

DÉVELOPPEMENT WEB CÔTÉ SERVEUR FRAMEWORK PYTHON-DJANGO

TRAVAUX PRATIQUE N°3 (PROJET MYSITE : SUITE TP N°2) MODELS ET DJANGO ADMIN

Objectif du TP:

- Création d'une classe Model : Pages Model
- Utiliser le site Django Admin

Les modèles sont des classes Django qui permettent de faire le lien avec la base de données de l'application. Django dispose d'un **ORM** (Object Relational Mapping) qui permet, avec du code Python, d'effectuer toutes les manipulations de données stockées dans la base de données sans avoir recours au langage SQL.

Un modèle est simplement une classe python qui hérite de la classe **Model** du module **django.db.models**. Le nom d'un modèle est transformé par l'ORM de Django en une table de la base de données.

Les champs de modèle

Les attributs de la classe modèle sont désignés par champs et correspondent aux attributs de la table de la base de données. On peut utiliser différents types de champs, qui ressemblent beaucoup aux différents types natifs disponibles avec Python.

Parmi ces champs, on retiendra par exemple les champs :

CharField <code>class SlugField(max_length=50, **options)</code>	Un champ pour les chaînes de caractères dont le nombre de caractères ne dépasse pas 255. Pour de long texte, utilisez TextField . Le composant de formulaire par défaut de ce champ est un TextInput .
TextField <code>class TextField(**options)</code>	Un champ de texte long. Le composant de formulaire par défaut de ce champ est un Textarea . Si vous indiquez un attribut max_length , celui-ci se répercute sur le composant Textarea du champ de formulaire généré automatiquement. Cependant, la limite n'est pas imposée au niveau du modèle, ni de la base de données. Pour cela, utilisez plutôt un champ CharField .
SlugField <code>class SlugField(max_length=50, **options)</code>	Slug est un terme anglophone de journalisme. Un slug est une brève étiquette d'un contenu, composée uniquement de lettres, de chiffres, de soulignements ou de tirets. Ils sont généralement utilisés dans les URL.
IntegerField <code>class IntegerField(**options)</code>	Stocke Un nombre entier. Les valeurs de -2147483648 à 2147483647 sont acceptées par toutes les base de données prises en charge par Django. Le composant de formulaire par défaut de ce champ est un NumberInput lorsque localize vaut False , ou TextInput dans le cas contraire.
FloatField <code>class FloatField(**options)</code>	Stock un nombre flottant représenté en Python par une instance de float . Le composant de formulaire par défaut de ce champ est un NumberInput lorsque localize vaut False , ou TextInput dans le cas contraire.
DecimalField <code>class DecimalField(max_digits=None, decimal_places=None, **options)</code>	Stocke un nombre décimal de taille fixe, représenté en Python par une instance de Decimal . Les saisies sont validées en utilisant DecimalValidator . Il requiert deux paramètres obligatoires : max_digits et decimal_places . Le composant de formulaire par défaut de ce champ est un NumberInput lorsque localize vaut False , ou TextInput dans le cas contraire. Note : Il est important d'être conscient des limites de votre SGBD à propos des champs décimaux.

BooleanField <code>class BooleanField(**options)</code>	<p>Stocke une valeur booléenne vrai ou faux.</p> <p>Le composant de formulaire par défaut de ce champ est un CheckboxInput, ou NullBooleanSelect si <code>null=True</code>.</p> <p>La valeur par défaut de BooleanField est None lorsque le paramètre default n'est pas défini.</p>
DateField <code>class DateField(auto_now=False, auto_now_add=False, **options)</code>	<p>Stocke une date, représentée en Python par une instance de datetime.date.</p> <p>Accepte quelques paramètres supplémentaires et facultatifs :</p> <p>auto_now : Assigne automatiquement la valeur du champ à la date du jour lors de chaque enregistrement de l'objet. Utile pour les horodatages de « dernière modification ». Notez que c'est <i>toujours</i> la date actuelle qui est utilisée ; il ne s'agit pas seulement d'une valeur par défaut que l'on peut surcharger.</p> <p>La mise à jour automatique de ce champ ne se produit que lors de l'appel à Model.save(). Le champ n'est pas mis à jour lors de mises à jour d'autres champs par d'autres façons comme par exemple QuerySet.update(), bien qu'il soit possible de spécifier explicitement une valeur pour ce champ lors d'une telle mise à jour.</p> <p>auto_now_add : Assigne automatiquement la valeur du champ à la date du jour lors du premier enregistrement de l'objet. Utile pour les horodatages de date de création. Notez que c'est <i>toujours</i> la date actuelle qui est utilisée ; il ne s'agit pas seulement d'une valeur par défaut que l'on peut surcharger. Ainsi, même si vous définissez une valeur pour ce champ lors de la création de l'objet, il sera ignoré. Si vous voulez pouvoir modifier ce champ, utilisez ce qui suit au lieu de définir auto_now_add=True :</p> <p>il faut définir le paramètre default=date.today - datetime.date.today()</p> <p>Le composant de formulaire par défaut de ce champ est un DateInput. L'interface d'administration ajoute un calendrier JavaScript ainsi qu'un raccourci pour « Aujourd'hui ». Contient une clé supplémentaire de message d'erreur invalid_date.</p>
DateTimeField <code>class DateTimeField(auto_now=False, auto_now_add=False, **options)</code>	<p>Une date et une heure, représentées en Python par une instance de datetime.datetime. Ce champ accepte les mêmes paramètres supplémentaires que le champ DateField.</p> <p>Le composant de formulaire par défaut de ce champ est un DateTimeInput. L'interface d'administration utilise deux composants TextInput séparés avec des raccourcis JavaScript.</p>
TimeField <code>class TimeField(auto_now=False, auto_now_add=False, **options)</code>	<p>Une heure, représentée en Python par une instance de datetime.time. Ce champ accepte les mêmes options d'auto-complétion qu'un champ DateField.</p> <p>Le composant de formulaire par défaut de ce champ est un TimeInput. L'interface d'administration ajoute des raccourcis JavaScript.</p>
EmailField <code>class EmailField(max_length=254, **options)</code>	<p>Un champ CharField qui vérifie que sa valeur est une adresse électronique valide en utilisant EmailValidator.</p>
URLField <code>class URLField(max_length=200, **options)</code>	<p>C'est un champ CharField pour les URL, validé par URLValidator.</p> <p>Le composant de formulaire par défaut de ce champ est un URLInput.</p> <p>Comme pour les autres sous-classes de CharField, URLField accepte le paramètre</p>

	<p>facultatif max_length. Si vous ne renseignez pas la valeur de max_length, elle prend la valeur 200 par défaut.</p>
<p>FileField</p> <p>class FileField(upload_to=None, max_length=100, **options)</p>	<p>Un champ de fichier à téléverser. Possède deux paramètres facultatifs :</p> <p>upload_to : Cet attribut permet de définir le répertoire de téléversement et le nom de fichier. Il peut être défini de deux manières. Dans les deux cas, la valeur est transmise à la méthode Storage.save().</p> <p>Si vous indiquez une valeur textuelle ou un chemin Path, la valeur peut contenir des chaînes de format strftime() qui seront remplacées par la date/heure du fichier téléversé (permettant ainsi de ne pas remplir exagérément le répertoire indiqué). Par exemple :</p> <pre>class MyModel(models.Model): # file will be uploaded to MEDIA_ROOT/uploads upload = models.FileField(upload_to='uploads/')</pre>
<p>ImageField</p> <p>class ImageField(upload_to=None, height_field=None, width_field=None, max_length=100, **options)</p>	<p>Hérite de tous les attributs et méthodes de FileField, mais valide également que l'objet téléversé soit une image valide.</p> <p>En complément des attributs spéciaux disponibles pour FileField, un champ ImageField possède aussi les attributs height et width.</p> <p>Pour faciliter l'interrogation de ces attributs, ImageField possède deux attributs facultatifs supplémentaires :</p> <p>ImageField.height_field</p> <p>Le nom d'un champ du modèle qui sera automatiquement renseigné avec la hauteur de l'image à chaque enregistrement de l'instance du modèle.</p> <p>ImageField.width_field</p> <p>Le nom d'un champ du modèle qui sera automatiquement renseigné avec la largeur de l'image à chaque enregistrement de l'instance du modèle.</p> <p>Nécessite la bibliothèque Pillow.</p> <p>Les instances de ImageField sont créées en tant que colonnes varchar dans la base de données avec une longueur par défaut maximale de 100 caractères. Comme pour d'autres champs, vous pouvez modifier la taille maximale en utilisant le paramètre max_length.</p>

	Le composant de formulaire par défaut de ce champ est un ClearableFileInput .
--	--

Le model Pages

La façon la plus simple d'apprendre comment les modèles fonctionnent est de créer un modèle – L'exemple ci-dessous crée le modèle Page de l'application pages:

E:/mydjango/mysite/pages/models.py

```
1 from django.db import models
2
3 class Page(models.Model):
4     title = models.CharField(max_length=60)
5     permalink = models.CharField(max_length=12, unique=True)
6     update_date = models.DateTimeField(verbose_name='Last Updated')
7     bodytext = models.TextField('Page Content', blank=True)
```

Maintenant, regardons de plus près le modèle Page :

- Ligne 1. Importez le package de modèles à partir de `django.db`. Si vous avez utilisé `startapp`, cette ligne sera ajoutée automatiquement dans votre fichier.
- Ligne 3. Créez la classe `Page`, qui doit **hériter** de la classe `Model` de Django.
- Lignes 4 à 7. Définissez les champs du modèle. A chaque champ du modèle, correspondra un champ dans la table qui correspond au modèle `Page`. Cette table porte le nom "`pages_page`" (<nom de l'application>_<nom du modèle>) et est créée automatiquement par l'ORM de Django. Nous verrons en détail dans un TP futur l'utilisation de chacun des champs du modèle :
 - ✓ **titre**. Le titre de votre page et sera placé dans l'élément `<title>` de votre template.
 - ✓ **Permalink**. Un lien permanent vers une page individuelle. Lien qui permet d'accéder à la page dont le code HTML du corps est stocké dans le champ **bodytext**. Cela aura plus de sens dans un TP futur lorsque nous rédigerons de nouvelles URL pour accéder à vos pages.
 - ✓ **update_date**. La date à laquelle la page a été mise à jour pour la dernière fois.
 - ✓ **bodytext**. Le contenu HTML de votre page Cela sera placé dans l'élément `<body>` de votre template.

Chacun de nos champs de modèle possède un type de champ Django associé et des options de champ. Le modèle `Page` utilise trois types de champs différents : `CharField`, `DateTimeField` et `TextField`. Jetons un coup d'œil aux types de champs et aux options plus en détail :

- **titre** : Un **CharField** est une courte ligne de texte (255 caractères maximum). Dans ce cas, l'option `max_length` définit la longueur maximale du titre de la page à 60 caractères.
- **Permalink** : Ce champ est aussi de type `CharField` et a deux options : `max_length = 12` ; et `unique = True`. **Permalink** contiendra l'URL vers une page, nous ne voulons pas qu'il soit dupliqué, donc cette option assure qu'une erreur sera jetée si vous essayez d'entrer une même valeur de **Permalink** pour deux pages différentes.
- **update_date** : Un champ **DateTimeField** enregistre un objet `datetime` Python. Pour créer un nom convivial pour ce champ, nous avons utilisé l'option `verbose_name = 'Last Updated'`.
- **bodytext**. Un **TextField** est un champ de texte volumineux pouvant contenir plusieurs milliers de caractères (le maximum dépend de la base de données). Ce champ possède deux options
 - **verbose-name** = « Page Content » définit un nom convivial pour le champ.
 - **blank = True**- pour autoriser de créer une page sans aucun contenu. La valeur par défaut pour cette option est `False`, donc si vous n'ajoutez aucun contenu de page, Django lance une erreur.

Nous avons couvert seulement quelques types de champs et options dans notre premier modèle. Si vous souhaitez créer des modèles avec des types de champs et des options différents, vous trouverez des tableaux de référence pour les types de champs communs et les options dans l'annexe.

Pour qu'un modèle soit accessible depuis le **site admin** de Django, il doit être enregistré. L'enregistrement et la configuration d'un modèle pour le **site admin** se font en ajoutant du code dans le fichier **admin.py** de l'application (modifications en gras):

D:/mydjango/mysite/pages/admin.py

```
1 from django.contrib import admin
2 from .models import Page
3
4 admin.site.register(Page)
```

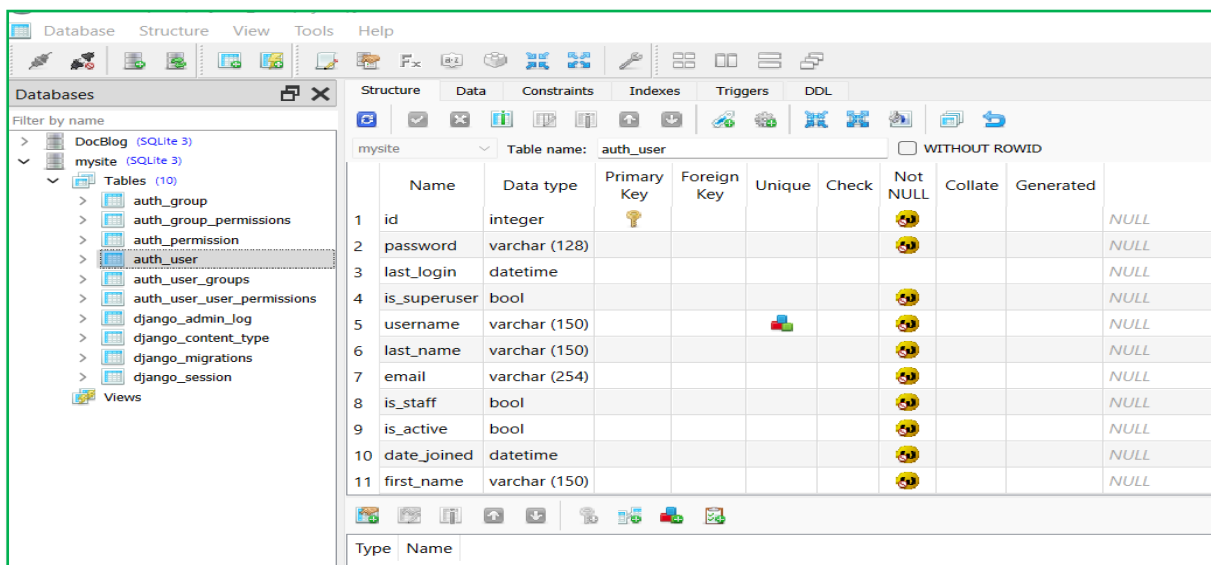
Nous avons ajouté deux lignes de code à notre fichier **admin.py** de l'application **pages**:

Ligne 2. Nous importons le modèle **Page** du fichier **models** de l'application

Ligne 4. Nous enregistrons le modèle **Page** avec l'admin

Maintenant nous devons juste créer une migration, pour que Django puisse créer la table qui correspond au modèle et l'ajouter à la base de données.

