
path('bands/<int:id>/', views.band_detail, name='band-detail')
```

Dans band_list.html

```
<li><a href="{% url 'band-detail' band.id %}">{{ band.name }}</a></li>
```

Au lieu de faire qui est un anti-pattern

```
<li><a href="/bands/{{ band.id }}/">{{ band.name }}</a></li>
```

Le lien de référencement href de band_list.html va voir ce qu'il y a dans urls.py "band-detail" est le nom de la vue. Pour créer des liens dans nos gabarits, plutôt que de répéter les chemins URL, nous donnons un `name` à notre modèle d'URL, puis nous utilisons la balise de gabarits `{% url %}` pour générer le lien.

Pour parcourir en boucle les annonces d'un groupe dans le gabarit, vous pouvez utiliser `{% for listing in band.listing_set.all %}`.

Form

listings/forms.py

```
from django import forms
class ContactUsForm(forms.Form):
   name = forms.CharField(required=False) # require=False ==> champ
facultatif
   email = forms.EmailField()
   message = forms.CharField(max_length=1000)
```

views.py

```
from django.core.mail import send_mail
from django.shortcuts import redirect  # ajoutez cet import
...
def contact(request):
  # ajoutez ces instructions d'impression afin que nous puissions jeter
un coup d'oeil à « request.method » et à « request.POST »
  print('La méthode de requête est : ', request.method)
  print('Les données POST sont : ', request.POST)
  if request.method == 'POST':
      form = ContactUsForm(request.POST)  # ajout d'un nouveau
formulaire ici
      if form.is_valid():
            send_mail(
            subject=f'Message from {form.cleaned_data["name"] or
"anonyme"} via MerchEx Contact Us form',
            message=form.cleaned_data['message'],
            from_email=form.cleaned_data['email'],
            recipient_list=['admin@merchex.xyz'],
        )
      return redirect('email-sent')
    # si le formulaire n'est pas valide, nous laissons l'exécution
continuer jusqu'au return
    # ci-dessous et afficher à nouveau le formulaire (avec des
erreurs).

  else:
      form = ContactUsForm()

  return render(request,
          'listings/contact.html',
          {'form': form})  # passe ce formulaire au gabarit
```

si on ne remplit pas tous les champs du formulaire, grâce à if request.method == 'POST', le formulaire indiquera que les champs vides doivent être rempli
SI les champs sont valides, alors un email est envoyé

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()

listings/templates/contact.html

```
<p>Nous sommes là pour vous aider.</p>
<form action="" method="post" novalidate>
{% csrf_token %}
<!-- {{ form }} -->  pas besoin d'<input>
{{ form.as_p }} # chaque paire balise de champ dans une balise <p>
<input type="submit" value="Envoyer">
</form>
```

`novalidate` désactive la validation de formulaire de votre navigateur.

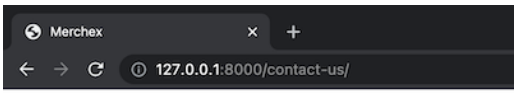
urls.py

```
path('contact-us/', views.contact, name='contact'),
```

Après rempli et soumis le formulaire, dans le terminal, le print de la fonction contact dans views.py renvoie

