

Écriture de votre première application Django, 1ère partie

Apprenons avec un exemple.

Tout au long de ce tutoriel, nous vous guiderons dans la création d'une application simple de sondage.

Cela consistera en deux parties :

- Un site public qui permet à des gens de voir les sondages et d'y répondre.
- Un site d'administration qui permet d'ajouter, de modifier et de supprimer des sondages.

Nous supposons que [Django est déjà installé](#). Vous pouvez savoir si Django est installé et sa version en exécutant la commande suivante :

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

Si Django est installé, vous devriez voir apparaître la version de l'installation. Dans le cas contraire, vous obtiendrez une erreur disant « No module named django » (aucun module nommé django).

Ce didacticiel est écrit pour Django 1.9 et Python 3.4 (ou plus récent). Si la version de Django ne correspond pas, référez-vous au didacticiel correspondant à votre version de Django en utilisant le sélecteur de version au bas de cette page, ou mettez à jour Django à la version la plus récente. Si vous utilisez encore Python 2.7, il sera nécessaire d'ajuster légèrement les exemples de code, comme expliqué dans les commentaires.

Consultez [Comment installer Django](#) pour tout conseil sur la manière de supprimer d'anciennes versions de Django pour en installer une nouvelle.

Où obtenir de l'aide :

Si vous avez des problèmes au long de ce tutoriel, écrivez un message sur [django-users](#) (anglophone) ou passez sur [#django-fr sur irc.freenode.net](#) pour discuter avec d'autres utilisateurs de Django qui pourraient vous aider.

Création d'un projet

Si c'est la première fois que vous utilisez Django, vous devrez vous occuper de quelques éléments de configuration initiaux. Plus précisément, vous devrez lancer la génération automatique de code qui mettra en place un [projet](#) Django – un ensemble de réglages particuliers à une instance de Django, qui comprend la configuration de la base de données, des options spécifiques à Django et d'autres propres à l'application.

Depuis un terminal en ligne de commande, déplacez-vous à l'aide de la commande `cd` dans un répertoire dans lequel vous souhaitez conserver votre code, puis lancez la commande suivante :

```
$ django-admin startproject mysite
```

Cela va créer un répertoire `mysite` dans le répertoire courant. Si cela ne fonctionne pas, consultez [Problèmes d'exécution de django-admin](#).

Note

Vous devez éviter de nommer vos projets en utilisant des noms réservés de Python ou des noms de composants de Django. Cela signifie en particulier que vous devez éviter d'utiliser des noms comme `django` (qui entrerait en conflit avec Django lui-même) ou `test` (qui entrerait en conflit avec un composant intégré de Python).

À quel endroit ce code devrait-il se trouver ?

Si vous avez une expérience en PHP, vous avez probablement l'habitude de placer votre code dans le répertoire racine de votre serveur Web (comme `/var/www/`). Avec Django, ne le faites pas. Ce n'est pas une bonne idée de mettre du code Python dans le répertoire racine de votre serveur Web, parce que cela crée le risque que l'on puisse voir votre code sur le Web, ce qui n'est pas bon pour la sécurité.

Mettez votre code dans un répertoire en dehors de la racine de votre serveur Web, comme par exemple `home/moncode`.

Voyons ce que [startproject](#) a créé :

```
mysite/  
  manage.py  
  mysite/  
    __init__.py  
    settings.py  
    urls.py  
    wsgi.py
```

Ces fichiers sont :

- Le premier répertoire racine `mysite/` n'est qu'un contenant pour votre projet. Son nom n'a pas d'importance pour Django ; vous pouvez le renommer comme vous voulez.
- `manage.py` : un utilitaire en ligne de commande qui vous permet d'interagir avec ce projet Django de différentes façons. Vous trouverez toutes les informations nécessaires sur `manage.py` dans [django-admin et manage.py](#).
- Le sous-répertoire `mysite/` correspond au paquet Python effectif de votre projet. C'est le nom du paquet Python que vous devrez utiliser pour importer ce qu'il contient (par ex. `mysite.urls`).

- `mysite/__init__.py`: An empty file that tells Python that this directory should be considered a Python package. If you're a Python beginner, read [more about packages](#) in the official Python docs.
- `mysite/settings.py` : réglages et configuration de ce projet Django. [Les réglages de Django](#) vous apprendra tout sur le fonctionnement des réglages.
- `mysite/urls.py` : les déclarations des URL de ce projet Django, une sorte de « table des matières » de votre site Django. Vous pouvez en lire plus sur les URL dans [Distribution des URL](#).
- `mysite/wsgi.py` : un point d'entrée pour les serveurs Web compatibles WSGI pour déployer votre projet. Voir [Comment déployer avec WSGI](#) pour plus de détails.