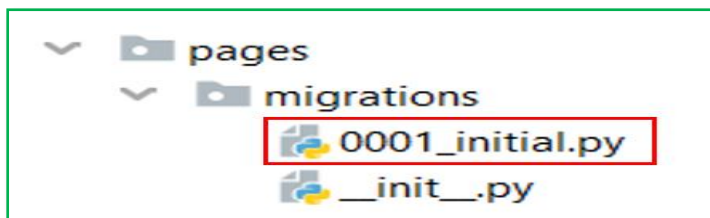
Avant d'effectuer la migration, ouvrons SQLite Studio pour examiner le contenu de la base de données du projet.



Préparons la migration :

```
(venv) D:\mydjango\mysite>python manage.py makemigrations pages
Migrations for 'pages':
  pages\migrations\0001_initial.py
    - Create model Page
```

Le fichier **0001_initial.py** est créé et ajouté au dossier migrations



Voici ci-dessous le contenu du fichier **0001_initial.py**:

```

from django.db import migrations, models

class Migration(migrations.Migration):
    initial = True
    dependencies = [
    ]
    operations = [
        migrations.CreateModel(
            name='Page',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('title', models.CharField(max_length=60)),
                ('permalink', models.CharField(max_length=12, unique=True)),
                ('update_date', models.DateTimeField(verbose_name='Last Updated')),
                ('bodytext', models.TextField(blank=True, verbose_name='Page Content')),
            ],
        ),
    ]

```

Remarque : si vous voulez le code SQL qui correspond à ce fichier 0001_initial.py avant d'effectuer la migration réelle, vous pouvez exécuter la commande :

(venv)E:\mydjango\mysite>python manage.py sqlmigrate app 0001

Ensuite, nous devons effectuer la migration réelle:

(venv) D:\mydjango\mysite>python manage.py migrate

Operations to perform:

Apply all migrations: admin, auth, contenttypes, pages, sessions

Running migrations:

Applying pages.0001_initial... OK

Examinons le contenu de la base de données du projet dans SQLiteStudio :

The screenshot shows the SQLiteStudio interface. On the left, the 'Databases' pane shows the 'mysite' database with its tables. The 'pages_page' table is selected. The main pane displays the table's structure with the following columns:

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Generated
1	id	integer	Yes				Yes		NULL
2	title	varchar (60)					Yes		NULL
3	permalink	varchar (12)			Yes		Yes		NULL
4	update_date	datetime					Yes		NULL
5	bodytext	text					Yes		NULL

Nous remarquons que Django a créé une nouvelle table dans la base de donnée avec le nom **pages_page** (<nom_application>_<nom du modele> en minuscule. Nous remarquons qu'un nouveau champ **id** qui joue le rôle de clé primaire a été ajouté aux champs du modèle Page.

L'admin de Django est une application qui permet de gérer le site. Il permet de gérer les utilisateurs, les groupes et les modèles. Dans les sections suivantes, nous allons apprendre à l'utiliser.

Un premier regard sur l'admin de Django

Pour la plupart des sites Web modernes, une interface d'administration (ou admin pour faire court) est une partie essentielle de l'infrastructure. Il s'agit d'une interface Web, limitée aux administrateurs de site de confiance, qui permet à un administrateur d'ajouter, de modifier et de supprimer du contenu de site.

Django est livré avec une interface d'administration intégrée. Avec l'admin de Django, vous pouvez authentifier les utilisateurs, afficher et gérer les formulaires et valider les entrées. Django fournit également une interface pratique à nos modèles, qui est ce que nous allons utiliser maintenant pour ajouter du contenu à notre application **pages**.

Utilisation du Site Admin

Lorsque vous avez exécuté startproject, Django a créé et configuré le site admin par défaut pour vous. Tout ce que vous devez faire est de créer un utilisateur admin (superutilisateur) pour vous connecter au site d'administration. Pour créer un utilisateur admin, exécutez la commande suivante à partir de votre environnement virtuel:

```
(venv) D:\mydjango\mysite>python manage.py createsuperuser
Username (leave blank to use 'lhass'): admin
Email address: admin@gmail.com
Password: Donner un mot de passe
Password (again): Confirmer le mot de passe
The password is too similar to the username.
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Maintenant que nous avons créé un utilisateur admin, nous sommes prêts à utiliser le site admin de Django. Lançons le serveur de développement et explorons.

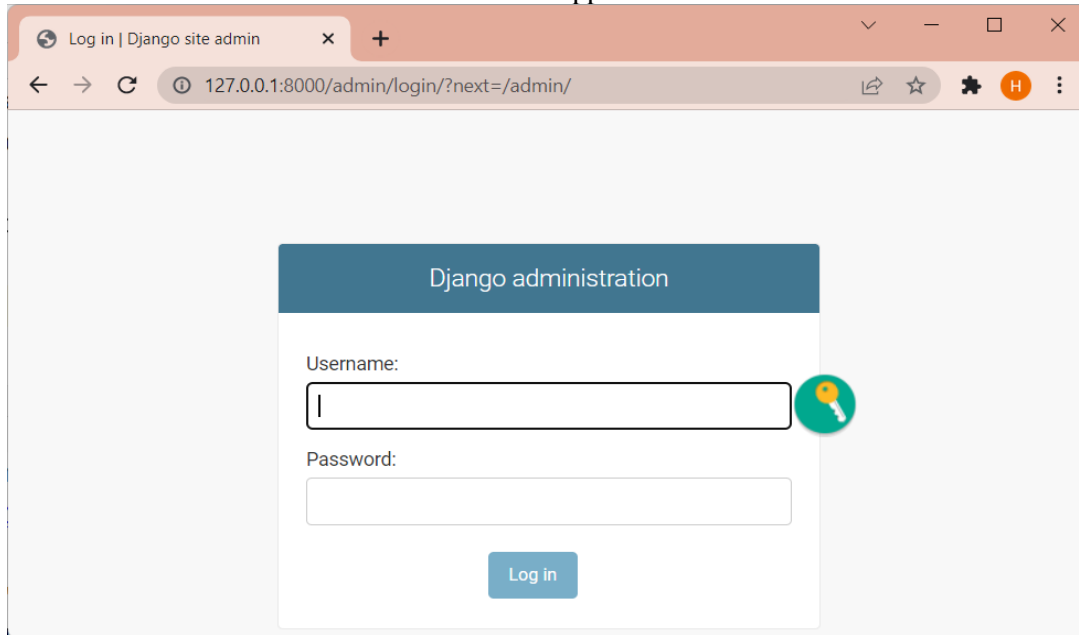
Lançons le serveur web

```
(venv) D:\mydjango\mysite>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

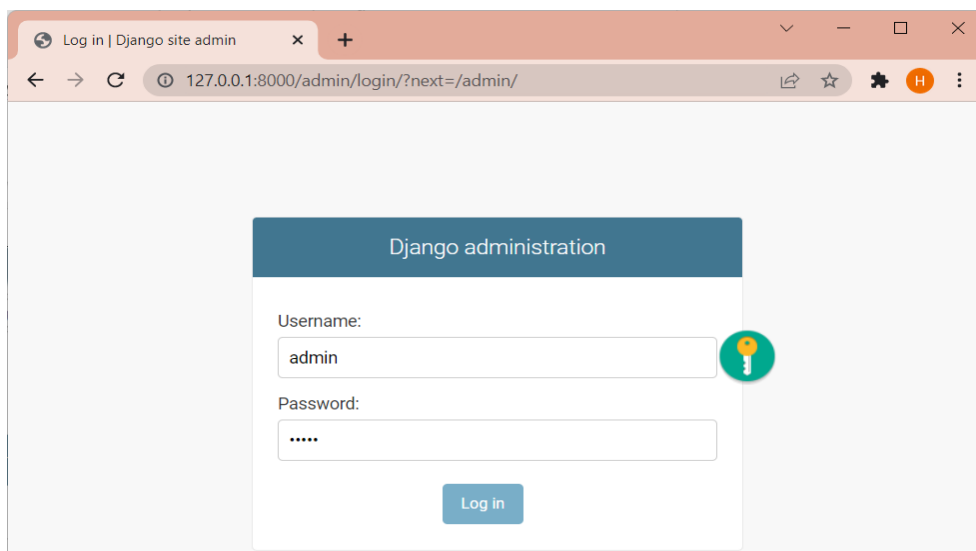
System check identified no issues (0 silenced).
March 04, 2022 - 18:51:42
Django version 3.1.7, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Ensuite, ouvrons le navigateur Web et saisissons l'url <http://127.0.0.1:8000/admin/>.

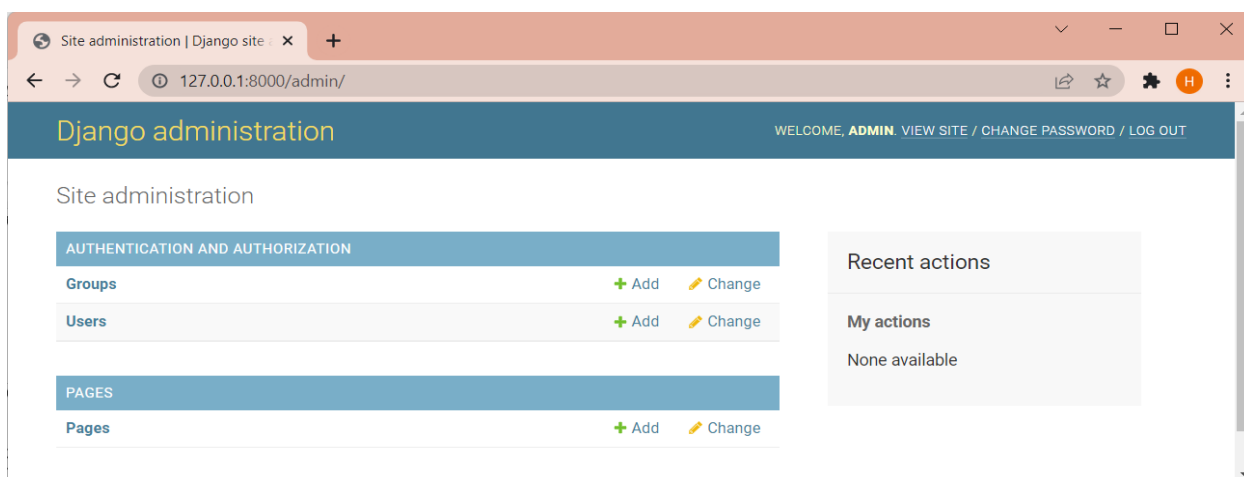
L'écran de connexion au site admin ci-dessous apparaît.



Connectons-nous au site admin de Django avec le nom d'utilisateur et le mot de passe que nous venons de créer.



Une fois connecté, nous devrions voir la page d'index d'administrateur de Django ci-dessous.



En haut de la page d'index se trouve le groupe « **AUTHENTICATION AND AUTHORIZATION** » avec deux types de contenu modifiable: **Groups** et **Users**.

Ils sont fournis par le framework d'authentification inclus dans Django. Nous regarderons les utilisateurs et les groupes dans un futur TP.

Sous le groupe **Authentication and Authorization** se trouve un groupe nommé **PAGES** ajouté par l'administrateur pour notre modèle **Page de notre application pages**. Nous allons utiliser ce groupe pour ajouter du contenu de page à notre site. Pour ce faire cliquez sur le lien « Add » à côté du signe + en couleur verte à droite de l'entrée Pages.

Le site admin est conçu pour être utilisé par des utilisateurs non techniques, et en tant que tel, il devrait être explicite. Néanmoins, couvrons quelques-unes des caractéristiques de base.

Chaque type de données dans le site admin de Django possède une *liste de modifications* et un *formulaire d'édition*.

Les *listes de modifications* vous montrent tous les objets disponibles dans la base de données et les *formulaires d'édition* vous permettent d'ajouter, de modifier ou de supprimer des enregistrements particuliers dans votre base de données. La figure ci-dessus montre le formulaire d'édition ouvert lorsque vous avez cliqué sur le lien **Ajouter**.

Lorsque vous ajoutez un enregistrement, le formulaire d'édition est vide, ce qui vous permet d'entrer de nouvelles informations dans la base de données. Remplissez les champs comme suit:

Lorsque vous entrez le contenu de la page, n'oubliez pas qu'il doit être du HTML pour bien s'afficher dans votre navigateur. À ce stade, une en-tête ou deux paragraphes est suffisant.