```
La méthode de requête est :  POST
Les données POST sont :  <QueryDict: {'csrfmiddlewaretoken': ['pBbKL4ZicAd7cZ6OFzU9dLcAHYHH1qhTReHWpJ5hyAqF3
ZQwbb1XUapvgFt1rrb1'], 'name': ['loic'], 'email': ['cheongloic@gmail.com'], 'message': ['blabla']}>
[29/Apr/2024 18:12:47] "POST /contact-us/ HTTP/1.1" 200 1336
```

`request.POST` est un `QueryDict` (qui est un type spécial de `dict` Python).



Contactez-nous

Nous sommes là pour vous aider

Nom :

- Saisissez une adresse de courriel valide.

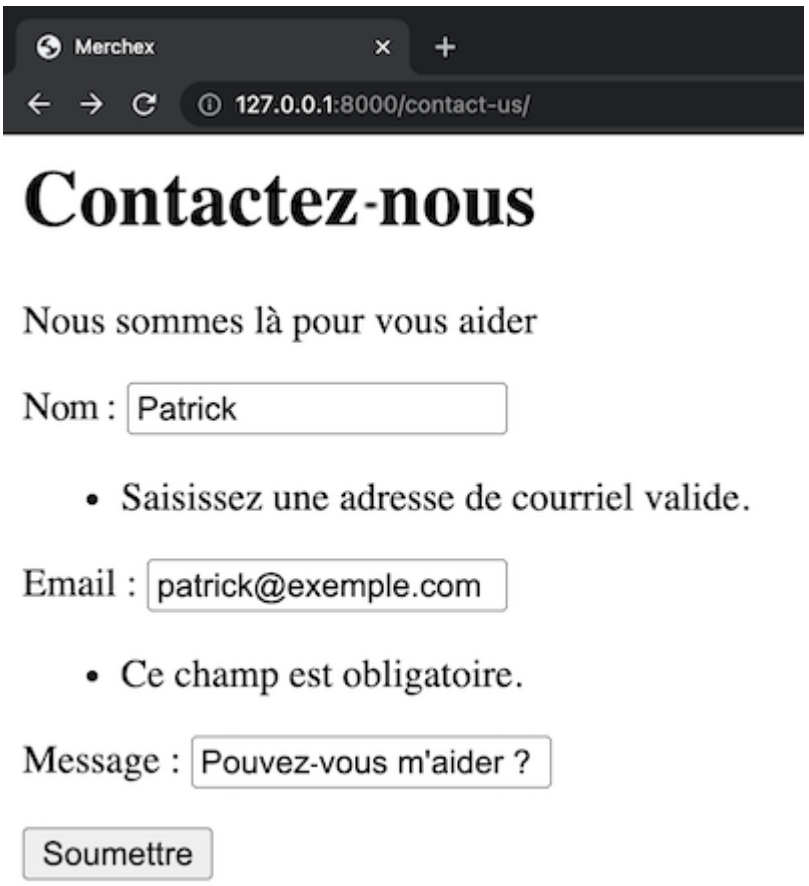
Email : pas-un-email-valide

- Ce champ est obligatoire.

Message :

Soumettre

# merchex/settings.py
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'




```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Message from Patrick via MerchEx Contact Us form
From: patrick@exemple.com
To: admin@merchex.xyz
Date: Tue, 05 Oct 2021 09:28:31 -0000
Message-ID:
 <163342611173.38095.16349944629161832729@48.1.168.192.in-addr.arpa>

Vous pouvez m'aider ?
-------------------------------------------------------------------------------
[05/Oct/2021 09:28:31] "POST /contact-us/ HTTP/1.1" 302 0
[05/Oct/2021 09:28:31] "GET /email-sent/ HTTP/1.1" 200 393
```

Créer des objets de modèles avec ModelForm

Nous voulons créer un formulaire pour rajouter un objet d'un model, pour éviter de faire un modèle redondant ressemblant au modèle de l'objet à ajouté, on utilise ModelForm

```
class BandForm(forms.Form):
   name = forms.CharField(max_length=100)
   biography = forms.CharField(max_length=1000)
   year_formed = forms.IntegerField(min_value=1900, max_value=2021)
   official_homepage = forms.URLField(required=False)
```

```
from django import forms
from listings.models import Band
...
class BandForm(forms.ModelForm):
   class Meta:
     model = Band
     fields = '__all__'
```

`Meta` spécifie le modèle pour lequel ce formulaire sera utilisé, et les champs de ce modèle à inclure dans ce formulaire (dans ce cas, tous)

views.py

```
from listings.forms import ContactUsForm, BandForm
...
def band_create(request):
   if request.method == 'POST':
        form = BandForm(request.POST)
        if form.is_valid():
            # créer une nouvelle « Band » et la sauvegarder dans la db
            band = form.save()
            # redirige vers la page de détail du groupe que nous venons
de créer
            # nous pouvons fournir les arguments du motif url comme
arguments à la fonction de redirection
            return redirect('band-detail', band.id)
    else:
        form = BandForm()
   return render(request,
            'listings/band_create.html',
            {'form': form})
```

urls.py

```
path('bands/add/', views.band_create, name='band-create'),
```

```
{% extends 'listings/base.html' %}
{% block content %}
```

```
 <h1>Créer un nouveau groupe</h1>
<form action="" method="post" novalidate>
  {% csrf_token %}
  {{ form.as_p }}
  <input type="submit" value="Envoyer">
</form>
{% endblock %}
```

Si nou voulons exclure certains champs du modèle dans le formulaire :

