Django

(Framework python complet et open source, conçu pour rendre le développement web efficace) s’occupe des taches courantes comme le traitement des requêtes http, le routage des url, et manipulation des bases de données.

Créer des modèles, vues, Templates.

Gérer des données avec l’ORM Django

Construire une interface CRUD.

Projet :

La bourse aux marchandises Merch Exchange. Il s'agira d'une application où les gens pourront répertorier divers articles de musique rares et de collection, tels que des disques, des affiches et des billets de concert. Les utilisateurs doivent pouvoir publier des listes des articles qu'ils souhaitent vendre.

Une image contenant texte, capture d’écran, écran

Description générée automatiquement

PyCharm : environnement de développement intégré, utilise une fonction de sauvegarde automatique, qui peut déclencher un rechargement du serveur de développement sans sauvegarde manuelle !

**python -m venv env** : on créée l’environnement virtuel env.

**env\Scripts\activate** : on active l’environnement virtuel.

Gardons la trace de tous ces packages dans un fichier requirements.txt. Il existe un raccourci très pratique pour faire cela :

→ **pip freeze > requirements.txt**

Une image contenant texte

Description générée automatiquement

**django-admin startproject merchex**: permet de créer une poignée de fichier du projet d’une manière automatique.

* Cela génère notre code de base Django : code de démarrage.

Une image contenant texte

Description générée automatiquement

Nous avons généré notre code de base en lançant la commande **django-admin(utilitaire de ligne de commande),** suivie de la sous-commande **startprojec**t et en passant l'argument **merchex** comme nom à utiliser pour le projet.

Une image contenant texte

Description générée automatiquement



Le manage.py remplace django-admin.

**Python manage.py runserver** = permet de lancer le serveur

Une image contenant texte

Description générée automatiquement

Les **migrations** représentent un moyen de configurer la base de données de notre application.

Ctrl + c = arrête le serveur.

**python manage.py migrate :** permet de créer un fichier de base de donnée.

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement



DNS domain name system.

Dans Django, une **application** est une sous-section de votre projet entier. Django nous encourage à compartimenter notre projet entier Django en applications, pour deux raisons principales :

* Cela permet de garder notre projet organisé et gérable au fur et à mesure qu'il se développe ;
* Cela signifie qu'une application peut éventuellement être réutilisée dans plusieurs projets.

Chaque application doit avoir un nom approprié qui représente le concept dont l'application est responsable. Le premier concept que nous voulons développer dans Merch Exchange est la liste des marchandises. Nous allons donc nommer notre première application « listings ».

**Python manage.py startapp listings** : on commence une application pour lister les marchandises qu’on nomme listings

La dernière étape de l'ajout de notre application « listings » à notre projet « merchex » consiste à « installer » l'application dans le projet.

La structure de notre projet :

* django-web-app/ - *le répertoire racine de notre repository*
* .gitignore
* requirements.txt - *une liste des packages requis*
* merchex/ - *le répertoire contenant notre projet Django, l'application, la base de données et l'utilitaire de ligne de commande*
* manage.py - *le script utilitaire de ligne de commande de Django*
* db.sqlite3 - *le fichier de la base de données de Django*
* merchex/ - *le répertoire du projet, généré par « django-admin startproject merchex » : la « tour de contrôle » de notre projet*
* settings.py - *la configuration de l'ensemble du projet*
* *...et d'autres fichiers relatifs au projet.*
* listings/ - *le répertoire de l’application généré par « python manage.py startapp listings »*
* *...les fichiers spécifiques aux applications que nous explorerons tout au long du cours*

**UNE VUE V**

*Une vue a pour fonction de répondre à la visite d'un utilisateur sur le site en renvoyant une page que l’utilisateur peut voir.*

*Ou en termes de Python :*

*Une vue est une fonction qui accepte un objet HttpRequest comme paramètre et retourne un objetHttpResponse .*

*Un****modèle d'URL****, c’est la façon dont nous indiquons à Django qu'il doit être à l'écoute d'une requête pour une URL donnée, puis appeler une vue spécifique pour générer une page.*

*URL //*

*Toute interaction avec un site web commence par une URL. Vous pouvez la taper directement dans votre barre d'adresse, la retrouver dans un signet ou cliquer sur un lien à partir d'une autre page ou d'un autre site, mais le résultat est le même : vous devez entrer une URL dans votre barre d'adresse avant que quoi que ce soit d'autre puisse se produire.*

*Sous le capot, votre navigateur commence à réfléchir à ce qu'il doit faire ensuite. Tout d'abord, il doit connaître l'adresse IP du serveur sur lequel ce site est hébergé. Pour ce faire, il effectue une recherche DNS sur le****nom de domaine****ou le****nom d'hôte****, c'est-à-dire tout ce qui se trouve entre les « :// » et le premier « / »*