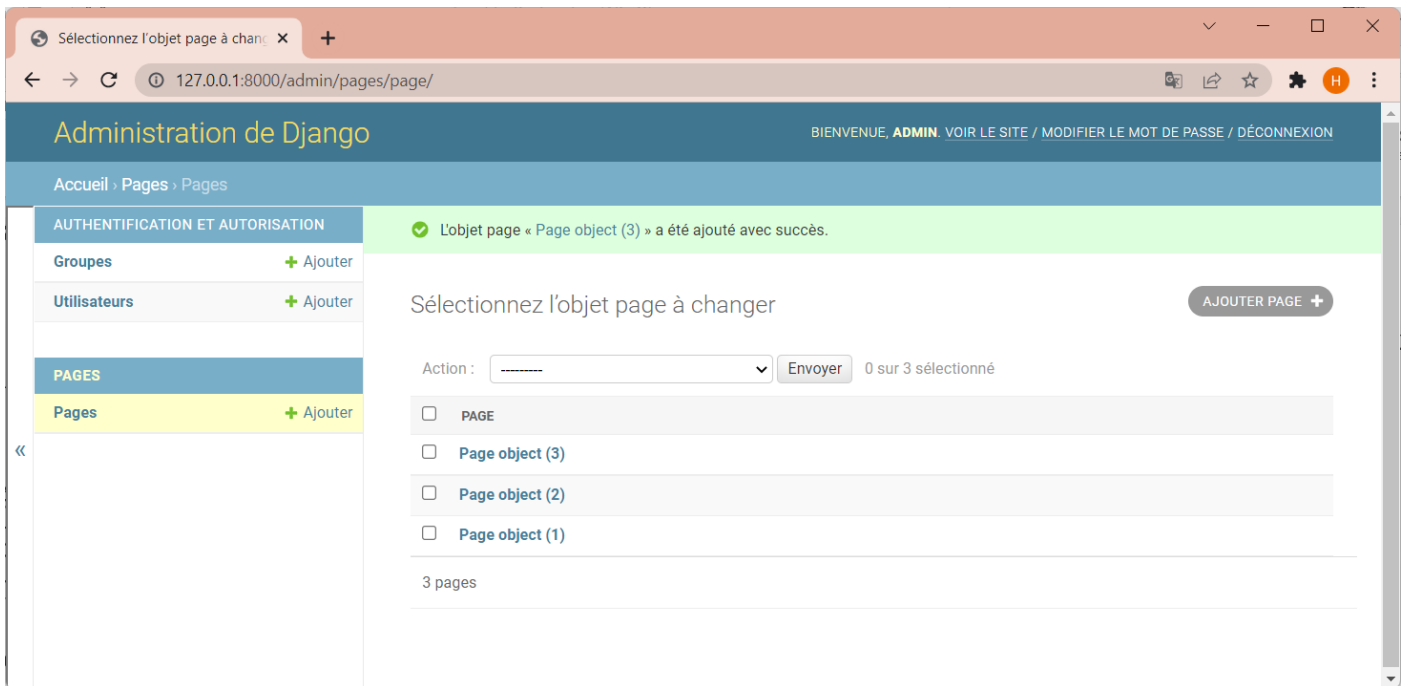
Notez également que notre formulaire d'édition utilise les noms de champs verbeux **Last Updated** et **Page Content** que nous avons entrés lorsque nous avons créé le modèle Pages.

Maintenant que vous avez entré les informations pour votre page d'accueil, cliquez sur **Save and add another** en bas à droite de votre écran. Ajoutez deux autres pages (**Last Updated** and **Page Content** peuvent être ce que vous voulez):

The screenshot shows the 'Ajout de page' form in the Django admin interface. The browser address bar shows '127.0.0.1:8000/admin/pages/page/add/'. A green success message at the top states: 'L'objet page « Page object (1) » a été ajouté avec succès. Vous pouvez ajouter un autre objet « page » ci-dessous.' The left sidebar has a menu with 'AUTHENTIFICATION ET AUTORISATION' (Groups, Users) and 'PAGES' (Pages). The form fields are: 'Title' (About us), 'Permalink' (/about), 'Last Updated' (Date: 04/03/2022, Heure: 18:57:05). The 'Page Content' field contains HTML: '<p>LP GLAASRI a été créée en 2013</p><p>L'effectif est de 36 étudiant par promotion</p><p>La formation comporte 9 modules et un Stage de Fin de Formation de 8 semaines</p>'. At the bottom are three buttons: 'Enregistrer et ajouter un nouveau', 'Enregistrer et continuer les modifications', and 'ENREGISTRER'.

The screenshot shows the 'Ajout de page' form for 'Page object (2)'. The browser address bar shows '127.0.0.1:8000/admin/pages/page/add/'. A green success message states: 'L'objet page « Page object (2) » a été ajouté avec succès. Vous pouvez ajouter un autre objet « page » ci-dessous.' The left sidebar is identical to the previous screenshot. The form fields are: 'Title' (Services), 'Permalink' (/services), 'Last Updated' (Date: 04/03/2022, Heure: 19:01:07). The 'Page Content' field contains HTML: '<h1>Services fournies par LPGLAASRI SARL</h1>Développement d'applications DesktopDéveloppement d'applications WebDéveloppement d'applications mobilesAdministration des réseauxAdministration des serveurs de Base de Données'. At the bottom are three buttons: 'Enregistrer et ajouter un nouveau', 'Enregistrer et continuer les modifications', and 'ENREGISTRER'.

Une fois que vous avez entré les informations pour votre page Services, cliquez sur **SAVE** plutôt que **Save and add another**. Cela vous amènera à la liste des *modifications de pages* (Figure ci-dessous). Vous pouvez également accéder à la liste des modifications de pages à partir de la page d'index du site admin en cliquant sur Pages à gauche du groupe ou en cliquant sur le second lien Pages dans le fil d'Ariane en haut à droite de la page d'édition.



Notez que, bien que la liste des modifications contienne trois entrées de page, elles sont appelées "Page Objet". C'est parce que nous n'avons pas dit à l'admin comment appeler nos objets Page.

C'est une réparation facile. Revenez dans le fichier `models.py` et ajouter la méthode `__str__` à la classe Page:

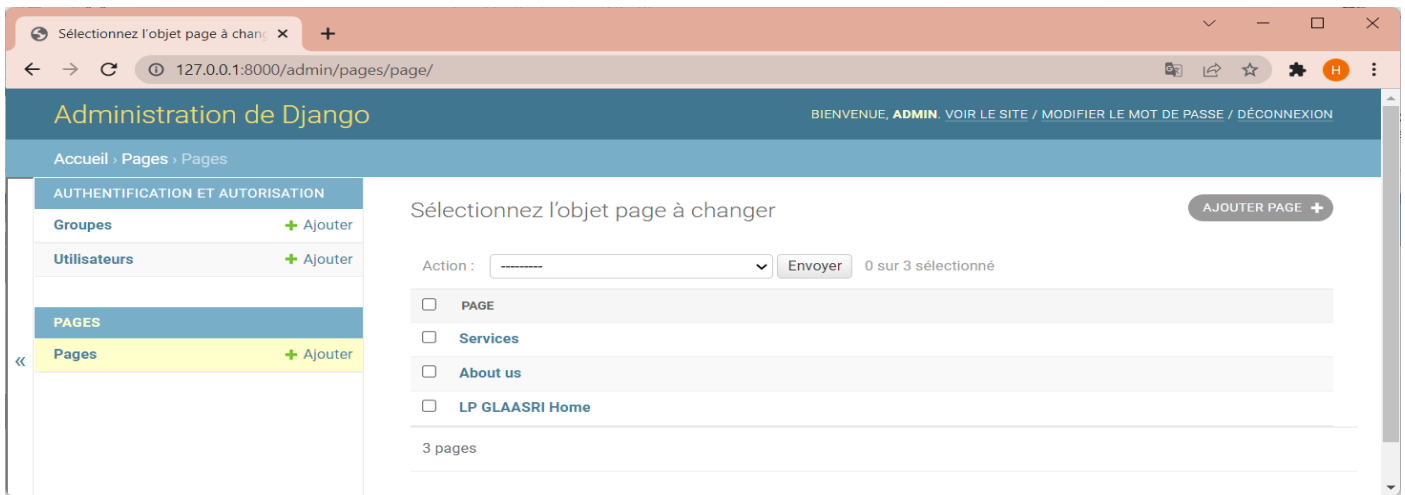
```
from django.db import models

class Page(models.Model):
    title = models.CharField(max_length=60)
    permalink = models.CharField(max_length=12, unique=True)
    update_date = models.DateTimeField(verbose_name='Last Updated')
    bodytext = models.TextField('Page Content', blank=True)

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

Alors qu'avons-nous fait ici? Dans le code encadré, nous avons créé une nouvelle méthode `__str__` qui est une fonction spéciale qui retourne une version lisible d'un objet instance de la classe Page (La valeur du champ title) à chaque fois que Python demande une représentation sous forme de chaîne de l'objet Page (ce que fait l'admin). S'il n'y a pas de méthode `__str__`, Python renvoie le type d'objet, d'où « Page Objet »

Dans notre `models.py` modifié, nous retournons simplement le titre de la page. Actualisez l'écran d'admin et vous devriez voir une liste de modifications un peu plus utile pour nos pages (Figure ci-dessous).



Il y a une dernière chose que nous voulons faire avec l'affichage de la liste des pages. Alors que nous avons maintenant les titres de la page, il reste encore du travail à faire pour rendre l'affichage de la liste plus utile:

1. Nous voulons voir quand chaque page a été mise à jour pour suivre les changements sur notre site;
2. Nous voulons afficher les titres des pages dans l'ordre alphabétique pour les rendre plus faciles à parcourir; et
3. Une fois qu'il y a plusieurs pages, nous voulons un moyen pratique de rechercher une page particulière.

Dans Django, ces changements sont très faciles à faire - vous ajoutez simplement une nouvelle classe à votre fichier `admin.py`:

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

class PageAdmin(admin.ModelAdmin):
    list_display = ('title', 'update_date')
    ordering = ('title',)
    search_fields = ('title',)

admin.site.register(Page, PageAdmin)
```

Remarque : Il est possible d'utiliser juste avant la classe le décorateur : `@admin.register(Page)` et commenter la ligne `admin.site.register(Page, PageAdmin)`

Examinons de près la nouvelle classe **PageAdmin**:

- La nouvelle classe **PageAdmin** hérite de la classe **admin.ModelAdmin** de Django.
- Le champ **list_display** indique à l'admin de Django quels champs de modèle afficher dans la liste des pages (c'est-à-dire, table colonnes). Ici, nous plaçons **list_display** dans un tuple contenant les champs **title** et **update_date**.
- Le champ **ordering** est un tuple qui indique à l'admin de Django quel champ utiliser pour trier la liste. Remarquez, comme le tuple n'a qu'un seul élément (singleton), il doit avoir une virgule à la fin.
- Le champ **search_fields** qui est un tuple indique à l'admin de Django quels champs devraient être recherchés lors de l'utilisation de la barre de recherche dans l'admin modèle. Comme **ordering**, **search_fields** est aussi un singleton, alors n'oubliez pas la virgule!

Si vous actualisez votre navigateur, l'administrateur de votre page doit ressembler à la figure ci-dessous.

Sélectionnez l'objet page à changer x +

127.0.0.1:8000/admin/pages/page/

Administration de Django

BIENVENUE, **ADMIN** VOIR LE SITE / MODIFIER LE MOT DE PASSE / DÉCONNEXION

Accueil > Pages > Pages

AUTHENTIFICATION ET AUTORISATION

Groupe + Ajouter

Utilisateurs + Ajouter

PAGES

Pages + Ajouter

Sélectionnez l'objet page à changer

AJOUTER PAGE +

Rechercher

Action : [-----] Envoyer 0 sur 3 sélectionné

<input type="checkbox"/>	TITLE	LAST UPDATED
<input type="checkbox"/>	About us	4 mars 2022 18:57
<input type="checkbox"/>	LP GLAASRI Home	4 mars 2022 18:51
<input type="checkbox"/>	Services	4 mars 2022 19:01

3 pages

Vous pouvez constater ce qui suit :

1. La liste a une nouvelle colonne affichant la date de dernière mise à jour pour chaque page;
2. Les pages sont maintenant listées par ordre alphabétique;
3. Une nouvelle barre de recherche a été ajoutée, qui permet à l'utilisateur de rechercher une page en utilisant le champ **title**.

Annexe : Référence des champs de modèle¶

Ce document contient toutes les références d'API de **Field**, y compris les détails que Django met à disposition concernant les [options de champs](#) et les [types de champs](#).

Note

Techniquement, ces modèles sont définis dans **django.db.models.fields**, mais par commodité ils sont importés dans **django.db.models** ; la convention standard est d'utiliser **from django.db import models** et de se référer aux champs avec **models.<Foo>Field**.

Options des champs¶

Les paramètres suivants sont disponibles pour tous les types de champs. Tous sont facultatifs.

null¶

Field.null¶

Si la valeur est **True**, Django stocke les valeurs vides avec **NULL** dans la base de données. La valeur par défaut est **False**.

Évitez d'utiliser **null** pour des champs textuels comme **CharField** ou **TextField**. Un champ textuel avec **null=True** implique qu'il y a deux valeurs possibles signifiant « pas de données » : **NULL** et la chaîne vide. Dans la plupart des cas, il est redondant d'avoir deux valeurs possibles indiquant qu'il n'y a pas de données. La convention dans Django est d'utiliser la chaîne vide, pas **NULL**. Une exception est lorsqu'un champ class: *CharField* possède à la fois **unique=True** et **blank=True**. Dans ce cas, **null=True** est nécessaire pour éviter des violations de contrainte unique quand plusieurs objets avec des valeurs vierges sont enregistrées.

Pour les champs textuels comme pour les champs non textuels, il est aussi nécessaire de définir **blank=True** si vous voulez autoriser les valeurs vides dans les formulaires, puisque le paramètre **null** ne se rapporte qu'au stockage dans la base de données (voir **blank**).

Note

Lors de l'utilisation du moteur de base de données Oracle, c'est toujours la valeur **NULL** qui est stockée pour signifier que la chaîne est vide, quelle que soit la valeur de cet attribut.

Si vous souhaitez accepter des valeurs **null** avec **BooleanField**, utilisez plutôt **NullBooleanField**.

blank¶

Field.blank¶

Si la valeur est **True**, le champ peut être vide. La valeur par défaut est **False**.