Le serveur de développement

Vérifions que votre projet Django fonctionne. Déplacez-vous dans le répertoire `mysite` si ce n'est pas déjà fait, et lancez les commandes suivantes :

```
$ python manage.py runserver
```

Vous verrez les messages suivants défiler en ligne de commande :

```
Performing system checks...
```

```
System check identified no issues (0 silenced).
```

```
You have unapplied migrations; your app may not work properly until they are applied.
```

```
Run 'python manage.py migrate' to apply them.
```

```
août 01, 2016 - 15:50:53
```

```
Django version 1.9, using settings 'mysite.settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

```
Quit the server with CONTROL-C.
```

Note

Ignorez pour l'instant l'avertissement au sujet des migrations de base de données non appliquées ; nous nous occuperons de la base de données tantôt.

Vous avez démarré le serveur de développement de Django, un serveur Web léger entièrement écrit en Python. Nous l'avons inclus avec Django de façon à vous permettre de développer rapidement, sans avoir à vous occuper de la configuration d'un serveur de production – comme Apache – tant que vous n'en avez pas besoin.

C'est le moment de noter soigneusement ceci : **n'utilisez jamais** ce serveur pour quoi que ce soit qui s'approche d'un environnement de production. Il est fait seulement pour tester votre travail pendant le développement (notre métier est le développement d'environnements Web, pas de serveurs Web).

Maintenant que le serveur tourne, allez à l'adresse <http://127.0.0.1:8000> avec votre navigateur Web. Vous verrez une page avec le message « Welcome to Django » sur un joli fond bleu pastel. Ça marche !

Modification du port

Par défaut, la commande [runserver](#) démarre le serveur de développement sur l'IP interne sur le port 8000.

Si vous voulez changer cette valeur, passez-la comme paramètre sur la ligne de commande. Par exemple, cette commande démarre le serveur sur le port 8080 :

```
$ python manage.py runserver 8080
```

Si vous voulez changer l'IP du serveur, passez-la comme paramètre avec le port. Pour écouter toutes les IP publiques (utile si vous souhaitez montrer votre travail à des personnes sur d'autres ordinateurs de votre réseau), faites :

```
$ python manage.py runserver 0.0.0.0:8000
```

La documentation complète du serveur de développement se trouve dans la référence de [runserver](#).

Rechargement automatique de [runserver](#)

Le serveur de développement recharge automatiquement le code Python lors de chaque requête si nécessaire. Vous ne devez pas redémarrer le serveur pour que les changements de code soient pris en compte. Cependant, certaines actions comme l'ajout de fichiers ne provoquent pas de redémarrage, il est donc nécessaire de redémarrer manuellement le serveur dans ces cas.

Création de l'application Polls

Maintenant que votre environnement – un « projet » – est en place, vous êtes prêt à commencer à travailler.

Chaque application que vous écrivez avec Django est en fait un paquet Python qui respecte certaines conventions. Django est livré avec un utilitaire qui génère automatiquement la structure des répertoires de base d'une application, ce qui vous permet de vous concentrer sur l'écriture du code, plutôt que sur la création de répertoires.

Projets vs. applications

Quelle est la différence entre un projet et une application ? Une application est une application Web qui fait quelque chose – par exemple un système de blog, une base de données publique ou une application de sondage. Un projet est un ensemble de réglages et d'applications pour un site Web particulier. Un projet peut contenir plusieurs applications. Une application peut apparaître dans plusieurs projets.

Your apps can live anywhere on your [Python path](#). In this tutorial, we'll create our poll app right next to your `manage.py` file so that it can be imported as its own top-level module, rather than a submodule of `mysite`.

Pour créer votre application, assurez-vous d’être dans le même répertoire que `manage.py` et saisissez cette commande :

```
$ python manage.py startapp polls
```

Cela va créer un répertoire `polls`, qui est structuré de la façon suivante :

```
polls/
  __init__.py
  admin.py
  apps.py
  migrations/
    __init__.py
  models.py
  tests.py
  views.py
```

Cette structure de répertoire accueillera l’application de sondage.

Écriture d’une première vue

Écrivons la première vue. Ouvrez le fichier `polls/views.py` et placez-y le code Python suivant :