*Maintenant que le navigateur sait où se trouve le serveur, il prépare une requête HTTP pour y aller. La demande contient :*

* *une méthode : dans notre cas,GET. On peut considérer que cela signifie « Obtenez-moi cette page, s'il vous plaît » ;*
* *le chemin de l'URL : c'est-à-dire tout ce qui se trouve à partir du premier slash (/).*

*Lorsque la demande arrive sur notre serveur, c'est maintenant au tour de notre application de déterminer ce que veut la requête. Elle le fait en examinant le chemin.*

*Voici quelques chemins URL que l'on peut s'attendre à voir dans les requêtes HTTP adressées à notre application web :*

* */about-us/*
* */contact-us/*
* */help/*

*Django a besoin d'un moyen de distinguer ces chemins, afin de pouvoir répondre avec le bon contenu.*

*Et c'est là que les modèles d'URL entrent en jeu.*

*Une image contenant texte

Description générée automatiquement*

* *Le premier argument est une chaîne de caractères. Il s'agit du chemin URL que nous allons faire correspondre : « about-us/ ».*
* *Le second argument doit être une fonction de vue que nous avons définie dans views.py. La demande sera transmise à cette vue, et la vue générera une page.*

*Une image contenant texte

Description générée automatiquement*

*Lorsqu'une URL correspond à un modèle d'URL, le déroulement passe à l'étape suivante : la demande HTTP est transmise à la vue spécifiée.*

*C'est maintenant au tour de notre vue de générer une réponse HTTP qui peut être envoyée au navigateur.*

*Django propose le contenu sans avoir besoin d'un fichier physique pour chaque page*

***En résumé***

* *Lorsque Django reçoit une requête HTTP, il tente de trouver une correspondance pour le chemin de cette requête dans une liste de modèles d'URL, définis dans urls.py.*
* *Si le chemin correspond à un modèle d'URL, Django transmet la requête à la vue correspondante, que nous définissons dans views.py.*
* *La vue est une fonction qui accepte la demande en tant que paramètre et qui renvoie une HttpResponse comme valeur de retour. Cette réponse contient le HTML que le navigateur utilise pour afficher la page.*

**ORM :** OBJECT RELATIONAL MAPPING programme qui nous permette de manipuler des classes python, classes qui seront convertis en tables et champs de bases de données

**Modèle :**

C’est l’élément qui définit les caractéristiques de ce qu’on veut stocker et manipuler dans notre base de données django. Possible grâce à l’ORM.

Vous créez un modèle dans Django en définissant une classe Python.

Notre modèle de groupe pourrait donc avoir comme caractéristiques un titre, un genre et l'année où elle est devenue active. Ces caractéristiques sont également connues sous le nom de **champs**.

Nous pouvons ensuite utiliser le modèle pour créer des objets individuels, ou instances, de ce modèle, qui ont chacun leurs propres caractéristiques uniques.

En général, dans les frameworks MVC et MVT, 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 : par exemple seulement pendant l'exécution de l'application.

De même, 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.

Tout ce que vous avez à faire est de faire en sorte que votre modèle hérite de la classe models.Model de Django.

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

Créons une migration qui nous aidera à construire une table

**python manage.py makemigrations (pour inclure le modèle dans notre base de données)**

Une image contenant texte

Description générée automatiquement

Cette migration va créer une table dans la base de données pour notre modèle de groupe.

**python manage.py migrate (actualization, on execute la migration)**

Une image contenant texte

Description générée automatiquement

Nous avons ajouté « listings » au   INSTALLED\_APPS  de notre projet. Django a recherché dans chacune de ces applications installées de nouvelles migrations à exécuter, il a trouvé notre nouvelle migration et l'a « appliquée » : il a exécuté ces instructions sur la base de données.

Une image contenant texte

Description générée automatiquement

Nous permettons d’ouvrir le shell. Utilisons le shell pour créer quelques objets de groupe, puis enregistrons ces objets dans la base de données.

Une image contenant texte

Description générée automatiquement

On importe Band, on instancie un nouvel objet, ‘le None signifie qu’il n’a pas d’identifiant, on save l’objet, et là son identifiant devient 1, il est stocké dans notre base de données.

Une image contenant texte

Description générée automatiquement

On fait la même chose mais en une seule ligne.

Une image contenant texte

Description générée automatiquement

On vérifie qu’on a bien 3 objets dans notre base.