Notez que c'est différent de **null**. **null** est purement lié à la base de données, alors que **blank** est lié à la validation. Quand un champ possède **blank=True**, la validation de formulaire permet la saisie de valeurs vides. Quand un champ possède **blank=False**, le champ doit être obligatoirement rempli.

choices¶

Field.choices¶

Un itérable (par exemple une liste ou un tuple) composé lui-même d'itérables de tuples binaires (par ex. **[(A, B), (A, B) ...]**) représentant les choix possibles pour ce champ. Si ce paramètre est présent, le composant de formulaire par défaut sera une liste déroulante contenant ces choix au lieu de la boîte de saisie de texte standard.

Le premier élément de chaque tuple est la valeur réelle à définir pour le modèle. Le second élément est la valeur visible par l'utilisateur. Par exemple :

```
YEAR_IN_SCHOOL_CHOICES = (  
    ('FR', 'Freshman'),  
    ('SO', 'Sophomore'),  
    ('JR', 'Junior'),  
    ('SR', 'Senior'),  
)
```

Il est généralement conseillé de définir les choix à l'intérieur de la classe d'un modèle et de définir des constantes judicieusement nommées pour chaque valeur :

```
from django.db import models  
  
class Student(models.Model):  
    FRESHMAN = 'FR'  
    SOPHOMORE = 'SO'  
    JUNIOR = 'JR'  
    SENIOR = 'SR'  
    YEAR_IN_SCHOOL_CHOICES = (  
        (FRESHMAN, 'Freshman'),  
        (SOPHOMORE, 'Sophomore'),  
        (JUNIOR, 'Junior'),  
        (SENIOR, 'Senior'),  
    )  
    year_in_school = models.CharField(  
        max_length=2,  
        choices=YEAR_IN_SCHOOL_CHOICES,  
        default=FRESHMAN,  
    )  
  
    def is_upperclass(self):  
        return self.year_in_school in (self.JUNIOR, self.SENIOR)
```

Bien qu'il soit possible de définir une liste de choix en dehors de la classe d'un modèle, puis de s'y référer, le fait de définir les choix et leurs étiquettes à l'intérieur de la classe du modèle permet de conserver toutes ces informations avec la classe

qui les exploite ainsi que de faciliter la référence à ces choix (par exemple, **Student.SOPHOMORE** fonctionnera partout où le modèle **Student** a été importé).

Vous pouvez aussi arranger les choix possibles dans des groupes nommés qui pourront être utilisés à des fins d'organisation :

```
MEDIA_CHOICES = (  
    ('Audio', (  
        ('vinyl', 'Vinyl'),  
        ('cd', 'CD'),  
    )  
,  
    ('Video', (  
        ('vhs', 'VHS Tape'),  
        ('dvd', 'DVD'),  
    )  
,  
    ('unknown', 'Unknown'),  
)
```

Le premier élément de chaque tuple est le nom utilisé pour le groupe. Le second élément est un objet itérable de tuples binaires contenant la valeur et l'étiquette à afficher pour cette option. Les options groupées peuvent être combinées à des options non groupées dans une seule liste (comme l'option *unknown* dans cet exemple).

Pour chaque champ de modèle ayant l'option **choices**, Django ajoute une méthode pour obtenir l'étiquette à afficher selon la valeur actuelle du champ. Voir **get_FOO_display()** dans la documentation de l'API de base de données.

Notez que ces choix peuvent être constitués par n'importe quel objet itérable, et pas nécessairement par une liste ou un tuple. Ceci vous permet de construire dynamiquement la liste des choix. Mais si vous commencer à bidouiller l'attribut **choices** pour qu'il soit dynamique, il est probable qu'il soit plus judicieux d'utiliser une vraie table de base de données contenant une **ForeignKey**. **choices** est conçu pour les données statiques qui ne changent pas souvent, voire jamais.

À l'exception des cas où **blank=False** est défini pour le champ avec un contenu **default**, une étiquette contenant "--" sera ajoutée à la liste de sélection. Pour surcharger ce comportement, ajoutez un tuple à **choices** contenant **None**. Par exemple, **(None, 'Votre chaîne à afficher')**. Il est aussi possible d'utiliser une chaîne vide au lieu de **None** lorsque c'est raisonnable, comme pour un champ **CharField**.

db_column¶

Field.db_column¶

Le nom de la colonne à utiliser pour ce champ dans la base de données. Si cet attribut est absent, Django utilise le nom du champ.

Si le nom de la colonne dans la base de données est un mot réservé en SQL ou s'il contient des caractères non autorisés dans le nom d'une variable Python (notamment, le trait d'union), ce n'est pas un souci. Django met entre guillemets les noms de table et de colonne de manière transparente.

`db_index¶`

`Field.db_index¶`

Si **True**, une index de base de données sera créé pour ce champ.

`db_tablespace¶`

`Field.db_tablespace¶`

Le nom de l'«[espace de tables de base de données](#)» à utiliser pour l'index de ce champ, si ce champ est indexé. La valeur par défaut est le réglage **DEFAULT_INDEX_TABLESPACE** du projet, s'il est défini, ou l'attribut **db_tablespace** du modèle, s'il existe. Si le moteur de base de données ne prend pas en charge les espaces de tables, cette option est ignorée.

`default¶`

`Field.default¶`

La valeur par défaut du champ. Cela peut être une valeur ou un objet exécutable. Dans ce dernier cas, l'objet est appelé lors de chaque création d'un nouvel objet.

La valeur par défaut ne peut pas être un objet mutant (instance de modèle, **list**, **set**, etc.), car toutes les nouvelles instances du modèle utiliseraient une référence vers la même instance de cet objet. Au lieu de cela, imbriquez la valeur par défaut souhaitée dans un exécutable. Par exemple, si vous vouliez indiquer un **dict** par défaut pour un champ **JSONField**, utilisez une fonction :

```
def contact_default():  
    return {"email": "to1@example.com"}  
  
contact_info = JSONField("ContactInfo", default=contact_default)
```

Les fonctions **lambda** ne peuvent pas être utilisées comme options de champs tels que **default** car elles ne peuvent pas être [sérialisées par les migrations](#). Consultez cette dernière documentation pour d'autres mises en garde.

Pour les champs comme **ForeignKey** qui correspondent à des instances de modèles, les valeurs par défaut devraient être la valeur du champ qu'ils représentent (**pk** sauf si **to_field** est défini) et non pas des instances de modèles.

La valeur par défaut est utilisée lorsque de nouvelles instances de modèle sont créées et qu'aucune valeur n'est fournie pour le champ. Lorsque le champ est une clé primaire, la valeur par défaut est aussi utilisée si le champ est défini à **None**.

`editable¶`

`Field.editable¶`

Si **False**, le champ ne sera pas affiché dans l'administration de Django, ni dans d'éventuels formulaires basés sur **ModelForm**. Il sera aussi omis dans la phase de [validation de modèle](#). La valeur par défaut est **True**.

`error_messages¶`

`Field.error_messages¶`

Le paramètre **error_messages** permet de redéfinir les messages par défaut que le champ renvoie. Passez un dictionnaire dont les clés correspondent aux messages d'erreur que vous voulez redéfinir.

Les clés des messages d'erreur comprennent **null**, **blank**, **invalid**, **invalid_choice**, **unique** et **unique_for_date**. D'autres clés de messages d'erreur sont définies pour chaque champ de la section [types de champs](#) ci-dessous.

Ces messages d'erreur ne se propagent souvent pas aux formulaires. Voir [Considérations sur les messages d'erreur des modèles](#).

help_text¶

Field.help_text¶

Texte d'aide supplémentaire à afficher avec le composant de formulaire. Utile pour la documentation même si le champ n'est pas utilisé dans un formulaire.

Notez que le HTML contenu dans cette valeur *n'est pas* échappé dans les formulaires générés automatiquement. Cela vous permet d'inclure du HTML dans **help_text** si vous le souhaitez. Par exemple :

```
help_text="Please use the following format: <em>YYYY-MM-DD</em>."
```

Accessoirement, vous pouvez utiliser du texte brut et **django.utils.html.escape()** pour échapper n'importe quel caractère spécial HTML. Assurez-vous d'échapper tout texte d'aide qui proviendrait d'utilisateurs non fiables afin d'éviter des attaques de script inter-site.

primary_key¶

Field.primary_key¶

Si la valeur est **True**, ce champ représentera la clé primaire du modèle.

Si vous n'indiquez aucun paramètre **primary_key=True** dans les champs d'un modèle, Django ajoute automatiquement un champ **AutoField** pour constituer une clé primaire ; il n'est donc pas nécessaire de définir le paramètre **primary_key=True** pour un champ sauf si vous souhaitez modifier le comportement par défaut de clé primaire automatique. Pour en savoir plus, consultez [Champs clé primaire automatiques](#).

primary_key=True implique **null=False** et **unique=True**. Une seule clé primaire est autorisée par objet.

Le champ de clé primaire est en lecture seule. Si vous modifiez la valeur de la clé primaire d'un objet existant et que vous l'enregistrez, un nouvel objet est créé en parallèle à l'ancien.

unique¶

Field.unique¶

Si la valeur est **True**, ce champ doit être unique dans toute la table.

Cet attribut est appliqué au niveau de la base de données ainsi que dans la validation des modèles. Si vous essayez d'enregistrer une instance d'un modèle avec une valeur d'un champ **unique** dupliquée, une exception **django.db.IntegrityError** sera levée par la méthode **save()** du modèle.

Cette option est valide pour tous les types de champs, sauf pour **ManyToManyField** et **OneToOneField**.

Notez que lorsque **unique** vaut **True**, il n'est pas nécessaire de définir **db_index**, car **unique** implique qu'un index sera créé.

Changed in Django 1.11:

Dans les versions précédentes, **unique=True** ne pouvait pas être utilisé avec les champs **FileField**.

unique_for_date¶

Field.unique_for_date¶

Indiquez le nom d'un champ de type **DateField** ou **DateTimeField** comme valeur pour cette option, pour rendre obligatoire l'unicité de la valeur de ce champ en fonction de la valeur du champ date indiqué.

Par exemple, si vous avez un champ **title** qui a **unique_for_date="pub_date"**, Django n'autorise pas la saisie de deux enregistrements avec le même **title** et la même **pub_date**.

Notez que si cet attribut est défini avec un objet **DateTimeField**, seule la partie date du champ est prise en compte. De plus, lorsque **USE_TZ** vaut **True**, le contrôle est effectué dans le **fuseau horaire en cours** au moment où l'objet est enregistré.

Cette contrainte est appliquée par **Model.validate_unique()** pendant la validation des modèles, et non pas au niveau de la base de données. Si une contrainte **unique_for_date** implique des champs qui ne font pas partie d'un formulaire **ModelForm** (par exemple si l'un des champs figure dans **exclude** ou que son attribut **editable** vaut **False**), **Model.validate_unique()** passe outre la validation de la contrainte concernée.

unique_for_month

Field.unique_for_month

Comme **unique_for_date**, mais requiert que le champ soit unique en rapport au mois du champ date.

unique_for_year

Field.unique_for_year

Comme **unique_for_date** et **unique_for_month**, mais avec l'année du champ date.

verbose_name

Field.verbose_name

Un nom humainement compréhensible pour le champ. Si cet attribut n'est pas renseigné, Django le crée automatiquement en utilisant le nom d'attribut du champ, en convertissant les soulignements en espaces. Voir **noms de champs verbeux**.

validators

Field.validators

Une liste de validateurs à exécuter pour ce champ. Consultez la **documentation des validateurs** pour plus d'informations.

Inscription et obtention d'expressions de recherche

Field implémente l'**API d'inscription d'expressions de recherche**. Cette API peut être utilisée pour personnaliser les recherches possibles pour une classe de champ et la manière d'obtenir ces recherches depuis un champ.

Types de champs

AutoField

class AutoField(options)[source]**

Un champ **IntegerField** qui incrémente automatiquement sa valeur par rapport aux identifiants disponibles. Vous n'avez habituellement pas besoin d'utiliser ce champ directement ; un champ clé primaire est automatiquement ajouté au modèle s'il n'est pas explicitement défini. Voir **Champs clé primaire automatiques**.

BigAutoField

class BigAutoField(options)[source]**

Un entier 64 bits, ressemblant à un **AutoField** sauf qu'il garantit la couverture des nombres de **1** à **9223372036854775807**.

BigIntegerField

`class BigIntegerField(**options)[source]`

Un entier 64 bits, ressemblant à un **IntegerField** sauf qu'il garantit la couverture des nombres de - **9223372036854775808** à **9223372036854775807**. Le composant de formulaire par défaut de ce champ est un **TextInput**.

BinaryField

`class BinaryField(**options)[source]`

Un champ pour stocker des données binaires brutes. Il ne peut recevoir que des octets (**bytes**). Il faut savoir que ce champ possède des fonctionnalités restreintes. Par exemple, il n'est pas possible de filtrer un jeu de requête par une valeur **BinaryField**. Il n'est également pas possible d'inclure un champ **BinaryField** dans un formulaire **ModelForm**.

Usage abusif de BinaryField

Bien qu'il puisse être tentant de vouloir stocker des fichiers dans la base de données, il s'agit d'une mauvaise idée dans 99 % des cas. Ce champ n'est *pas* une solution de remplacement d'une bonne gestion des **fichiers statiques**.

BooleanField

`class BooleanField(**options)[source]`

Un champ vrai/faux.

Le composant de formulaire par défaut de ce champ est un **CheckboxInput**.

Si vous devez accepter les valeurs **null**, utilisez plutôt **NullBooleanField**.

La valeur par défaut de **BooleanField** est **None** lorsque **Field.default** n'est pas défini.

CharField

`class CharField(max_length=None, **options)[source]`

Un champ pour les chaînes de caractères, courtes ou longues.

Pour une grande quantité de texte, utilisez **TextField**.

Le composant de formulaire par défaut de ce champ est un **TextInput**.

CharField a un paramètre obligatoire supplémentaire :

CharField.max_length

La taille maximale (en caractères) du champ. Cette taille est définie au niveau de la base de données et appliquée lors de la validation par Django.