```
polls/views.py

from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world. You're at the polls index.")
```

C’est la vue Django la plus simple possible. Pour appeler cette vue, il s’agit de l’associer à une URL, et pour cela nous avons besoin d’un `URLconf`.

Pour créer un `URLconf` dans le répertoire `polls`, créez un fichier nommé `urls.py`. Votre répertoire d’application devrait maintenant ressembler à ceci :

```
polls/
  __init__.py
  admin.py
  apps.py
  migrations/
    __init__.py
  models.py
  tests.py
  urls.py
  views.py
```

Dans le fichier `polls/urls.py`, insérez le code suivant :

```
polls/urls.py

from django.conf.urls import url

from . import views
```

```
urlpatterns = [  
    url(r'^$', views.index, name='index'),  
]
```

L'étape suivante est de faire pointer l'URLconf racine vers le module `polls.urls`. Dans `mysite/urls.py`, ajoutez une importation `django.conf.urls.include` et insérez un appel [`include\(\)`](#) dans la liste `urlpatterns`, ce qui donnera :

`mysite/urls.py`

```
from django.conf.urls import include, url  
from django.contrib import admin  
  
urlpatterns = [  
    url(r'^polls/', include('polls.urls')),  
    url(r'^admin/', admin.site.urls),  
]
```

The [`include\(\)`](#) function allows referencing other URLconfs. Note that the regular expressions for the [`include\(\)`](#) function doesn't have a `$` (end-of-string match character) but rather a trailing slash. Whenever Django encounters [`include\(\)`](#), it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

L'idée derrière [`include\(\)`](#) est de faciliter la connexion d'URL. Comme l'application de sondages possède son propre URLconf (`polls/urls.py`), ses URL peuvent être injectés sous « `/polls/` », sous « `/fun_polls/` » ou sous « `/content/polls/` » ou tout autre chemin racine sans que cela change quoi que ce soit au fonctionnement de l'application.

Quand utiliser [`include\(\)`](#)

Il faut toujours utiliser `include()` lorsque l'on veut inclure d'autres motifs d'URL. `admin.site.urls` est la seule exception à cette règle.

Cela ne correspond pas à ce que vous voyez ?

Si vous voyez `include(admin.site.urls)` au lieu d'un simple `admin.site.urls`, il est alors probable que vous utilisez une version de Django qui ne correspond pas à la version de ce tutoriel. Consultez une version plus ancienne de ce tutoriel ou installez une version plus récente de Django.

Vous avez maintenant relié une vue `index` dans la configuration d'URL. Vérifions qu'elle fonctionne en lançant la commande suivante :