```
class BandForm(forms.ModelForm):
    class Meta:
        model = Band
        # fields = '__all__' # supprimez cette ligne
        exclude = ('active', 'official_homepage')  # ajoutez cette
ligne
```

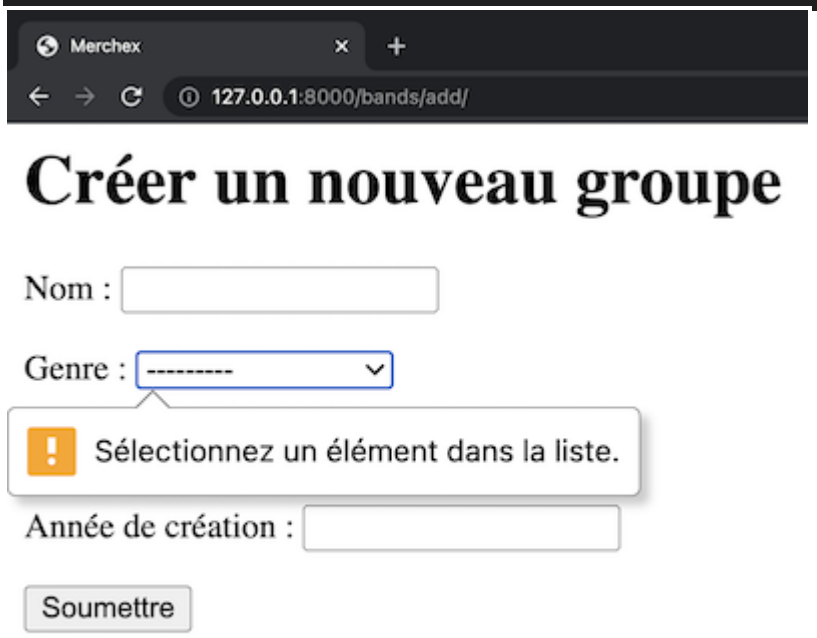
Le formulaire fonctionnera toujours et nous pourrons créer un nouvel objet `Band` sans aucune erreur, même si nous avons laissé ces champs de côté, parce que `active` a une valeur par défaut de `True` et `official_homepage` autorise les valeurs NULL avec `null=True`.

Si vous voulez exclure certains des autres champs (non nullables) du formulaire, vous devrez d'abord leur donner une valeur par défaut, ou les rendre nullables, dans listings/models.py. Vous devrez également effectuer et exécuter une migration pour mettre à jour le schéma de la base de données, car les valeurs par défaut et les valeurs NULL sont configurées dans la base de données elle-même.

```
<form action="" method="post" novalidate>
```

novalidate a désactivé la validation côté client dans votre navigateur. Réactivons-la en enlevant `novalidate` du gabarit band_create

```
<form action="" method="post">
```



Bonne pratique de la cybersécurité de Django :
https://docs.djangoproject.com/fr/3.2/topics/security/
Toujours valider les données côté serveur et non côté client

Puisque nous parlons de sécurité, c'est le moment d'expliquer ce qu'est le `{% csrf_token %}` dans tous nos formulaires. CSRF est l'abréviation de Cross Site Request Forgery. Il s'agit d'une méthode visant à inciter les utilisateurs à effectuer des opérations en créant (ou en « forgeant ») des requêtes. Le `csrf_token` empêche cela, en générant un jeton (token) aléatoire pour chaque requête. Si une requête POST ultérieure n'inclut pas ce jeton exact, Django sait que la requête est sans doute une contrefaçon et lève une erreur pour arrêter l'exécution.

Page Update

views.py

```
def band_update(request, id):
    band = Band.objects.get(id=id)
    # form = BandForm(instance=band)  # on pré-remplir le formulaire
avec un groupe existant
    if request.method == 'POST':
        form = BandForm(request.POST, instance=band)
        if form.is_valid():
            # mettre à jour le groupe existant dans la base de données
            form.save()
```

```
            # rediriger vers la page détaillée du groupe que nous
venons de mettre à jour
            return redirect('band-detail', band.id)
    else:
        form = BandForm(instance=band)
    return render(request,
                  'listings/band_update.html',
                  {'form': form})
```

L'instance permet de remplir le formulaire avec les données pré-existants
band_update.html

```
<h1>Mise à jour du groupe</h1>

<form action="" method="post">
{% csrf_token %}
{{ form.as_p }}
<input type="submit" value="Envoyer">
</form>
```

urls.py

```
path('bands/<int:id>/change/', views.band_update, name='band-update'),
```

Supprimer des objets

views.py

```
def band_delete(request, id):
    band = Band.objects.get(id=id)
    if request.method == 'POST':
        # supprimer le groupe de la base de données
        band.delete()
        # rediriger vers la liste des groupes
        return redirect('band_list')

    # pas besoin de « else » ici. Si c'est une demande GET, continuez
simplement
    return render(request,
           'listings/band_delete.html',
           {'band': band})
```

band_delete.html

```
<h1>Supprimer le groupe</h1>

<p>Etes-vous sûr de vouloir supprimer le groupe : {{ band.name }} ?</p>

<form action="" method="post">
{% csrf_token %}
<input type="submit" value="Supprimer">
</form>
```