Note

Si vous écrivez une application qui doit être portable sur plusieurs moteurs de base de données, il faut savoir qu'il y a des restrictions à propos de **max_length** pour certains de ces moteurs. Référez-vous aux **notes sur les moteurs de base de données** pour plus de détails.

DateField

`class DateField(auto_now=False, auto_now_add=False, **options)[source]`

Une date, représentée en Python par une instance de **datetime.date**. Accepte quelques paramètres supplémentaires et facultatifs :

DateField.auto_now

Assigne automatiquement la valeur du champ à la date du jour lors de chaque enregistrement de l'objet. Utile pour les horodatages de « dernière modification ». Notez que c'est *toujours* la date actuelle qui est utilisée ; il ne s'agit pas seulement d'une valeur par défaut que l'on peut surcharger.

La mise à jour automatique de ce champ ne se produit que lors de l'appel à **Model.save()**. Le champ n'est pas mis à jour lors de mises à jour d'autres champs par d'autres façons comme par exemple **QuerySet.update()**, bien qu'il soit possible de spécifier explicitement une valeur pour ce champ lors d'une telle mise à jour.

DateField.auto_now_add

Assigne automatiquement la valeur du champ à la date du jour lors du premier enregistrement de l'objet. Utile pour les horodatages de date de création. Notez que c'est *toujours* la date actuelle qui est utilisée ; il ne s'agit pas seulement d'une valeur par défaut que l'on peut surcharger. Ainsi, même si vous définissez une valeur pour ce champ lors de la création de l'objet, il sera ignoré. Si vous voulez pouvoir modifier ce champ, utilisez ce qui suit au lieu de définir **auto_now_add=True**:

- Pour un champ **DateField**: **default=date.today** - de **datetime.date.today()**
- Pour un champ **DateTimeField**: **default=timezone.now** - de **django.utils.timezone.now()**

Le composant de formulaire par défaut de ce champ est un **TextInput**. L'interface d'administration ajoute un calendrier JavaScript ainsi qu'un raccourci pour « Aujourd'hui ». Contient une clé supplémentaire de message d'erreur **invalid_date**.

Les options **auto_now_add**, **auto_now** et **default** sont mutuellement exclusives. Toute combinaison de ces options produira une erreur.

Note

Avec l'implémentation actuelle, si **auto_now** ou **auto_now_add** sont activés, les paramètres de champ suivants sont définis automatiquement : **editable=False** et **blank=True**.

Note

Les options **auto_now** et **auto_now_add** utilisent toujours la date dans le **fuseau horaire par défaut** au moment de la création ou de la mise à jour. Si vos besoins sont différents, vous pouvez envisager d'utiliser votre propre fonction de valeur par défaut ou de surcharger **save()** au lieu de faire appel à **auto_now** ou **auto_now_add**. Ou encore d'utiliser un champ **DateTimeField** au lieu d'un champ **DateField** et de décider de la manière de gérer la conversion d'une heure/date vers une date au moment de son affichage.

DateTimeField

```
class DateTimeField(auto_now=False, auto_now_add=False, **options)[source]
```

Une date et une heure, représentées en Python par une instance de **datetime.datetime**. Ce champ accepte les mêmes paramètres supplémentaires que le champ **DateField**.

Le composant de formulaire par défaut de ce champ est un **TextInput**. L'interface d'administration utilise deux composants **TextInput** séparés avec des raccourcis JavaScript.

DecimalField

```
class DecimalField(max_digits=None, decimal_places=None, **options)[source]
```

Un nombre décimal de taille fixe, représenté en Python par une instance de **Decimal**. Il requiert deux paramètres **obligatoires** :

`DecimalField.max_digits`

Le nombre maximum de chiffres autorisés dans le nombre. Notez que ce nombre doit être plus grand ou égal à `decimal_places`.

`DecimalField.decimal_places`

Le nombre de décimales à stocker avec le nombre.

Par exemple, pour enregistrer un nombre jusqu'à **999** avec une précision de 2 chiffres après la virgule, vous devriez utiliser :

```
models.DecimalField(..., max_digits=5, decimal_places=2)
```

Et pour enregistrer un nombre jusqu'à un milliard environ avec une précision de 10 chiffres après la virgule :

```
models.DecimalField(..., max_digits=19, decimal_places=10)
```

Le composant de formulaire par défaut de ce champ est un **NumberInput** lorsque `localize` vaut **False**, ou **TextInput** dans le cas contraire.

Note

Pour plus d'informations sur les différences entre les classes **FloatField** et **DecimalField**, consultez [FloatField vs. DecimalField](#).

`DurationField`

`class DurationField(**options)[source]`

Un champ pour stocker des périodes de temps, représentées en Python par des objets **timedelta**. Avec PostgreSQL, le type de données utilisé est un **interval** et avec Oracle, le type de données est **INTERVAL DAY(9) TO SECOND(6)**. Sinon, c'est un grand nombre entier **bigint** de microsecondes qui est utilisé.

Note

L'arithmétique des champs **DurationField** fonctionne la plupart du temps. Cependant, pour toutes les bases de données autres que PostgreSQL, la comparaison de valeurs d'un champ **DurationField** avec l'arithmétique des instances **DateTimeField** ne fonctionne pas comme on pourrait l'espérer.

`EmailField`

`class EmailField(max_length=254, **options)[source]`

Un champ **CharField** qui vérifie que sa valeur est une adresse électronique valide. Il utilise **EmailValidator** pour valider les valeurs saisies.

`FileField`

`class FileField(upload_to=None, max_length=100, **options)[source]`

Un champ de fichier à téléverser.

Note

Le paramètre **primary_key** n'est pas pris en charge et génère une erreur s'il est utilisé.

Possède deux paramètres facultatifs :

`FileField.upload_to`

Cet attribut permet de définir le répertoire de téléversement et le nom de fichier. Il peut être défini de deux manières. Dans les deux cas, la valeur est transmise à la méthode **Storage.save()**.

Si vous indiquez une valeur textuelle, celle-ci peut contenir des chaînes de format **strftime()** qui seront remplacées par la date/heure du fichier téléversé (permettant ainsi de ne pas remplir exagérément le répertoire indiqué). Par exemple :

```
class MyModel(models.Model):  
    # file will be uploaded to MEDIA_ROOT/uploads  
    upload = models.FileField(upload_to='uploads/')  
    # or...  
    # file will be saved to MEDIA_ROOT/uploads/2015/01/30  
    upload = models.FileField(upload_to='uploads/%Y/%m/%d/')
```

Si vous utilisez le stockage par défaut **FileSystemStorage**, la valeur textuelle sera ajoutée au chemin **MEDIA_ROOT** pour déterminer l'emplacement du système de fichiers local où les fichiers téléversés seront stockés. Si vous utilisez un autre stockage, vérifiez la documentation de ce dernier pour savoir comment il traite **upload_to**.

upload_to peut aussi être un objet exécutable, comme une fonction. Il sera appelé pour obtenir le chemin de téléversement, incluant le nom du fichier. Cet objet exécutable doit accepter deux paramètres et renvoyer un chemin de type Unix (avec des barres obliques) qui sera transmis au système de stockage. Les deux paramètres sont :

Paramètre	Description
instance	Une instance du modèle où le champ FileField est défini. Plus spécifiquement, il s'agit de l'instance à laquelle le fichier actuel est joint. Dans la plupart des cas, cet objet n'aura pas encore été enregistré dans la base de données, donc en cas d'utilisation du champ AutoField par défaut, <i>il pourrait bien ne pas avoir encore de valeur pour le champ de sa clé primaire.</i>
filename	Le nom donné originellement au fichier. Sa prise en compte dans la détermination du chemin d'accès final n'est pas obligatoire.

Par exemple :

```
def user_directory_path(instance, filename):  
    # file will be uploaded to MEDIA_ROOT/user_<id>/<filename>  
    return 'user_{0}/{1}'.format(instance.user.id, filename)  
  
class MyModel(models.Model):  
    upload = models.FileField(upload_to=user_directory_path)
```

FileField.storage

Un objet de stockage, qui prend en charge l'enregistrement et la récupération des fichiers. Consultez [Gestion des fichiers](#) pour plus de détails sur la manière de fournir un tel objet.

Le composant de formulaire par défaut de ce champ est un **ClearableFileInput**.

L'utilisation d'un **FileField** ou d'un **ImageField** (voir ci-après) dans une classe de modèle se fait en quelques étapes :

1. Dans votre fichier de réglages, vous devez indiquer dans **MEDIA_ROOT** le chemin d'accès absolu vers un répertoire où Django enregistrera les fichiers téléversés (pour des raisons de performance, ces fichiers ne sont pas stockés en base de données). Indiquez dans **MEDIA_URL** l'URL publique de base correspondant à ce répertoire. Assurez vous que ce répertoire est accessible en écriture par l'utilisateur du serveur Web.
2. Ajoutez le champ **FileField** ou **ImageField** à votre modèle, en définissant l'option **upload_to** pour indiquer dans quel sous-répertoire de **MEDIA_ROOT** les fichiers doivent être téléversés.
3. Tout ce qui sera stocké dans la base de données est le chemin d'accès au fichier (relatif à **MEDIA_ROOT**). Il est très commode et courant d'utiliser l'attribut **url** offert par Django. Par exemple, considérant un **ImageField** nommé **mug_shot**, vous pouvez obtenir l'URL absolue de cette image dans un gabarit avec `{{ object.mug_shot.url }}`.

Par exemple, supposons que **MEDIA_ROOT** contient `'/home/media'` et que la valeur de **upload_to** est `'photos/%Y/%m/%d'`. La partie `'%Y/%m/%d'` de **upload_to** est du formatage `strftime()` ; `'%Y'` correspond à l'année sur quatre chiffres, `'%m'` correspond au mois sur deux chiffres et `'%d'` correspond au jour sur deux chiffres. Si vous téléversez un fichier le 15 janvier 2007, il sera enregistré dans le répertoire `/home/media/photos/2007/01/15`.

Si vous souhaitez obtenir le nom du fichier téléversé ou sa taille, vous pouvez utiliser ses attributs **name** et **size** respectivement. Pour plus d'informations sur les attributs et méthodes disponibles, consultez la référence de la classe **File** et le guide thématique [Gestion des fichiers](#).

Note

Le fichier est enregistré durant la phase d'enregistrement du modèle dans la base de données, il n'est donc pas possible de se baser sur le nom de fichier réel sur le disque tant que le modèle lui-même n'a pas été enregistré.

L'URL relative du fichier téléversé peut être obtenue en utilisant l'attribut **url**. En interne, c'est la méthode **url()** de la classe **Storage** sous-jacente qui est appelée.

Notez que chaque fois que vous avez affaire à des fichiers téléversés, vous devriez faire très attention à l'endroit où vous les enregistrez ainsi qu'à leur type, pour éviter toute faille de sécurité. *Vérifiez tous les fichiers téléversés*, ainsi vous serez sûr que ces fichiers sont bien ce qu'ils doivent être. Par exemple, si vous laissez quelqu'un téléverser aveuglément des fichiers sans les valider à destination d'un répertoire se trouvant sous la racine des documents de votre serveur Web, cette personne pourrait envoyer un script CGI ou PHP et le faire exécuter en visitant son URL sur votre site. Ne permettez pas cela.

Notez également que même l'envoi d'un fichier HTML peut poser des problèmes de sécurité équivalents aux attaques XSS ou CSRF, car ces fichiers peuvent être interprétés par un navigateur (même si le serveur n'est pas impliqué dans ce cas).

Les instances de **FileField** sont créées en tant que colonnes **varchar** dans la base de données avec une longueur par défaut maximale de 100 caractères. Comme pour d'autres champs, vous pouvez modifier la taille maximale en utilisant le paramètre **max_length**.

FileField et FieldFile

```
class FieldFile[source]
```

Lorsque vous accédez à un **FileField** d'un modèle, vous recevez une instance de **FieldFile** comme substitut d'accès au fichier sous-jacent.

L'API de **FieldFile** reflète celle de **File**, avec une différence clé : *l'objet adapté par la classe n'est pas forcément une adaptation de l'objet fichier natif de Python. C'est au contraire une adaptation du résultat de la méthode **Storage.open()**, qui peut être un objet **File** ou une implémentation d'un stockage personnalisé de l'API de **File**.

En plus de l'API héritée de **File**, comme **read()** et **write()**, **FieldFile** comprend plusieurs méthodes pouvant être utilisées pour interagir avec le fichier sous-jacent :

Avertissement

Deux méthodes de cette classe, **save()** et **delete()** enregistrent par défaut l'objet modèle du **FieldFile** associé dans la base de données.

FieldFile.name

Le nom du fichier contenant le chemin relatif depuis la racine de **Storage** du champ **FileField** associé.

FieldFile.size

Le résultat de la méthode sous-jacente **Storage.size()**.

FieldFile.url

Une propriété en lecture seule pour accéder à l'URL relative du fichier, au travers de l'appel à la méthode **url()** de la classe **Storage** sous-jacente.

FieldFile.open(mode='rb')[source]

Ouvre ou réouvre le fichier associé à cette instance dans le **mode** spécifié. Au contraire de la méthode Python standard **open()**, elle ne renvoie pas de descripteur de fichier.

Comme le fichier sous-jacent est ouvert implicitement lors de son accès, il peut être inutile d'appeler cette méthode, sauf si l'on veut réinitialiser le pointeur du fichier sous-jacent ou que l'on veut changer le **mode**.

FieldFile.close()[source]

Même comportement que la méthode **file.close()** standard de Python et ferme le fichier associé à l'instance actuelle.

FieldFile.save(name, content, save=True)[source]

Cette méthode accepte un nom de fichier et le contenu du fichier, les transmet à la classe de stockage du champ puis associe le fichier ainsi stocké avec le champ du modèle. Si vous souhaitez attribuer manuellement des données de fichier avec des instances **FileField** d'un modèle, la méthode **save()** est utilisée pour rendre persistantes ces données de fichier.