```
$ python manage.py runserver
```

Ouvrez <http://localhost:8000/polls/> dans votre navigateur et vous devriez voir le texte “*Hello, world. You're at the polls index.*” qui a été défini dans la vue `index`.

La fonction `url()` reçoit quatre paramètres, dont deux sont obligatoires : `regex` et `view`, et deux facultatifs : `kwargs` et `name`. À ce stade, il est intéressant d'examiner le rôle de chacun de ces paramètres.

Paramètre d'`url()` : `regex`

Le terme « regex » est communément utilisé comme raccourci pour « expression régulière », qui consiste en une syntaxe pour retrouver des motifs dans des chaînes de caractères, ou dans ce cas, dans des modèles d'URL. Django commence par la première expression régulière puis continue de parcourir la liste en comparant l'URL reçue avec chaque expression jusqu'à ce qu'il en trouve une qui correspond.

Notez que ces expressions régulières ne cherchent pas dans les paramètres GET et POST, ni dans le nom de domaine. Par exemple, dans une requête vers `https://www.example.com/myapp/`, l'URLconf va chercher `myapp/`. Dans une requête vers `https://www.example.com/myapp/?page=3`, l'URLconf va aussi chercher `myapp/`.

Si vous avez besoin d'aide avec les expressions régulières, jetez un oeil à l'[article Wikipedia](#) et à la documentation du module Python [re](#). À noter aussi que le livre « Mastering Regular Expressions », écrit par Jeffrey Friedl, est remarquable. En pratique, il n'y a cependant pas besoin d'être un expert des expressions régulières, car il suffit de savoir capturer des motifs simples. En fait, les expressions complexes risquent de pénaliser les performances, et il faut plutôt éviter de faire appel à tout le potentiel des expressions régulières.

Enfin, une note sur la performance : ces expressions régulières sont compilées la première fois que le module contenant l'URLconf est chargé. Elles sont extrêmement rapides (tant qu'elles ne sont pas trop complexes, comme indiqué ci-dessus).

Paramètre d'`url()` : `view`

Lorsque Django trouve une expression régulière qui correspond, il appelle la fonction de vue spécifiée, avec un objet [HttpRequest](#) comme premier paramètre et toutes valeurs « capturées » par l'expression régulière comme autres paramètres. Si l'expression régulière utilise des captures simples, les valeurs sont passées comme paramètres positionnels ; si ce sont des captures nommées, les valeurs sont passées comme paramètres nommés. Nous montrerons cela par un exemple un peu plus loin.

Paramètre d'`url()` : `kwargs`

Des paramètres nommés arbitraires peuvent être transmis dans un dictionnaire vers la vue cible. Nous n'allons pas exploiter cette fonctionnalité dans ce tutoriel.

Paramètre d'[url\(\)](#) : name

Naming your URL lets you refer to it unambiguously from elsewhere in Django, especially from within templates. This powerful feature allows you to make global changes to the URL patterns of your project while only touching a single file.

Lorsque vous serez familiarisé avec le flux de base des requêtes et réponses, lisez la [partie 2 de ce tutoriel](#) pour commencer de travailler avec la base de données.

[Guide d'installation rapide](#)

[Écriture de votre première application Django, 2ème partie](#)

Écriture de votre première application Django, 2ème partie

Ce tutoriel commence là où le [tutoriel 1](#) s'achève. Nous allons configurer la base de données, créer le premier modèle et aborder une introduction rapide au site d'administration généré automatiquement par Django.

Configuration de la base de données

Maintenant, ouvrez `mysite/settings.py`. C'est un module Python tout à fait normal, avec des variables de module qui représentent des réglages de Django.

La configuration par défaut utilise SQLite. Si vous débutez avec les bases de données ou que vous voulez juste essayer Django, il s'agit du choix le plus simple. SQLite est inclus dans Python, vous n'aurez donc rien d'autre à installer pour utiliser ce type de base de données. Lorsque vous démarrez votre premier projet réel, cependant, vous pouvez utiliser une base de données plus robuste comme PostgreSQL, afin d'éviter les maux de tête consécutifs au changement perpétuel d'une base de données à l'autre.

Si vous souhaitez utiliser une autre base de données, installez le [connecteur de base de données](#) approprié, et changez les clés suivantes dans l'élément 'default' de [DATABASES](#) pour indiquer les paramètres de connexion de votre base de données :

- [ENGINE](#) – Choisissez parmi 'django.db.backends.sqlite3', 'django.db.backends.postgresql', 'django.db.backends.mysql' ou 'django.db.backends.oracle'. D'autres moteurs sont [également disponibles](#).
- [NAME](#) – Le nom de votre base de données. Si vous utilisez SQLite, la base de données sera un fichier sur votre ordinateur. Dans ce cas, [NAME](#) doit être le chemin absolu complet de celui-ci, y

compris le nom de fichier. La valeur par défaut, `os.path.join(BASE_DIR, 'db.sqlite3')`, stocke ce fichier dans le répertoire de votre projet.

Si vous utilisez une autre base de données que SQLite, des réglages supplémentaires doivent être indiqués, comme [USER](#), [PASSWORD](#) ou [HOST](#). Pour plus de détails, consultez la documentation de référence de [DATABASES](#).

Pour les bases de données autres que SQLite

Si vous utilisez une base de données autre que SQLite, assurez-vous maintenant d'avoir créé la base de données. Faites-le avec `CREATE DATABASE nom_de_la_base;` dans le shell interactif de votre base de données.

Vérifiez également que l'utilisateur de base de données indiqué dans le fichier `mysite/settings.py` possède la permission de créer des bases de données. Cela permet de créer automatiquement une [base de données de test](#) qui sera nécessaire plus tard dans le tutoriel.

Si vous utilisez SQLite, vous n'avez rien à créer à l'avance - le fichier de la base de données sera automatiquement créé lorsque ce sera nécessaire.

Puisque vous êtes en train d'éditer `mysite/settings.py`, définissez [TIME_ZONE](#) selon votre fuseau horaire.

Notez également le réglage [INSTALLED_APPS](#) au début du fichier. Cette variable contient le nom des applications Django qui sont actives dans cette instance de Django. Les applications peuvent être utilisées dans des projets différents, et vous pouvez emballer et distribuer les vôtres pour que d'autres les utilisent dans leurs projets.

Par défaut, [INSTALLED_APPS](#) contient les applications suivantes, qui sont toutes contenues dans Django :

- [django.contrib.admin](#) – Le site d'administration. Vous l'utiliserez très bientôt.
- [django.contrib.auth](#) – Un système d'authentification.
- [django.contrib.contenttypes](#) – Une structure pour les types de contenu (content types).
- [django.contrib.sessions](#) – Un cadre pour les sessions.
- [django.contrib.messages](#) – Un cadre pour l'envoi de messages.
- [django.contrib.staticfiles](#) – Une structure pour la prise en charge des fichiers statiques.

Ces applications sont incluses par défaut par commodité parce que ce sont les plus communément utilisées.

Certaines de ces applications utilisent toutefois au moins une table de la base de données, donc il nous faut créer les tables dans la base avant de pouvoir les utiliser. Pour ce faire, lancez la commande suivante :