**En résumé**

* Un modèle définit les caractéristiques et les comportements d'un objet dont vous voulez garder la trace dans votre application. Il ressemble beaucoup à une classe standard, mais en plus, un modèle sait comment enregistrer (« persister ») ses données dans une base de données.
* Une migration est un ensemble d'instructions qui font passer notre base de données d'un état à un autre, par exemple en créant une nouvelle table. Nous pouvons utiliser le CLI(commande line-interface) de Django pour générer et exécuter les migrations à notre place.
* Nous pouvons utiliser le shell de Django pour insérer de nouveaux objets dans notre base de données.
* Dans une vue, nous pouvons récupérer des objets dans la base de données et afficher leurs données dans nos pages.

**Templates :**

Un fichier html capable d’interpréter du code python, il reçoit des données depuis le modèle et intégrer des mécanismes comme les boucles. Donc on découpe notre programme en respectant modèle (stock les données) vue (récupère les données du modèle et les injectent dans le bon Template) Template qui s’occupe de l’affichage du code html et des données qui lui sont confiés

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Tout d'abord, nous importons la fonction render. Cet élément est probablement déjà présent puisqu'il est inclus dans le code de base, mais ajoutez-le si nécessaire.

Dans la déclaration de retour, nous n'appelons plus le constructeur **HttpResponse**. Au lieu de cela, nous appelons la fonction **render** avec 2 arguments :

* l'objet request qui est passé dans la fonction hello;
* une chaîne de caractères contenant le chemin d'accès au fichier gabarit que nous avons créé.

La fonction **render** crée un objet **HttpResponse** avec le HTML de notre modèle et le renvoie. Notre vue renvoie donc toujours une HttpResponse (ce qu'elle doit faire, pour être une vue).

def Hello(request):  
 bands = Band.objects.all()  
 listings = Listing.objects.all()  
 return render(request, 'listings/hello.html',{'first\_band': bands[0],'second\_band': bands[1],'third\_band': bands[2],'first\_listing': listings[0]} )

Dictionnaire contextuel, first\_band, Deviennent des variables qu’on peut utiliser.

Le code html

<html>  
 <head><title>Buy players</title></head>  
 <body>  
 <h1>Hello Django !</h1>  
 <p>My top players are :</p>  
 <ul>  
 <li>{{first\_band.name}} from {{first\_listing.title}}</li>  
 <li>{{second\_band.name}} from {{first\_listing.title}}</li>  
 <li>{{third\_band.name}} </li>  
 </ul>  
 </body>  
</html>

Le résultat :

Une image contenant texte

Description générée automatiquement

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.

Passant les tous

def Hello(request):  
 bands = Band.objects.all()  
 listings = Listing.objects.all()  
 return render(request, 'listings/hello.html',{'band':bands, 'listing':listings })

code html

<html>  
 <head><title>Buy players</title></head>  
 <body>  
 <h1>Hello Django !</h1>  
 <p>My top players are :</p>  
 <ul>  
 <li>{{band.0.name}} from {{listing.0.title}}</li>  
 <li>{{band.1.name}} from {{listing.0.title}}</li>  
 <li>{{band.2.name}} </li>  
 </ul>  
 </body>  
</html>

En résumé, les gabarits sont un moyen pour définir le contenu d'une page qui *ne change pas*. À l'intérieur de ces gabarits, nous insérons des variables de gabarits, qui servent d'espaces réservés pour le contenu qui *change*.

Print les names en utilisant une boucle

<html>  
 <head><title>Buy players</title></head>  
 <body>  
 <h1>Hello EGOIST!</h1>  
 <p>My top players are :  
 {% for band in band %}  
 {{ band.name }},  
 {% endfor %}  
 </p>  
 </body>  
</html>

Voici quelques éléments à prendre en compte :

* Les boucles et autres instructions logiques sont entourées de crochets et de signes de pourcentage ({% ... %}). Il s'agit de balises de gabarits.
* Une instructionfor est construite en utilisant une syntaxe similaire à celle de Python, mais sans les deux points de fin (: ).
* Nous avons tendance à utiliser le format conventionnel de Python for singulier in pluriel dans les instructions for.
* Une balise de gabarit for doit posséder une balise de fermeture endfor plus loin dans le gabarit.
* L'espace entre les balises for et endfor peut contenir du texte, du HTML et même des variables de gabarits Django.
* <html>  
   <head><title>Buy players</title></head>  
   <body>  
   <h1>Hello EGOIST!</h1>  
   <p>My top players are :  
   {% for band in band %}  
   <li>{{ band.name |upper}}</li>  
   {% endfor %}  
   </p>  
   </body>  
  </html>

Vous appliquez un filtre à une variable en utilisant la barre verticale ( | ).