Requiert deux paramètres obligatoires : **name** correspondant au nom du fichier et **content** qui représente un objet contenant le contenu du fichier. Le paramètre facultatif **save** indique si l'instance de modèle doit être enregistrée après la modification du fichier associé à ce champ. Sa valeur par défaut est **True**.

Notez que le paramètre **content** doit être une instance de **django.core.files.File**, et non pas de l'objet **file** de Python. Vous pouvez construire une instance de **File** à partir d'un objet Python **file** existant comme ceci :

```
from django.core.files import File

# Open an existing file using Python's built-in open()

f = open('/path/to/hello.world')

myfile = File(f)
```

Ou il est aussi possible de le construire à partir d'une chaîne de caractères Python comme ceci :

```
from django.core.files.base import ContentFile  
myfile = ContentFile("hello world")
```

Pour plus d'informations, voir [Gestion des fichiers](#).

FieldFile.delete(save=True)[source]

Supprime le fichier associé à cette instance et efface tous les attributs du champ. Remarque : cette méthode ferme le fichier s'il se trouve être ouvert lorsque **delete()** est appelée.

Le paramètre facultatif **save** indique si l'instance de modèle doit être enregistrée après la suppression du fichier associé à ce champ. Sa valeur par défaut est **True**.

Notez que lorsqu'un modèle est supprimé, les fichiers liés ne sont pas supprimés. Si vous devez effacer les fichiers orphelins, c'est à vous de le faire (par exemple avec une commande de gestion personnalisée lancée manuellement ou programmée de manière périodique par un outil tel que cron).

FilePathField

class FilePathField(path=None, match=None, recursive=False, max_length=100, **options)[source]

Un **CharField** dont les choix sont limités aux noms de fichiers dans un répertoire déterminé du système de fichiers. Il possède trois paramètres spéciaux, dont le premier est **obligatoire** :

FilePathField.path

Obligatoire. Le chemin d'accès absolu vers le répertoire dont le contenu fournit la source des choix du **FilePathField**. Exemple : **"/home/images"**.

FilePathField.match

Facultatif. Une expression régulière sous forme de chaîne de caractères qui sera utilisée par **FilePathField** pour filtrer les noms de fichier. Notez que cette expression régulière sera appliquée au nom de fichier seul, et non pas à son chemin d'accès absolu. Exemple : **"foo.*\.txt\$"** correspond au fichier **foo23.txt**, mais pas à **bar.txt** ni à **foo23.png**.

FilePathField.recursive

Facultatif. Vaut **True** ou **False** (valeur par défaut). Indique si tous les sous-répertoires de **path** doivent être inclus.

FilePathField.allow_files

Facultatif. Vaut **True** (valeur par défaut) ou **False**. Indique si les fichiers de l'emplacement spécifié doivent être inclus. Il faut que l'une des deux valeurs, ce champ ou **allow_folders**, soit **True**.

FilePathField.allow_folders

Facultatif. Vaut **True** ou **False** (valeur par défaut). Indique si tous les répertoires à l'intérieur de l'emplacement spécifié doivent être inclus. Il faut que l'une des deux valeurs, ce champ ou **allow_files**, soit **True**.

Ces paramètres peuvent bien sûr être utilisés conjointement.

Le piège potentiel est que **match** s'applique au nom du fichier uniquement, et non pas au chemin d'accès absolu. Donc, dans cet exemple :

```
FilePathField(path="/home/images", match="foo.*", recursive=True)
```

... correspondra à `/home/images/foo.png` mais pas à `/home/images/foo/bar.png` car `match` s'applique uniquement au nom du fichier (`foo.png` et `bar.png`).

Les instances de **FilePathField** sont créées en tant que colonnes **varchar** dans la base de données avec une longueur par défaut maximale de 100 caractères. Comme pour d'autres champs, vous pouvez modifier la taille maximale en utilisant le paramètre `max_length`.

FloatField

```
class FloatField(**options)[source]
```

Un nombre flottant représenté en Python par une instance de **float**.

Le composant de formulaire par défaut de ce champ est un **NumberInput** lorsque `localize` vaut **False**, ou **TextInput** dans le cas contraire.

FloatField vs. DecimalField

La classe **FloatField** est parfois confondue avec la classe **DecimalField**. Même si elles représentent les deux des nombres réels, elles les stockent de manière différente. **FloatField** utilise en interne le type Python **float** alors que **DecimalField** utilise le type Python **Decimal**. Pour plus d'informations sur la différence entre ces deux types, consultez la documentation Python du module **decimal**.

ImageField

```
class ImageField(upload_to=None, height_field=None, width_field=None, max_length=100, **options)[source]
```

Hérite de tous les attributs et méthodes de **FileField**, mais valide également que l'objet téléversé est une image valide.

En complément des attributs spéciaux disponibles pour **FileField**, un champ **ImageField** possède aussi les attributs **height** et **width**.

Pour faciliter l'interrogation de ces attributs, **ImageField** possède deux attributs facultatifs supplémentaires :

ImageField.height_field

Le nom d'un champ du modèle qui sera automatiquement renseigné avec la hauteur de l'image à chaque enregistrement de l'instance du modèle.

ImageField.width_field

Le nom d'un champ du modèle qui sera automatiquement renseigné avec la largeur de l'image à chaque enregistrement de l'instance du modèle.

Nécessite la bibliothèque [Pillow](#).

Les instances de **ImageField** sont créées en tant que colonnes **varchar** dans la base de données avec une longueur par défaut maximale de 100 caractères. Comme pour d'autres champs, vous pouvez modifier la taille maximale en utilisant le paramètre `max_length`.

Le composant de formulaire par défaut de ce champ est un **ClearableFileInput**.

IntegerField

```
class IntegerField(**options)[source]
```

Un nombre entier. Les valeurs de **-2147483648** à **2147483647** sont acceptées par toutes les base de données prises en charge par Django. Le composant de formulaire par défaut de ce champ est un **NumberInput** lorsque **localize** vaut **False**, ou **TextInput** dans le cas contraire.

GenericIPAddressField

```
class GenericIPAddressField(protocol='both', unpack_ipv4=False, **options)[source]
```

Une adresse IPv4 ou IPv6 au format textuel (par ex. **192.0.2.30** ou **2a02:42fe::4**). Le composant de formulaire par défaut de ce champ est un **TextInput**.

La normalisation d'adresse IPv6 respecte la section 2.2 de la **RFC 4291#section-2.2**, y compris l'utilisation du format IPv4 suggéré dans le 3e paragraphe de cette section, comme **::ffff:192.0.2.0**. Par exemple, **2001:0::0:01** sera normalisé en **2001::1** et **::ffff:0a0a:0a0a** en **::ffff:10.10.10.10**. Tous les caractères sont convertis en minuscules.

GenericIPAddressField.protocol

Limite la validité des saisies au protocole indiqué. Les valeurs possibles sont **'both'** (les deux protocoles acceptés, valeur par défaut), **'IPv4'** ou **'IPv6'**.

GenericIPAddressField.unpack_ipv4

Décode les adresses IPv4 mappées comme **::ffff:192.0.2.1**. Si cette option est activée, cette adresse serait décodée en **192.0.2.1**. L'option est désactivée par défaut. Utilisable uniquement quand **protocol** est défini à **'both'**.

Si vous autorisez les valeurs vierges, vous devez aussi autoriser les valeurs nulles, car les valeurs vierges sont stockées par une valeur nulle.

NullBooleanField

```
class NullBooleanField(**options)[source]
```

Comme un champ **BooleanField**, mais autorise la valeur **NULL** comme choix possible. Préférez ce champ à un **BooleanField** avec l'option **null=True**. Le composant de formulaire par défaut de ce champ est un **NullBooleanSelect**.

PositiveIntegerField

```
class PositiveIntegerField(**options)[source]
```

Comme un champ **IntegerField**, mais doit être un entier positif ou zéro (**0**). Les valeurs de **0** à **2147483647** sont acceptées par toutes les base de données prises en charge par Django. La valeur **0** est acceptée pour des raisons de rétrocompatibilité.

PositiveSmallIntegerField

```
class PositiveSmallIntegerField(**options)[source]
```

Comme un **PositiveIntegerField**, mais n'autorise que des valeurs plus petites qu'un certain plafond (dépendant du moteur de base de données). Toutes les valeurs de **0** à **32767** sont acceptées par toutes les bases de données prises en charge officiellement par Django.

SlugField

```
class SlugField(max_length=50, **options)[source]
```

Slug est un terme anglophone de journalisme. Un slug est une brève étiquette d'un contenu, composée uniquement de lettres, de chiffres, de soulignements ou de tirets. Ils sont généralement utilisés dans les URL.

Comme pour le champ **CharField**, vous pouvez indiquer le paramètre **max_length** (lisez les notes à propos de la portabilité entre bases de données de **max_length**). Si **max_length** n'est pas précisé, Django utilise la valeur 50 par défaut.

L'option **Field.db_index** est implicitement égale à **True**.

Il est souvent pratique de pouvoir automatiquement renseigner un **SlugField** en se basant sur la valeur d'un autre attribut. Vous pouvez le faire dans l'interface d'administration en utilisant **prepopulated_fields**.

SlugField.allow_unicode

Si la valeur est **True**, le champ accepte des lettres Unicode en plus des lettres ASCII de base. La valeur par défaut est **False**.

SmallIntegerField

class SmallIntegerField(options)[source]**

Comme un **IntegerField**, mais n'autorise que des valeurs plus petites qu'un certain plafond (dépendant du moteur de base de données). Toutes les valeurs de **-32768** jusqu'à **32767** sont acceptées par toutes les bases de données prises en charge officiellement par Django.

TextField

class TextField(options)[source]**

Un champ de texte long. Le composant de formulaire par défaut de ce champ est un **Textarea**.

Si vous indiquez un attribut **max_length**, celui-ci se répercute sur le composant **Textarea** du champ de formulaire généré automatiquement. Cependant, la limite n'est pas imposée au niveau du modèle, ni de la base de données. Pour cela, utilisez plutôt un champ **CharField**.

TimeField

class TimeField(auto_now=False, auto_now_add=False, **options)[source]

Une heure, représentée en Python par une instance de **datetime.time**. Ce champ accepte les mêmes options d'auto-complétion qu'un champ **DateField**.

Le composant de formulaire par défaut de ce champ est un **TextInput**. L'interface d'administration ajoute des raccourcis JavaScript.

URLField

class URLField(max_length=200, **options)[source]

Un champ **CharField** pour les URL.

Le composant de formulaire par défaut de ce champ est un **TextInput**.

Comme pour les autres sous-classes de **CharField**, **URLField** accepte le paramètre facultatif **max_length**. Si vous ne renseignez pas la valeur de **max_length**, elle prend la valeur 200 par défaut.

UUIDField

class UUIDField(options)[source]**

Un champ pour stocker des identifiants universels uniques (UUID). Utilisez la classe Python **UUID**. Avec PostgreSQL, le type de données **uuid** est employé, sinon c'est un type **char(32)**.

Les identifiants universels uniques sont une bonne alternative aux champs **AutoField** pour les clés primaires **primary_key**. La base de données ne produit pas de UUID à votre place, c'est pourquoi il est recommandé d'utiliser **default**:

```
import uuid

from django.db import models

class MyUUIDModel(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    # other fields
```

Notez que nous passons bien un objet exécutable (sans les parenthèses) à **default**, et non pas une instance de **UUID**.

Champs pour les relations¶

Django définit aussi un ensemble de champs représentant les relations.

ForeignKey¶

```
class ForeignKey(to, on_delete, **options)[source]¶
```

Une relation plusieurs-à-un. Exige deux paramètres : la classe à laquelle le modèle est lié et l'option **on_delete**.

Pour créer une relation récursive (un objet avec une relation plusieurs-à-un vers lui-même), utilisez **models.ForeignKey('self', on_delete=models.CASCADE)**.

Si vous avez besoin de créer une relation vers un modèle qui n'a pas encore été défini, vous pouvez utiliser le nom de ce modèle, à la place de l'objet modèle lui-même :

```
from django.db import models

class Car(models.Model):
    manufacturer = models.ForeignKey(
        'Manufacturer',
        on_delete=models.CASCADE,
    )
    # ...

class Manufacturer(models.Model):
    # ...

    pass
```

Les relations définies de cette façon pour des **modèles abstraits** sont résolues au moment où le modèle est hérité par un modèle concret et ne sont pas en lien avec l'attribut **app_label** du modèle abstrait :

products/models.py

```
from django.db import models

class AbstractCar(models.Model):
    manufacturer = models.ForeignKey('Manufacturer', on_delete=models.CASCADE)

    class Meta:
        abstract = True
```

production/models.py

```
from django.db import models
from products.models import AbstractCar

class Manufacturer(models.Model):
    pass

class Car(AbstractCar):
    pass

# Car.manufacturer will point to `production.Manufacturer` here.
```

Pour faire référence à un modèle d'une autre application, vous pouvez explicitement indiquer un modèle avec le chemin vers son application. Par exemple, si le modèle **Manufacturer** précédent est défini dans une autre application appelée **production**, il faudrait utiliser :

```
class Car(models.Model):
    manufacturer = models.ForeignKey(
        'production.Manufacturer',
        on_delete=models.CASCADE,
    )
```

Cette méthode de référence, appelée une relation différée, peut être utile pour la résolution des dépendances d'importation circulaires entre deux applications.

Un index de base de données est automatiquement créé pour les champs **ForeignKey**. Vous pouvez le désactiver en définissant **db_index** à **False**. Il est parfois souhaitable d'éviter la création inutile d'un index si la clé étrangère est créée par cohérence plutôt que pour les jointures ou si vous allez créer un autre index tel qu'un index partiel ou un index sur plusieurs colonnes.

Représentation en base de données¶

En arrière-plan, Django ajoute `"_id"` au nom du champ pour créer ses noms de colonnes de base de données. Dans l'exemple précédent, la table de base de données pour le modèle `Car` aura une colonne `manufacturer_id` (vous pouvez changer ceci explicitement en définissant `db_column`). Cependant, votre code ne devrait jamais avoir affaire directement aux noms des colonnes de base de données, à moins d'écrire soi-même du SQL personnalisé. Vous utiliserez toujours les noms des champs des objets de vos modèles.