```
$ python manage.py migrate
```

La commande [`migrate`](#) examine le réglage [`INSTALLED_APPS`](#) et crée les tables de base de données nécessaires en fonction des réglages de base de données dans votre fichier `mysite/settings.py` et des migrations de base de données contenues dans l'application (nous les aborderons plus tard). Vous verrez apparaître un message pour chaque migration appliquée. Si cela vous intéresse, lancez le client en ligne de commande de votre base de données et tapez `\dt` (PostgreSQL), `SHOW TABLES;` (MySQL), `. schema` (SQLite) ou `SELECT TABLE_NAME FROM USER_TABLES;` (Oracle) pour afficher les tables créées par Django.

Pour les minimalistes

Comme il a été indiqué ci-dessus, les applications incluses par défaut sont les plus communes, mais tout le monde n'en a pas forcément besoin. Si vous n'avez pas besoin d'une d'entre elles (ou de toutes), vous êtes libre de commenter ou effacer les lignes concernées du réglage [`INSTALLED_APPS`](#) avant de lancer [`migrate`](#). La commande [`migrate`](#) n'exécutera les migrations que pour les applications listées dans [`INSTALLED_APPS`](#).

Création des modèles

Nous allons maintenant définir les modèles – essentiellement, le schéma de base de données, avec quelques métadonnées supplémentaires.

Philosophie

Un modèle est la source d'information unique et définitive pour vos données. Il contient les champs essentiels et le comportement attendu des données que vous stockerez. Django respecte la [`philosophie DRY`](#) (Don't Repeat Yourself, « ne vous répétez pas »). Le but est de définir le modèle des données à un seul endroit, et ensuite de dériver automatiquement ce qui est nécessaire à partir de celui-ci.

Ceci inclut les migrations. Au contraire de Ruby On Rails, par exemple, les migrations sont entièrement dérivées du fichier des modèles et ne sont fondamentalement qu'un historique que Django peut parcourir pour mettre à jour le schéma de la base de données pour qu'il corresponde aux modèles actuels.

Dans notre application de sondage simple, nous allons créer deux modèles : `Question` et `Choice` (choix). Une `Question` possède une question et une date de mise en ligne. Un choix a deux champs : le texte représentant le choix et le décompte des votes. Chaque choix est associé à une `Question`.

Ces concepts sont représentés par de simples classes Python. Éditez le fichier `polls/models.py` de façon à ce qu'il ressemble à ceci :

polls/models.py

```
from django.db import models
```

```
class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

Le code est trivial. Chaque modèle est représenté par une classe qui hérite de [django.db.models.Model](#). Chaque modèle possède des variables de classe, chacune d'entre elles représentant un champ de la base de données pour ce modèle.

Chaque champ est représenté par une instance d'une classe [Field](#) – par exemple, [CharField](#) pour les champs de type caractère, et [DateTimeField](#) pour les champs date et heure. Cela indique à Django le type de données que contient chaque champ.

Le nom de chaque instance de [Field](#) (par exemple, `question_text` ou `pub_date`) est le nom du champ en interne. Vous l'utiliserez dans votre code Python et votre base de données l'utilisera comme nom de colonne.

Vous pouvez utiliser le premier paramètre de position (facultatif) d'un [Field](#) pour donner un nom plus lisible au champ. C'est utilisé par le système d'inspection de Django, et aussi pour la documentation. Si ce paramètre est absent, Django utilisera le nom du champ interne. Dans l'exemple, nous n'avons défini qu'un seul nom plus lisible, pour `Question.pub_date`. Pour tous les autres champs, nous avons considéré que le nom interne était suffisamment lisible.