<p> I love {{band|length}} player</p>

Une image contenant texte

Description générée automatiquement

* <p>
* J'ai..
* {% if band|length < 5 %}
* peu de
* {% elif band|length < 10 %}
* quelques
* {% else %}
* beaucoup de
* {% endif %}
* groupes préférés.
* </p>

En résumé

* Les gabarits sont l'endroit où nous définissons tous les éléments de présentation d'une page ; pour une application web, c'est le HTML.
* La vue peut ainsi se concentrer sur la logique, dont la récupération des données correctes à injecter dans la page.
* Nous injectons des données dans un gabarit à l'aide de variables de gabarits.
* Nous utilisons les balises de gabarits pour les boucles, les embranchements et le formatage dans les gabarits.

Une image contenant texte

Description générée automatiquement

On a ajouté une balise block qu’on appelle content.

Nous pouvons considérer la balise block comme un espace réservé, dans lequel nous pouvons injecter du contenu ; en ce sens, il est similaire aux variables de gabarits.

Une image contenant texte

Description générée automatiquement

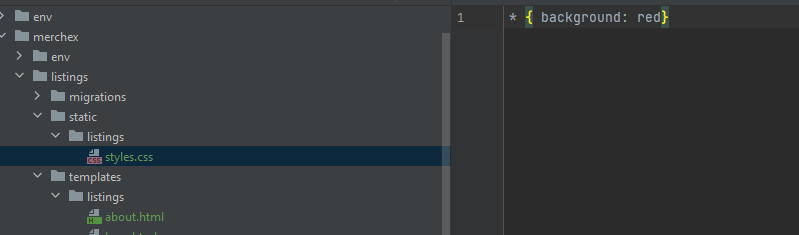
Une image contenant texte

Description générée automatiquement

La balise de gabarits extends en haut indique à Django que nous voulons que ce gabarit **hérite** de notre gabarit de base.

Les fichiers tels que les feuilles de style CSS sont appelés fichiers statiques dans Django, car une fois l'application en cours d'exécution, ils ne changent pas.

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 appeléstyles.css.



On fait le link entre notre page html et css

Une image contenant texte

Description générée automatiquement

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

Une image contenant texte

Description générée automatiquement

SGBD = système de gestion de base de données.

* Contrôle l’accès aux données
* Vérifie la structure des données
* Etablit des contraintes (ex, l’âge ne peut pas être négatif)

Les contraintes sur les champs des modèles

Exp le modèle Band,

class Band(models.Model):  
 name = models.fields.CharField(max\_length=100)

on ajoutera plusieurs champs:

class Band(models.Model):  
 name = models.fields.CharField(max\_length=100)  
 genre = models.fields.CharField(max\_lenght=100)  
 biography = models.fields.CharField(max\_lenght=100)  
 year\_formed = models.fields.IntegerField()  
 active = models.fields.BooleanField()  
 official\_page = models.fields.URLField()

year\_formed ne contiendra que l’année donc un entier (ex 2000), si c’était une date nous utiliserons plutôt DateField.

year\_formed = models.fields.IntegerField(validators = [MinValueValidator(1900), MaxValueValidator(2021)])

N’autorise que les années entre 1900 et 2021

class Band(models.Model):  
 name = models.fields.CharField(max\_length=100)  
 genre = models.fields.CharField(max\_lenght=100)  
 biography = models.fields.CharField(max\_lenght=100)  
 year\_formed = models.fields.IntegerField(validators = [MinValueValidator(1900), MaxValueValidator(2021)])  
 active = models.fields.BooleanField(default=True)  
 official\_page = models.fields.URLField(null=True, blank=True)

* Pour les CharField, l'option max\_length est obligatoire, nous obtenons une erreur si nous essayons d'exécuter l'application sans elle.
* Le null=True dans official\_page permet d’autoriser de laisser ce champs vide si un groupe n’as pas de page officiel
* Et 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 aimerions que le champ genre soit limité à une liste de choix que nous spécifions.

Une image contenant texte

Description générée automatiquement

On crée une classe Genre définissant les choix du genre, c’est une classe imbriquée, c'est une classe définie dans une autre classe.

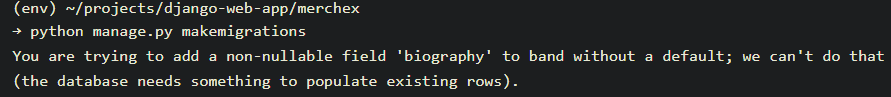
* La classe Genre hérite de models.TextChoices : c'est une classe dans Django qui est conçue pour définir une liste de choix.
* Dans Genre, nous ajoutons une constante pour chaque choix de genre que nous voulons autoriser, (par exemple HIP\_HOP).
* Pour chaque constante, nous définissons une clé ( 'H' ,'P' ,'R' ) : c'est la valeur qui sera stockée dans la base de données pour ce genre.