Paramètres

Le champ `ForeignKey` accepte d'autres paramètres qui définissent les détails du fonctionnement de la relation.

`ForeignKey.on_delete`

Lorsqu'un objet référencé par une `ForeignKey` est supprimé, Django simule le comportement de la contrainte SQL définie par le paramètre `on_delete`. Par exemple, si le champ `ForeignKey` peut contenir la valeur `null` et que vous vouliez qu'il prenne cette valeur lorsque l'objet référencé est supprimé :

```
user = models.ForeignKey(
    User,
    models.SET_NULL,
    blank=True,
    null=True,
)
```

Les valeurs possibles de `on_delete` sont énumérées dans `django.db.models`:

- `CASCADE`[\[source\]](#)

Supprime en cascade. Django simule le comportement de la contrainte SQL `ON DELETE CASCADE` et supprime aussi l'objet contenant la clé `ForeignKey`.

- `PROTECT`[\[source\]](#)

Empêche la suppression de l'objet référencé en levant une exception `ProtectedError`, une sous-classe de `django.db.IntegrityError`.

- `SET_NULL`[\[source\]](#)

Place la valeur nulle dans `ForeignKey` ; ce n'est possible que si le paramètre `null` vaut `True`.

- `SET_DEFAULT`[\[source\]](#)

Définit la valeur de `ForeignKey` à sa valeur par défaut ; il faut évidemment qu'une valeur par défaut existe pour le champ `ForeignKey`.

- `SET()`[\[source\]](#)

Définit la valeur de `ForeignKey` à celle qui est transmise à `SET()`, ou, si un objet exécutable est transmis, au résultat de l'appel à cet objet. Dans la plupart des cas, il sera nécessaire de transmettre un objet exécutable pour éviter de devoir lancer des requêtes au moment de l'importation du fichier `models.py` :

```
from django.conf import settings
from django.contrib.auth import get_user_model
from django.db import models
```

```
def get_sentinel_user():
    return get_user_model().objects.get_or_create(username='deleted')[0]

class MyModel(models.Model):
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.SET(get_sentinel_user),
    )
```

- **DO_NOTHING**[\[source\]](#)[¶](#)

Ne fait rien. Si le moteur de base de données assure l'intégrité référentielle, ceci génère une exception **IntegrityError** sauf si vous ajoutez manuellement une contrainte SQL **ON DELETE** au champ de base de données.

ForeignKey.limit_choices_to[¶](#)

Définit une limite des choix disponibles pour ce champ lorsqu'il est affiché par un **ModelForm** ou dans le site d'administration (par défaut, tous les objets du jeu de requête sont offerts comme choix). Cette limite peut être un dictionnaire, un objet **Q** ou un objet exécutable renvoyant un dictionnaire ou un objet **Q**.

Par exemple :

```
staff_member = models.ForeignKey(
    User,
    on_delete=models.CASCADE,
    limit_choices_to={'is_staff': True},
)
```

fait que le champ correspondant du formulaire **ModelForm** ne présente que la liste des **Users** ayant **is_staff=True**. Cela peut être utile dans l'administration de Django.

La forme « objet exécutable » peut être pratique par exemple quand on l'utilise avec le module Python **datetime** pour limiter les choix possibles en fonction d'intervalles de temps. Par exemple :

```
def limit_pub_date_choices():
    return {'pub_date__lte': datetime.date.utcnow()}

limit_choices_to = limit_pub_date_choices
```

Si **limit_choices_to** est ou renvoie un **objet Q**, ce qui est utile pour des **requêtes complexes**, il n'aura d'effet sur les choix disponibles dans l'interface d'administration que si le champ ne figure pas dans la propriété **raw_id_fields** du formulaire **ModelAdmin** du modèle.

Note

Si un objet exécutable est utilisé pour **limit_choices_to**, il sera appelé lors de chaque instanciation de formulaire. Il se peut également qu'il soit appelé lors de la validation d'un modèle, par exemple par une commande d'administration ou par le site d'administration. Ce dernier construit les jeux de requête pour valider les saisies de formulaires dans divers cas limites plusieurs fois, il faut donc tenir compte de l'éventualité de plusieurs appels de l'objet exécutable.

ForeignKey.related_name

Le nom à utiliser pour la relation inverse depuis l'objet lié vers celui-ci. Il s'agit aussi de la valeur par défaut de **related_query_name** (le nom à utiliser comme nom de filtre inverse à partir du modèle cible). Voir la [documentation des objets liés](#) pour une explication complète et des exemples. Notez que vous devez définir cette valeur quand vous définissez une relation pour un **modèle abstrait** ; et quand vous le faites, une **syntaxe particulière** est autorisée.

Si vous préférez que Django ne crée pas de relation inverse, définissez **related_name** à '+' ou terminez ce nom avec '+'. Par exemple, ceci assure que le modèle **User** n'aura pas de relation inverse à ce modèle :

```
user = models.ForeignKey(
    User,
    on_delete=models.CASCADE,
    related_name='+',
)
```

ForeignKey.related_query_name

Le nom à utiliser comme nom de filtre inverse à partir du modèle cible. La valeur par défaut est identique à **related_name** ou à **default_related_name** si l'un de ceux-ci est défini, sinon elle correspond au nom du modèle :

```
# Declare the ForeignKey with related_query_name

class Tag(models.Model):
    article = models.ForeignKey(
        Article,
        on_delete=models.CASCADE,
        related_name="tags",
        related_query_name="tag",
    )
    name = models.CharField(max_length=255)

# That's now the name of the reverse filter

Article.objects.filter(tag__name="important")
```

Comme **related_name**, **related_query_name** prend en charge les étiquettes d'application et l'interpolation de classe via **une syntaxe particulière**.

ForeignKey.to_field

Le champ sur lequel se fait la relation d'objet. Par défaut, Django utilise la clé primaire de l'objet lié. Si vous faites référence à un autre champ, ce champ doit avoir **unique=True**.

ForeignKey.db_constraint¶

Contrôle si une contrainte doit être créée en base de données pour cette clé étrangère. La valeur par défaut est **True**, et c'est généralement le bon choix. En définissant la valeur **False**, l'impact sur l'intégrité des données peut être très négatif. Ceci dit, certains scénarios peuvent justifier ce réglage :

- Vous possédez déjà des données qui ne sont pas valides.
- Vous partitionnez votre base de données.

Lorsque ce paramètre est défini à **False**, l'accès à un objet lié qui n'existe pas génère son exception **DoesNotExist**.

ForeignKey.swappable¶

Contrôle la réaction du système de migrations si cette clé **ForeignKey** pointe vers un modèle permutable. Quand elle vaut **True** (valeur par défaut), si la clé **ForeignKey** pointe vers un modèle qui correspond à la valeur actuelle de **settings.AUTH_USER_MODEL** (ou un autre réglage de modèle permutable), la relation est stockée dans la migration en utilisant une référence au réglage et non pas directement au modèle.

Cette valeur ne devrait être changée à **False** que si l'on est certain que le modèle doit toujours pointer vers le modèle permuté, par exemple quand il s'agit d'un modèle de profil conçu spécialement pour le modèle d'utilisateur personnalisé.

En définissant cette propriété à **False**, cela ne signifie pas que vous pouvez faire référence à un modèle permutable même quand il a été permuté, **False** signifie simplement que les migrations effectuées avec cette clé **ForeignKey** vont toujours faire référence au modèle exact que vous avez défini (si un utilisateur essaie de faire fonctionner le modèle avec un modèle d'utilisateur pour lequel il n'a pas été prévu, les choses vont mal se passer).

En cas de doute, laissez cette propriété à sa valeur par défaut, **True**.

ManyToManyField¶

class ManyToManyField(to, **options)[source]¶

Une relation plusieurs-à-plusieurs. Exige un paramètre positionnel : la classe à laquelle le modèle est lié, qui fonctionne exactement de la même manière que pour **ForeignKey**, y compris les relations **récurives** et **différées**.

Les objets liés peuvent être ajoutés, supprimés ou créés avec le gestionnaire **RelatedManager** du champ.

Représentation en base de données¶

En arrière-plan, Django crée une table de jointure intermédiaire pour représenter la relation plusieurs-à-plusieurs. Par défaut, le nom de cette table est généré en utilisant le nom du champ plusieurs-à-plusieurs et le nom de la table de son modèle. Étant donné que certaines bases de données ne gèrent pas des noms de tables au-delà d'une certaine taille, ce nom de table sera automatiquement tronqué à 64 caractères et un hachage unique sera utilisé. Cela signifie que vous pourriez voir des noms de table comme **author_books_9cdf4**; ceci est tout à fait normal. Vous pouvez manuellement attribuer un nom à la table de jointure en utilisant l'option **db_table**.

Paramètres¶

ManyToManyField accepte un ensemble de paramètres supplémentaires, tous facultatifs, qui contrôlent le fonctionnement de la relation.

ManyToManyField.related_name¶

Comme pour `ForeignKey.related_name`.

`ManyToManyField.related_query_name`

Comme pour `ForeignKey.related_query_name`.

`ManyToManyField.limit_choices_to`

Comme pour `ForeignKey.limit_choices_to`.

`limit_choices_to` n'a aucun effet sur un `ManyToManyField` avec une table intermédiaire personnalisée définie avec le paramètre `through`.

`ManyToManyField.symmetrical`

Utilisé uniquement dans la définition d'un `ManyToManyField` sur lui-même. Considérons le modèle suivant :

```
from django.db import models

class Person(models.Model):

    friends = models.ManyToManyField("self")
```

Quand Django parcourt ce modèle, il constate que le modèle a un `ManyToManyField` sur lui-même, et par conséquence, il n'ajoute pas d'attribut `person_set` à la classe `Person`. Il considère que le champ `ManyToManyField` est symétrique, c'est-à-dire : si je suis votre ami, vous êtes vous aussi mon ami.

Si vous ne voulez pas de symétrie dans une relation plusieurs-à-plusieurs avec `self`, définissez le paramètre `symmetrical` à `False`. Cela forcera Django à ajouter un descripteur pour la relation inverse, autorisant la relation `ManyToManyField` à ne pas être symétrique.

`ManyToManyField.through`

Django génère automatiquement une table pour gérer les relations plusieurs-à-plusieurs. Cependant, si vous désirez spécifier manuellement la table intermédiaire, vous pouvez utiliser l'option `through` pour indiquer le modèle Django qui représente cette table intermédiaire.

L'usage le plus fréquent de cette option est lorsque vous souhaitez associer des données supplémentaires à une relation plusieurs-à-plusieurs.

Si vous ne spécifiez pas de modèle intermédiaire `through` de façon explicite, il est tout de même possible d'utiliser une classe de modèle `through` implicite pour directement accéder à la table créée pour stocker les associations. Elle comporte trois champs pour faire le lien entre les modèles.

Si les modèles cible et source diffèrent, les champs suivants sont générés :

- `id`: la clé primaire de la relation.
- `<modèle_conteneur>_id`: le champ `id` du modèle qui déclare le champ `ManyToManyField`.
- `<autre_modèle>_id`: le champ `id` du modèle vers lequel pointe le champ `ManyToManyField`.

Si le champ `ManyToManyField` possède le même modèle comme cible et source, les champs suivants sont générés :

- `id`: la clé primaire de la relation.
- `from_<model>_id`: l'identifiant de l'instance qui pointe vers le modèle (c'est-à-dire l'instance source).

- **to_<model>_id**: l'identifiant de l'instance vers laquelle pointe la relation (c'est-à-dire l'instance de modèle cible).

Cette classe peut être utilisée pour interroger des lignes associées à une instance de modèle donnée, tout comme pour un modèle normal.

ManyToManyField.through_fields

Utilisé seulement quand un modèle intermédiaire personnalisé a été défini. Django détermine normalement de manière automatique les champs du modèle intermédiaire utilisés pour établir une relation plusieurs-à-plusieurs. Cependant, considérez les modèles suivants :