Certaines classes [Field](#) possèdent des paramètres obligatoires. La classe [CharField](#), par exemple, a besoin d'un attribut `max_length`. Ce n'est pas seulement utilisé dans le schéma de base de la base de données, mais également pour valider les champs, comme nous allons voir prochainement.

Un champ [Field](#) peut aussi autoriser des paramètres facultatifs ; dans notre cas, nous avons défini à 0 la valeur `default` de `votes`.

Finalement, notez que nous définissons une relation, en utilisant [ForeignKey](#). Cela indique à Django que chaque vote (`Choice`) n'est relié qu'à une seule `Question`. Django propose tous les modèles classiques de relations : plusieurs-à-un, plusieurs-à-plusieurs, un-à-un.

Activation des modèles

Ce petit morceau de code décrivant les modèles fournit beaucoup d'informations à Django. Cela lui permet de :

- Créer un schéma de base de données (instructions `CREATE TABLE`) pour cette application.
- Créer une API Python d'accès aux bases de données pour accéder aux objets `Question` et `Choice`.

Mais il faut d'abord indiquer à notre projet que l'application de sondages `polls` est installée.

Philosophie

Les applications de Django sont comme des pièces d'un jeu de construction : vous pouvez utiliser une application dans plusieurs projets, et vous pouvez distribuer les applications, parce qu'elles n'ont pas besoin d'être liées à une installation Django particulière.

Éditez encore une fois le fichier `mysite/settings.py` et modifiez le réglage [`INSTALLED_APPS`](#) pour qu'il contienne la chaîne de caractères `'polls.apps.PollsConfig'`. Il devrait ressembler à ceci :

`mysite/settings.py`

```
INSTALLED_APPS = [
    'polls.apps.PollsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Maintenant, Django sait qu'il doit inclure l'application `polls`. Exécutons une autre commande :

```
$ python manage.py makemigrations polls
```

Vous devriez voir quelque chose de similaire à ceci :

```
Migrations for 'polls':
  0001_initial.py:
    - Create model Choice
    - Create model Question
    - Add field question to choice
```

En exécutant `makemigrations`, vous indiquez à Django que vous avez effectué des changements à vos modèles (dans ce cas, vous en avez créé) et que vous aimeriez que ces changements soient stockés sous forme de *migration*.

Les migrations sont le moyen utilisé par Django pour stocker les modifications de vos modèles (et donc de votre schéma de base de données), ce ne sont que des fichiers sur le disque. Vous pouvez consulter la migration pour vos nouveaux modèles si vous le voulez ; il s'agit du fichier `polls/migrations/0001_initial.py`. Soyez sans crainte, vous n'êtes pas censé les lire chaque fois que Django en crée, mais ils sont conçus pour être humainement lisibles au cas où vous auriez besoin d'adapter manuellement les processus de modification de Django.

Il existe une commande qui exécute les migrations et gère automatiquement votre schéma de base de données, elle s'appelle [migrate](#). Nous y viendrons bientôt, mais tout d'abord, voyons les instructions SQL que la migration produit. La commande [sqlmigrate](#) accepte des noms de migrations et affiche le code SQL correspondant :

```
$ python manage.py sqlmigrate polls 0001
```

Vous devriez voir quelque chose de similaire à ceci (remis en forme par souci de lisibilité) :

```
BEGIN;
--
-- Create model Choice
--
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
--
-- Create model Question
--
CREATE TABLE "polls_question" (
    "id" serial NOT NULL PRIMARY KEY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
--
-- Add field question to choice
--
ALTER TABLE "polls_choice" ADD COLUMN "question_id" integer NOT NULL;
ALTER TABLE "polls_choice" ALTER COLUMN "question_id" DROP DEFAULT;
CREATE INDEX "polls_choice_7aa0f6ee" ON "polls_choice" ("question_id");
ALTER TABLE "polls_choice"
    ADD CONSTRAINT "polls_choice_question_id_246c99a640fbbd72_fk_polls_question_id"
        FOREIGN KEY ("question_id")
        REFERENCES "polls_question" ("id")
        DEFERRABLE INITIALLY DEFERRED;