On essaie d’instancier des objets : on ne peut PAS

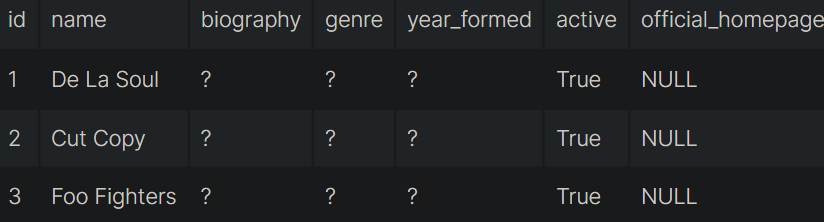
Une image contenant texte

Description générée automatiquement

Avant de commencer à créer des objets il faut repasser la commande makemigrations et migrate.



On devra ajouter des valeurs pour nos objets déjà crées



On passe une chaine vide pr biography ‘ ’, on choisit un genre parmi les choix, et une année 2000

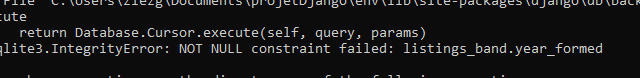
On corrigera tous cela après.

Apres le makemigrations , on exécute cette migration.

Une image contenant texte

Description générée automatiquement

On repasse dans le shell, on a une erreur, car year\_formed n’autorise pas le fait de contenir une valeur null







Pour les valeurs None , une tentative d’enregistrement comme NULL sera effectuée dans la base de données. Mettons donc year\_formed à quelque chose :



Pk on n’a pas eu le même prob pr name, ..

Une image contenant texte

Description générée automatiquement

Car django les a définis comme une chaîne vide, dont techniquement pas NULL

*Découvrons une interface que nous pouvons utiliser pour gérer ces modèles, qui est intégrée à Django :*

**Le site d'administration. CRUD**

C : create

R : read

U : update

D :delete

On commence par créer un superuser : un utilisateur qui peut tout faire.

**Python manage.py createsuperuser**

On va dans le fichier admin.py dans listings et on

from django.contrib import admin  
from listings.models import Band, Listing  
  
  
# Register your models here.  
admin.site.register(Band)

Ensuite, exécutez le serveur de développement, et dans votre navigateur, allez sur <http://127.0.0.1:8000/admin/> :

Une image contenant texte, capture d’écran, moniteur, noir

Description générée automatiquement

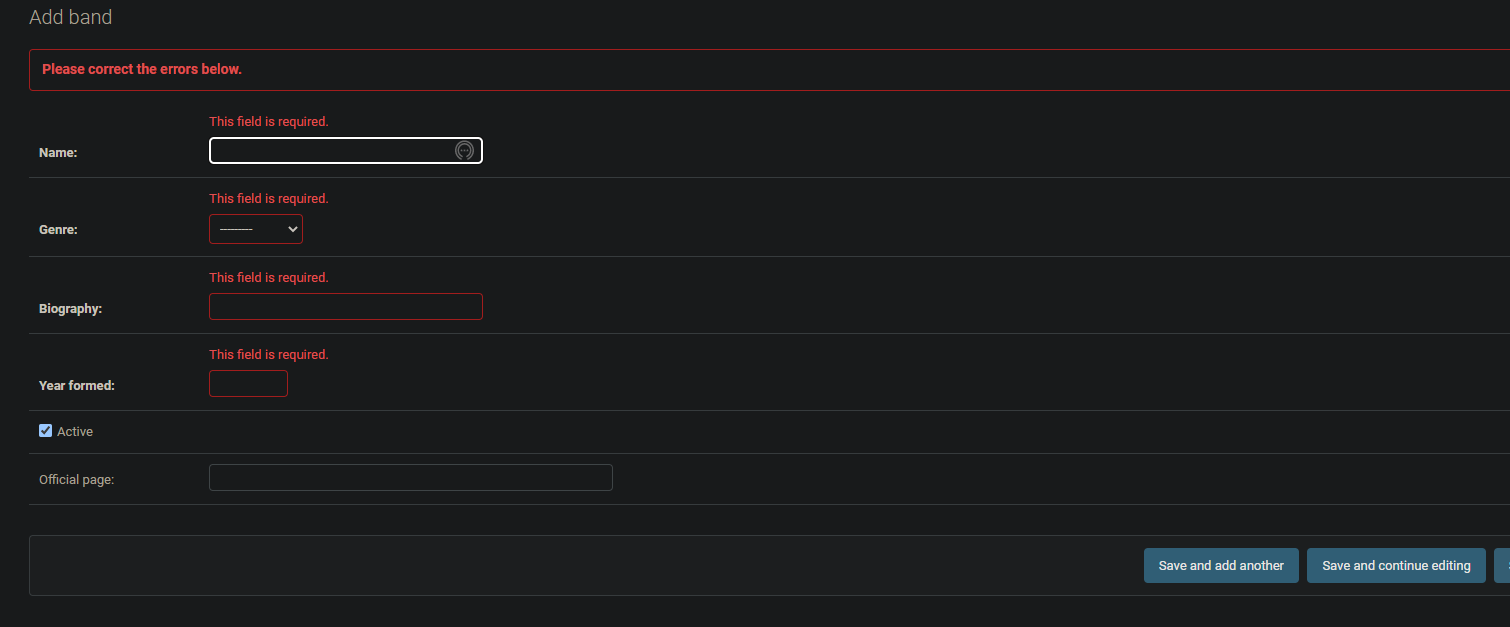
Une image contenant texte, capture d’écran, moniteur, noir