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=50)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(
        Person,
        through='Membership',
        through_fields=('group', 'person'),
    )

class Membership(models.Model):
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    inviter = models.ForeignKey(
        Person,
        on_delete=models.CASCADE,
        related_name="membership_invites",
    )
    invite_reason = models.CharField(max_length=64)
```

Membership possède *deux* clés étrangères vers **Person** (**person** et **inviter**), ce qui rend la relation ambiguë et Django ne peut pas savoir laquelle des deux clés il doit utiliser. Dans ce cas, vous devez définir explicitement la clé étrangère que Django doit utiliser au moyen de **through_fields**, comme dans l'exemple ci-dessus.

through_fields accepte un tuple binaire ('champ1', 'champ2'), où **champ1** est le nom de la clé étrangère vers le modèle qui définit la relation **ManyToManyField** (*group* dans ce cas) et **champ2** le nom de la clé étrangère vers le modèle cible (**person** dans ce cas).

Lorsque vous avez plus d'une clé étrangère d'un modèle intermédiaire vers l'un (ou même les deux) des modèles participant à une relation plusieurs-à-plusieurs, vous devez définir **through_fields**. Cela s'applique aussi aux **relations récursives** lorsqu'un modèle intermédiaire est utilisé et qu'il y a plus de deux clés étrangères vers le modèle, ou que vous souhaitez définir explicitement laquelle des deux Django doit utiliser.

Les relations récursives utilisant un modèle intermédiaire sont toujours définies comme non symétriques, c'est-à-dire avec **symmetrical=False**. De ce fait, un concept de « source » et de « cible » entre en jeu. Dans ce cas, **champ1** sera considéré comme la « source » de la relation et **champ2** comme la « cible ».

ManyToManyField.db_table

Le nom de la table à créer pour enregistrer les données de la relation plusieurs-à-plusieurs. Si ce paramètre n'est pas renseigné, Django génère un nom par défaut basé sur le nom de la table du modèle définissant la relation et le nom du champ lui-même.

ManyToManyField.db_constraint

Contrôle si des contraintes doivent être créées en base de données pour les clés étrangères de la table intermédiaire. La valeur par défaut est **True**, et c'est généralement le bon choix. En définissant la valeur **False**, l'impact sur l'intégrité des données peut être très négatif. Ceci dit, certains scénarios peuvent justifier ce réglage :

- Vous possédez déjà des données qui ne sont pas valides.
- Vous partitionnez votre base de données.

Il est faux de passer à la fois les paramètres **db_constraint** et **through**.

ManyToManyField.swappable

Contrôle la réaction du système de migrations si ce champ **ManyToManyField** pointe vers un modèle permutable. Quand elle vaut **True** (valeur par défaut), si le champ **ManyToManyField** pointe vers un modèle qui correspond à la valeur actuelle de **settings.AUTH_USER_MODEL** (ou un autre réglage de modèle permutable), la relation est stockée dans la migration en utilisant une référence au réglage et non pas directement au modèle.

Cette valeur ne devrait être changée à **False** que si l'on est certain que le modèle doit toujours pointer vers le modèle permuté, par exemple quand il s'agit d'un modèle de profil conçu spécialement pour le modèle d'utilisateur personnalisé.

En cas de doute, laissez cette propriété à sa valeur par défaut, **True**.

ManyToManyField ne prend pas en charge le paramètre **validators**.

null n'a aucun effet puisqu'il n'est pas possible d'exiger une relation au niveau de la base de données.

OneToOneField

class OneToOneField(to, on_delete, parent_link=False, **options)[source]

Une relation un-à-un. Conceptuellement, ceci est similaire à un champ **ForeignKey** avec l'attribut **unique=True**, mais le côté « inverse » de la relation renvoie directement un objet unique.

Ceci est très utile comme clé primaire d'un modèle qui « étend » un autre modèle d'une certaine manière ; par exemple, l'**Héritage multi-table** est implémenté en ajoutant une relation un-à-un implicite depuis le modèle fils vers le modèle parent.

Un paramètre positionnel est obligatoire : la classe à laquelle le modèle est lié. Ceci fonctionne exactement de la même manière que pour **ForeignKey**, y compris toutes les options concernant les relations **récurives** et **différées**.

Si vous ne renseignez pas le paramètre **related_name** du champ **OneToOneField**, Django utilise le nom du modèle actuel en minuscules comme valeur par défaut.

Avec l'exemple suivant :

```
from django.conf import settings
from django.db import models

class MySpecialUser(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
    )
    supervisor = models.OneToOneField(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name='supervisor_of',
    )
```

le modèle **User** résultant possédera les attributs suivants :

```
>>> user = User.objects.get(pk=1)
>>> hasattr(user, 'myspecialuser')
True
>>> hasattr(user, 'supervisor_of')
True
```

Une exception **DoesNotExist** est générée lors de l'accès à la relation inverse si aucune ligne correspondante n'existe dans la table liée. Par exemple, si un utilisateur ne possède pas de superviseur dans une instance de modèle **MySpecialUser**:

```
>>> user.supervisor_of
Traceback (most recent call last):
...
DoesNotExist: User matching query does not exist.
```

De plus, **OneToOneField** accepte tous les paramètres supplémentaires acceptés par **ForeignKey**, plus un paramètre supplémentaire :

OneToOneField.parent_link¶

Si ce paramètre vaut **True** et qu'il est utilisé dans un modèle qui hérite d'un autre **modèle concret**, cela indique que ce champ devrait être utilisé comme lien vers la classe parente, à la place d'un **OneToOneField** supplémentaire qui devrait normalement être implicitement créé par l'héritage.

Voir **Relations un-à-un** pour des exemples d'utilisation du champ **OneToOneField**.

Référence d'API des champs¶

class Field[source]¶

Field est une classe abstraite qui représente une colonne de table de base de données. Django utilise des champs pour créer une table de base de données (**db_type()**), pour faire correspondre des types Python à une base de données (**get_prep_value()**) et inversement (**from_db_value()**).

Un champ est donc une pièce essentielle dans différentes API de Django, notamment pour les **modèles** et les **jeux de requête**.

Dans les modèles, un champ est instancié comme attribut de classe et représente une colonne de table de base de données, voir **Modèles**. Il possède des attributs comme **null** et **unique**, ainsi que des méthodes que Django utilise pour faire correspondre la valeur du champ à des valeurs spécifiques à la base de données.

Un champ **Field** est une sous-classe de **RegisterLookupMixin**, ce qui fait qu'à la fois **Transform** et **Lookup** peuvent y être inscrits pour être exploités dans les requêtes **QuerySet** (par ex. **nom_champ__exact="foo"**). Toutes les :ref:expressions de requête intégrées <field-lookups>` sont inscrites par défaut.

Tous les types de champs fournis par Django, tels que **CharField**, sont des implémentations dérivées de **Field**. Si vous avez besoin d'un nouveau type de champ, vous pouvez soit hériter d'un des champs de Django ou écrire une classe **Field** à partir de zéro. Dans tous les cas, consultez **Écriture de champs de modèles personnalisés**.

description¶

Une description verbeuse du champ, par exemple pour l'application **django.contrib.admindocs**.

La description peut prendre la forme :

```
description = _("String (up to %(max_length)s)")
```

où les paramètres sont interpolés à partir du **__dict__** du champ.

Pour faire correspondre un champ **Field** à un type particulier d'une base de données, Django expose plusieurs méthodes :

get_internal_type()[source]¶

Renvoie une chaîne nommant ce champ à l'usage du moteur de base de données spécifique. Par défaut, c'est le nom de la classe qui est renvoyé.

Voir **Émulation de types de champs intégrés** pour son utilisation dans des types de champs personnalisés.

db_type(connection)[source]¶

Renvoie le type de donnée de la colonne de la base de données de la classe **Field**, prenant en compte l'objet **connection**.

Voir **Types de base de données personnalisés** pour son utilisation dans des types de champs personnalisés.

`rel_db_type(connection)[source]`[¶](#)

Renvoie le type de donnée de la colonne de la base de données, tel que **ForeignKey** et **OneToOneField** qui pointent vers **Field**, prenant en compte l'objet **connection**.

Voir **Types de base de données personnalisés** pour son utilisation dans des types de champs personnalisés.

Il existe trois situations principales où Django a besoin de faire interagir le moteur de base de données et les champs :

- lorsqu'il interroge la base de données (valeur Python -> valeur pour le moteur de base de données)
- lorsqu'il charge des données à partir de la base de données (valeur du moteur de base de données -> valeur Python)
- lorsqu'il enregistre vers la base de données (valeur Python -> valeur pour le moteur de base de données)

Lors d'une interrogation, `get_db_prep_value()` et `get_prep_value()` sont utilisés :

`get_prep_value(value)[source]`[¶](#)

value est la valeur actuelle de l'attribut de modèle, et la méthode doit renvoyer les données dans un format préparé spécialement pour être utilisé comme paramètre de la requête.

Voir **Conversion d'objets Python en valeurs de requête** pour son utilisation.

`get_db_prep_value(value, connection, prepared=False)[source]`[¶](#)

Convertit **value** vers une valeur spécifique au moteur de base de données. Par défaut, il renvoie **value** si **prepared=True** et `get_prep_value()` sinon.

Voir **Conversion de valeurs de requête en valeurs de base de données** pour son utilisation.

Lors du chargement de données, `from_db_value()` est utilisé :

`from_db_value(value, expression, connection)`[¶](#)

Convertit la valeur renvoyée par la base de données en un objet Python. C'est l'inverse de `get_prep_value()`.

Cette méthode n'est pas utilisée pour la plupart des champs intégrés, car le moteur de base de données renvoie déjà le bon type Python, ou le moteur se charge lui-même de la conversion.

Voir **Conversion de valeurs en objets Python** pour son utilisation.

Note

Pour des raisons de performance, `from_db_value` n'est pas implémentée comme opération blanche pour les champs qui n'en ont pas besoin (tous les champs Django). Par conséquent, vous ne pouvez pas appeler **super** dans votre définition.

Lors de l'enregistrement, `pre_save()` et `get_db_prep_save()` sont utilisés :

`get_db_prep_save(value, connection)[source]`[¶](#)

Comme pour `get_db_prep_value()`, mais appelé lorsque la valeur du champ doit être *enregistrée* dans la base de données. Renvoie par défaut `get_db_prep_value()`.

`pre_save(model_instance, add)[source]`[¶](#)

Méthode appelée avant `get_db_prep_save()` pour préparer la valeur avant d'être enregistrée (par ex. pour `DateTimeField.auto_now`).

`model_instance` est l'instance à laquelle appartient ce champ et `add` indique si l'instance est enregistrée dans la base de données pour la première fois.

Cette méthode doit renvoyer la valeur de l'attribut correspondant de `model_instance` pour ce champ. Le nom d'attribut est dans `self.attname` (défini par `Field`).

Voir [Pré-traitement des valeurs avant enregistrement](#) pour son utilisation.

Les champs reçoivent souvent leur valeur dans un type différent, que ce soit par la sérialisation ou par les formulaires.

`to_python(value)[source]`[¶](#)

Convertit la valeur dans le bon objet Python. Elle opère à l'inverse de `value_to_string()` et est également appelée dans `clean()`.

Voir [Conversion de valeurs en objets Python](#) pour son utilisation.

En plus de l'enregistrement dans la base de données, le champ doit aussi savoir comment sérialiser sa valeur :

`value_to_string(obj)[source]`[¶](#)

Convertit `obj` en une chaîne. Utilisé pour sérialiser la valeur du champ.

Voir [Conversion des données de champs pour la sérialisation](#) pour son utilisation.

Lors de l'utilisation de **formulaires de modèles**, le champ `Field` doit savoir quel champ de formulaire doit le représenter en appelant :

`formfield(form_class=None, choices_form_class=None, **kwargs)[source]`[¶](#)

Renvoie le champ de formulaire `django.forms.Field` utilisé par défaut pour ce champ dans un formulaire `ModelForm`.

Par défaut, si `form_class` et `choices_form_class` valent les deux `None`, c'est un champ `CharField` qui sera renvoyé. Si le champ possède l'attribut `choices` et que `choices_form_class` n'est pas renseigné, ce sera un champ `TypedChoiceField`.

Voir [Sélection du champ de formulaire pour un champ de modèle](#) pour son utilisation.

`deconstruct()[source]`[¶](#)

Renvoie un tuple à 4 éléments avec suffisamment d'informations pour recréer le champ :

1. Le nom du champ dans le modèle.
2. Le chemin d'importation du champ (par ex. `"django.db.models.IntegerField"`). Il est recommandé d'indiquer la version la plus portable, donc moins le chemin est spécifique, mieux c'est.
3. Une liste de paramètres positionnels.
4. Un dictionnaire de paramètres nommés.

Cette méthode doit être ajoutée aux champs créés avant la version 1.7 afin de pouvoir migrer leurs données en utilisant les [Migrations](#).

Référence d'attributs des champs¶

Chaque instance **Field** contient plusieurs attributs qui permettent d'inspecter son comportement. Utilisez ces attributs plutôt que de vous baser sur des contrôles de type **isinstance** lorsque vous avez besoin d'écrire du code qui dépend de la fonctionnalité d'un champ. Ces attributs peuvent être utilisés conjointement avec l'"API **Model._meta** pour préciser une recherche de types de champs spécifiques. Les champs de modèles personnalisés devraient implémenter ces drapeaux.

Attributs pour les champs¶

Field.auto_created¶

Drapeau booléen indiquant si le champ a été créé automatiquement, comme pour le champ **OneToOneField** utilisé dans un contexte d'héritage de modèles.

Field.concrete¶

Drapeau booléen indiquant si le champ est associé à une colonne de base de données.

Field.hidden¶

Drapeau booléen indiquant si un champ est utilisé pour appuyer la fonctionnalité d'un autre champ non masqué (par ex. les champs **content_type** et **object_id** qui constituent une relation **GenericForeignKey**). Le drapeau **hidden** est utilisé pour distinguer ce qui constitue le sous-ensemble public des champs du modèle de tous les champs du modèle.

Note

Options.get_fields() exclut par défaut les champs masqués. Indiquez **include_hidden=True** pour renvoyer les champs masqués dans sa réponse.

Field.is_relation¶

Drapeau booléen indiquant si un champ contient des références à un ou plusieurs autres modèles pour sa fonctionnalité (par ex. **ForeignKey**, **ManyToManyField**, **OneToOneField**, etc.).

Field.model¶

Renvoie le modèle dans lequel le champ est défini. Si un champ est défini dans la classe parent d'un modèle, **model** désigne alors la classe parente, et non pas la classe de l'instance.

Attributs pour les champs de relation¶

Ces attributs sont utilisés pour interroger la cardinalité et d'autres détails d'une relation. Ces attributs sont présents sur tous les champs ; cependant, ils n'ont des valeurs booléennes (plutôt que **None**) que si le champ est de type relationnel (**Field.is_relation=True**).

Field.many_to_many¶

Drapeau booléen qui vaut **True** si le champ est une relation plusieurs-à-plusieurs ; **False** sinon. Le seul champ inclus dans Django pour lequel cet attribut vaut **True** est **ManyToManyField**.

Field.many_to_one¶

Drapeau booléen qui vaut **True** si le champ est une relation plusieurs-à-un, comme pour un champ **ForeignKey** ; **False** sinon.

Field.one_to_many¶

Drapeau booléen qui vaut **True** si le champ est une relation un-à-plusieurs, comme pour **GenericRelation** ou la contrepartie d'une clé **ForeignKey** ; **False** sinon.

Field.one_to_one

Drapeau booléen qui vaut **True** si le champ est une relation un-à-un, comme pour un champ **OneToOneField** ; **False** sinon.

Field.related_model

Pointe vers le modèle en lien avec le champ. Par exemple, **Author** dans **ForeignKey(Author, on_delete=models.CASCADE)**. Le **related_model** d'une clé **GenericForeignKey** vaut toujours **None**.