COMMIT;
```

Notez les points suivants :

- Ce que vous verrez dépendra de la base de données que vous utilisez. L'exemple ci-dessus est généré pour PostgreSQL.
- Les noms de tables sont générés automatiquement en combinant le nom de l'application (`polls`) et le nom du modèle en minuscules – `question` et `choice` (vous pouvez modifier ce comportement).
- Des clés primaires (ID) sont ajoutées automatiquement (vous pouvez modifier ceci également).
- Par convention, Django ajoute `"_id"` au nom de champ de la clé étrangère. Et oui, vous pouvez aussi changer ça.

- La relation de clé étrangère est rendue explicite par une contrainte `FOREIGN KEY`. Ne prenez pas garde aux parties `DEFERRABLE`; elles ne font qu'indiquer à PostgreSQL de ne pas contrôler la clé étrangère avant la fin de la transaction.
- Ce que vous voyez est adapté à la base de données que vous utilisez. Ainsi, des champs spécifiques à celle-ci comme `auto_increment` (MySQL), `serial` (PostgreSQL) ou `integer primary key autoincrement` (SQLite) sont gérés pour vous automatiquement. Tout comme pour les guillemets autour des noms de champs (simples ou doubles).
- La commande [sqlmigrate](#) n'exécute pas réellement la migration dans votre base de données - elle se contente de l'afficher à l'écran de façon à vous permettre de voir le code SQL que Django pense nécessaire. C'est utile pour savoir ce que Django s'apprête à faire ou si vous avez des administrateurs de base de données qui exigent des scripts SQL pour faire les modifications.

Si cela vous intéresse, vous pouvez aussi exécuter [python manage.py check](#); cette commande vérifie la conformité de votre projet sans appliquer de migration et sans toucher à la base de données.

Maintenant, exécutez à nouveau la commande [migrate](#) pour créer les tables des modèles dans votre base de données :

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, polls, auth, sessions
Running migrations:
  Rendering model states... DONE
  Applying polls.0001_initial... OK
```

La commande [migrate](#) sélectionne toutes les migrations qui n'ont pas été appliquées (Django garde la trace des migrations appliquées en utilisant une table spéciale dans la base de données : `django_migrations`) puis les exécute dans la base de données, ce qui consiste essentiellement à synchroniser les changements des modèles avec le schéma de la base de données.

Les migrations sont très puissantes et permettent de gérer les changements de modèles dans le temps, au cours du développement d'un projet, sans devoir supprimer la base de données ou ses tables et en refaire de nouvelles. Une migration s'attache à mettre à jour la base de données en live, sans perte de données. Nous les aborderons plus en détails dans une partie ultérieure de ce didacticiel, mais pour l'instant, retenez le guide en trois étapes pour effectuer des modifications aux modèles :

- Modifiez les modèles (dans `models.py`).
- Exécutez [python manage.py makemigrations](#) pour créer des migrations correspondant à ces changements.
- Exécutez [python manage.py migrate](#) pour appliquer ces modifications à la base de données.

La raison de séparer les commandes pour créer et appliquer les migrations est que celles-ci vont être ajoutées dans votre système de gestion de versions et qu'elles seront livrées avec l'application ; elles ne font pas que faciliter le développement, elles sont également exploitables par d'autres développeurs ou en production.

Lisez la [documentation de django-admin](#) pour avoir toutes les informations sur ce que `manage.py` peut faire.

Jouer avec l'interface de programmation (API)

Maintenant, utilisons un shell interactif Python pour jouer avec l'API que Django met gratuitement à votre disposition. Pour lancer un shell Python, utilisez cette commande :

```
$ python manage.py shell
```

Nous utilisons celle-ci au lieu de simplement taper « python », parce que `manage.py` définit la variable d'environnement `DJANGO_SETTINGS_MODULE`, qui indique à Django le chemin d'importation Python vers votre fichier `mysite/settings.py`.

Se passer de `manage.py`

Si vous préférez ne pas utiliser `manage.py`, pas de problème. Il suffit de définir la variable d'environnement `DJANGO_SETTINGS_MODULE` à `mysite.settings`, de lancer un shell Python standard et de configurer Django :

```
>>> import django
>>> django.setup()
```

Si une exception `AttributeError` apparaît, il est alors probable que vous utilisez une version de Django qui ne correspond pas à la version de ce tutoriel. Consultez alors une version plus ancienne de ce tutoriel ou installez une version plus récente de Django.

Vous devez exécuter `python` dans le même répertoire que celui où se trouve `manage.py`, ou assurez-vous que ce répertoire est dans le chemin Python afin que `import mysite` fonctionne.

Pour plus d'informations sur tout ceci, voyez la [documentation de django-admin](#).