Description générée automatiquementUne image contenant texte, capture d’écran, écran, sombre

Description générée automatiquement



Noter qu’on ne peut pas save un object qui est vide



Car juste le champ d’official\_page qui ya blank = True

Une image contenant texte, moniteur, capture d’écran, écran

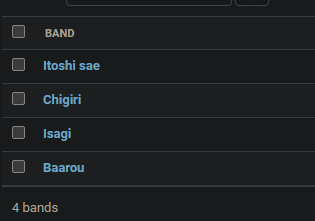
Description générée automatiquement

Au lien d’avoir Band object(x) on essaie d’afficher le nom.

Une image contenant texte

Description générée automatiquement

=>



Une image contenant texte

Description générée automatiquement

=>

Une image contenant texte

Description générée automatiquement

* Nous avons modifié la façon dont un Band est représenté sous forme de chaîne de caractères en utilisant la méthode \_\_str\_\_.
* Nous avons créé une 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.
* Nous avons donné à BandAdmin 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.
* Enfin, nous avons mis à jour l'appel à admin.site.register afin de lui passer la nouvelle classe BandAdmin. Cette dernière étape est nécessaire pour raccorder le tout.

En résumé

* Le site d'administration de Django est une interface utilisateur conçue pour que les administrateurs du site puissent effectuer manuellement des opérations CRUD sur les objets du modèle.
* Nous enregistrons nos modèles pour qu'ils apparaissent dans le site d'administration. Nous pouvons spécifier les champs que nous voulons afficher dans la vue en liste.
* Pour accéder au site d'administration, nous créons d'abord un compte superutilisateur avec python manage.py createsuperuser, puis nous nous connectons à <http://127.0.0.1:8000/admin/>.

Relations entre tables :

Many-to-one,

On suppose que chaque joueur ne pour être que dans une seule équipe mais une équipe peut avoir plusieurs joueurs

Cela suggère que nous devons ajouter un nouveau champ au modèle Band, pour agir comme un lien vers le modèle Listing, et il y a un type spécial de champ pour cela, appelé ForeignKey.

Une image contenant texte

Description générée automatiquement



Band.listing nous permet d’avoir le groupe de chaque joueur

* Nous passons également trois arguments à ForeignKey :
* Le modèle auquel on veut se rattacher : Listing ;
* null=True: parce que nous voulons permettre la création de joueurs même si ils 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 objetsBand sont supprimés. Il existe de multiples options pour cela, comme :
  + 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,

Nous ne voulons pas supprimer l'objet Band si une équipe est supprimé, nous utiliserons donc SET\_NULL.

En résumé

* + Nous pouvons créer des relations one-to-many en utilisant le champForeignKey.
  + Nous contrôlons la stratégie de ce qui se passe lorsqu'un modèle lié est supprimé, en utilisant l'argument on\_delete.
  + Les relations one-to-many peuvent être gérées dans l'administration de Django.

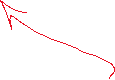
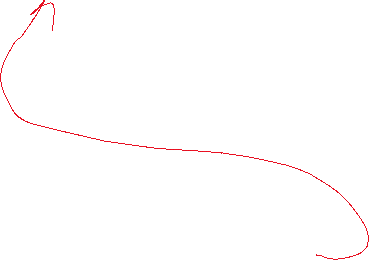
Si on commet une erreur et on exécute une mauvaise migration :

**python manage.py showmigrations :** permet de lister toutes les migrations

Identifiez ensuite la migration qui doit être annulée

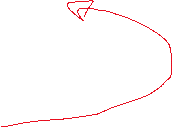
Une image contenant texte

Description générée automatiquement



Dernière migration dans listings ‘normalement’

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.



python manage.py migrate listings 0005\_listing\_band

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

**Conflit de migrations:**

Less occurrent when working alone

More occurrent when working in a group

2 personnes travaillent à part, faites de modifications, créent des migrations et commit sur git

Si on essaie de migrer les 2 migrations, message d’erreur car même identifiant

→ **python manage.py makemigrations –merge**

→ **python manage.py merge**

**Création d’une page web CRUD**

**On commence par le R : read**

Afficher les données à nos utilisateurs :

* Une **vue en liste**, qui affiche une liste de *tous* les objets d'un modèle avec un minimum de détails, par exemple juste le titre d’une annonce ou juste le nom du Groupe.
* Une  **vue détaillée**, qui affiche *un* objet avec tous ses détails et tous les champs affichés.

Une vue en liste :

Une image contenant texte

Description générée automatiquement

Une vue en détail :

Il faut ajouter une url

**Une image contenant texte

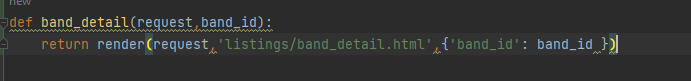
Description générée automatiquement**

Avec ce chemin, nous faisons correspondre « bands/ », suivi de n'importe quel nombre entier, suivi d'un slash final (« / »). Nous « capturons » l'entier du chemin en ajoutant <int:id> dans le chemin. Le int est le type de données que nous nous attendons à recevoir: un entier. Le id devient le nom d'un argument qui est passé à la vue.



On définie band\_detail()







Le gabarit

Une image contenant texte

Description générée automatiquement

Résultat

Une image contenant texte

Description générée automatiquement

Maintenant on essaye d’afficher le band avec cet id :

Une image contenant texte

Description générée automatiquement

Le gabarit

Une image contenant texte

Description générée automatiquement

Une image contenant texte, capture d’écran, orange, sombre

Description générée automatiquement

Nous voyons ici une nouvelle méthode: **Band.objects.get(**), dont le but est de retourner un seul objet. L’objettre code, nous écrivons **.get(id=id),** ce qui revient à dire « obtenez-moi l'objet qui a cet id ». Ensuite, nous passons ce band au modèle. On le rend plus détaillé :

Le gabarit :

Une image contenant texte

Description générée automatiquement

On obtient :

Une image contenant texte

Description générée automatiquement



On change le genre dans le gabarit

Get\_genre\_display

Une image contenant texte

Description générée automatiquement



Une image contenant texte

Description générée automatiquement



On veut convertir le true en yes :

On utilise yesno

Une image contenant texte

Description générée automatiquement





Nous ferons de la page d'accueil officielle un lien cliquable.

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

Une image contenant texte

Description générée automatiquement

Si id n’existe pas (dépasse 5)

Une image contenant texte

Description générée automatiquement

Ou

Une image contenant texte

Description générée automatiquement

N’oublie pas d’importer Http404 from django.http

**Lions la vue en liste et la vue détaillée :**

On peut changer le gabarit des bands listes :

Une image contenant texte

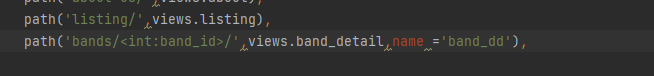
Description générée automatiquement

Mais c’est de l’anti-pattern

Donc on procède comme suit :

Une image contenant texte

Description générée automatiquement

En changeant l’url, en ajoutant un nom 

Url dans le fichier html est comme une fonction python.

Nous passons deux arguments à la balise url: band-detail et band.id. C'est comme passer des arguments à une fonction Python, mais dans les balises de gabarits, et nous n'avons pas besoin de saisir les parenthèses.

Faisons maintenant le sens inverse : vue détaillée -> vue liste :

Dans le gabarit de la vue détaillée

Une image contenant texte

Description générée automatiquement

En modifiant l’url :

Une image contenant texte

Description générée automatiquement

**On passe au C-U-D des opérateurs d’écriture sur les données**

Les formulaires permettent d’envoyer des données du front-end au back-end.

Créer un formulaire de « Contact » qui permet aux utilisateurs d'envoyer un message aux administrateurs de l'application.

**Une image contenant texte, capture d’écran, écran

Description générée automatiquement**



On crée un fichier forms.py, et on définit la classe contactusform.

Même forme que Model

Nous avons défini trois champs de formulaire dans notre **ContactUsForm**. Les champs de formulaire sont similaires aux champs de modèle : nous avons différents types de champs pour différents types de données. Nous pouvons également préciser quand les champs doivent être facultatifs avec **required=False**, ici, nous permettons à l'utilisateur de rester anonyme en rendant le champ**name**facultatif. Et nous pouvons définir **max\_length**, tout comme nous le faisons dans un modèle.

Une image contenant texte

Description générée automatiquement

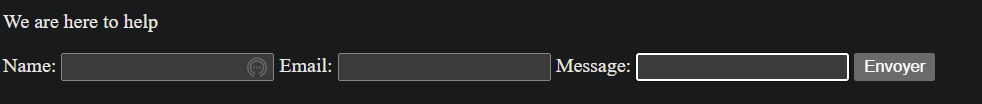
On ajoute cette page à notre urlpatterns

Une image contenant texte

Description générée automatiquement



On obtient :



Dans notre gabarit, nous avons ajouté une balise <form> avec un <input> de type="submit" à l'intérieur. C'est la norme pour tout formulaire HTML que vous pouvez créer.

Au lieu de taper manuellement les balises <input> pour chacun des champs de notre formulaire, nous tapons simplement {{ form }}. Nous ajoutons également {% csrf\_token %  }

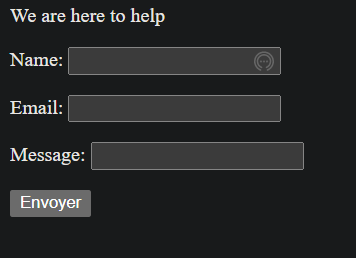
Une image contenant texte

Description générée automatiquement

En examinant le code générer par notre navigateur web :

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.

Pour afficher les ‘champs’ :



On modifie le gabarit de contact :

Une image contenant texte

Description générée automatiquement

* 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/". 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.

Donc on sait que nos données de notre formulaire arriveront au serveur sous forme de requête POST, on va voir comment gérer ces requêtes.

Tout d'abord, appelons cette vue comme une requête GET. Le moyen le plus sûr de le faire est de cliquer dans la barre d'adresse du navigateur et d'appuyer sur Entrée . (Cliquer sur « recharger » peut provoquer un POST si vous le faites immédiatement après avoir soumis le formulaire)

Il y a deux types de request : un de type GET c’est quand on va dans la barre de navigateur on clique sur Entré et l’autre de type post quand on remplit des champs et on clique sur Envoyer

Ensuite, nous devons d'une manière ou d'une autre gérer les deux scénarios de demande dans notre vue :

* S'il s'agit d'une requête GET, nous devons afficher un formulaire vide à l'utilisateur.
* S'il s'agit d'une demande POST, nous devons examiner les données et voir si elles sont valides

Pour cela, on utilise if :

Une image contenant texte

Description générée automatiquement

Dans les deux branches de l'instruction if, nous créons un formulaire qui est transmis au modèle, mais dans le cas d'une requête POST, nous remplissons également le formulaire avec les données POST.

Maintenant, lorsque nous soumettons un formulaire rempli, les données sont toujours visibles dans le navigateur lorsque la page est rechargée. Mais ce n'est pas tout : si nous soumettons des données non valides, le formulaire affichera des messages d'erreur : Une image contenant texte

Description générée automatiquement

Ce que nous voyons ici est une validation côté serveur : notre formulaire Django a validé les champs par rapport à nos règles, a généré des messages d'erreur en cas de problème, puis les a retournés dans le modèle comme faisant partie du formulaire.

J'ai ajouté. errorlist{ margin-bottom: -10px } à mon fichier CSS pour qu'il soit plus évident de savoir quelle erreur appartient à quelle entrée.

L'utilisateur peut alors modifier les valeurs et soumettre à nouveau le formulaire, et Django vérifiera à nouveau le formulaire. Ce cycle peut se répéter autant de fois que nécessaire jusqu'à ce que tous les champs du formulaire soient valides.

* si c'est une requête POST...
  + si les données du formulaire sont valides, envoyer un e-mail ;
  + si les données du formulaire ne sont pas valides, afficher à nouveau le formulaire avec des messages d'erreur (comme nous le faisons déjà).

Une image contenant texte

Description générée automatiquement Une image contenant texte

Description générée automatiquement

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().**

On ajoute dans le settings.py :

**redirect** est un raccourci pratique. Nous pouvons lui fournir un modèle d'URL avec des arguments ou directement un chemin d'URL. Une image contenant texte

Description générée automatiquement

avec l’import :

from django.shortcuts import redirect

Il faut créer le email-sent.

Nous utilisons la balise de **gabarit  url**  pour générer des liens dans un gabarit. Nous passons le  **name**  du motif de l'URL comme argument, par exemple  **{% url 'band-create' %}**

Tous les modèles d'URL configurés sont répertoriés, ainsi que le chemin actuel qui ne correspond pas. Si vous voyez cette erreur et que vous vous demandez pourquoi vous ne pouvez pas charger une page qui devrait exister, veillez à comparer soigneusement le chemin que vous avez demandé avec votre liste de modèles d'URL.



Urlpattern qui nous permet de Créer des band (le c de crud)

Dans le fichier forms.py

Une image contenant texte

Description générée automatiquement

Permet de définir un formulaire depuis un modèle.

La nouvelle classe contient une classe imbriquée, **Meta**, qui spécifie le modèle pour lequel ce formulaire sera utilisé, et les champs de ce modèle à inclure dans ce formulaire (dans ce cas, tous).