Une fois dans le shell, explorez l'[API de base de données](#):

```
>>> from polls.models import Question, Choice    # Import the model classes we just
wrote.

# No questions are in the system yet.
>>> Question.objects.all()
[]

# Create a new Question.
# Support for time zones is enabled in the default settings file, so
# Django expects a datetime with tzinfo for pub_date. Use timezone.now()
# instead of datetime.datetime.now() and it will do the right thing.
```

```

>>> from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.
>>> q.save()

# Now it has an ID. Note that this might say "1L" instead of "1", depending
# on which database you're using. That's no biggie; it just means your
# database backend prefers to return integers as Python long integer
# objects.
>>> q.id
1

# Access model field values via Python attributes.
>>> q.question_text
"What's new?"
>>> q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Change values by changing the attributes, then calling save().
>>> q.question_text = "What's up?"
>>> q.save()

# objects.all() displays all the questions in the database.
>>> Question.objects.all()
[<Question: Question object>]

```

Une seconde. `<Question: Question object>` n'est, à l'évidence, pas une représentation de cet objet très utile. On va arranger cela en éditant le modèle `Question` (dans le fichier `polls/models.py`) et en ajoutant une méthode `__str__()` à `Question` et à `Choice`:

`polls/models.py`

```

from django.db import models
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible # only if you need to support Python 2
class Question(models.Model):
    # ...
    def __str__(self):
        return self.question_text

@python_2_unicode_compatible # only if you need to support Python 2
class Choice(models.Model):
    # ...
    def __str__(self):
        return self.choice_text

```

Il est important d'ajouter des méthodes `__str__()` à vos modèles, non seulement parce que c'est plus pratique lorsque vous utilisez le shell interactif, mais aussi parce que la représentation des objets est très utilisée dans l'interface d'administration automatique de Django.

Notez que ce sont des méthodes Python classiques. Ajoutons une méthode personnalisée, juste pour la démonstration :

`polls/models.py`


```

import datetime

from django.db import models
from django.utils import timezone

class Question(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)

```

Notez l'ajout de `import datetime` et de `from django.utils import timezone`, pour référencer respectivement le module [datetime](#) standard de Python et les utilitaires de Django liés aux fuseaux horaires de [django.utils.timezone](#). Si vous n'êtes pas habitué à la gestion des fuseaux horaires avec Python, vous pouvez en apprendre plus en consultant la [documentation sur les fuseaux horaires](#).

Enregistrez ces modifications et retournons au shell interactif de Python en exécutant à nouveau `python manage.py shell`:

```

>>> from polls.models import Question, Choice

# Make sure our __str__() addition worked.
>>> Question.objects.all()
[<Question: What's up?>]

# Django provides a rich database lookup API that's entirely driven by
# keyword arguments.
>>> Question.objects.filter(id=1)
[<Question: What's up?>]
>>> Question.objects.filter(question_text__startswith='What')
[<Question: What's up?>]

# Get the question that was published this year.
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
<Question: What's up?>

# Request an ID that doesn't exist, this will raise an exception.
>>> Question.objects.get(id=2)
Traceback (most recent call last):
...
DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides a
# shortcut for primary-key exact lookups.
# The following is identical to Question.objects.get(id=1).
>>> Question.objects.get(pk=1)
<Question: What's up?>

# Make sure our custom method worked.
>>> q = Question.objects.get(pk=1)
>>> q.was_published_recently()
True

```

```

# Give the Question a couple of Choices. The create call constructs a new
# Choice object, does the INSERT statement, adds the choice to the set
# of available choices and returns the new Choice object. Django creates
# a set to hold the "other side" of a ForeignKey relation
# (e.g. a question's choice) which can be accessed via the API.
>>> q = Question.objects.get(pk=1)

# Display any choices from the related object set -- none so far.
>>> q.choice_set.all()
[]

# Create three choices.
>>> q.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)

# Choice objects have API access to their related Question objects.
>>> c.question
<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.
>>> q.choice_set.all()
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
>>> q.choice_set.count()
3

# The API automatically follows relationships as far as you need.
# Use double underscores to separate relationships.
# This works as many levels deep as you want; there's no limit.
# Find all Choices for any question whose pub_date is in this year
# (reusing the 'current_year' variable we created above).
>>> Choice.objects.filter(question__pub_date__year=current_year)
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]

# Let's delete one of the choices. Use delete() for that.
>>> c = q.choice_set.filter(choice_text__startswith='Just hacking')
>>> c.delete()

```

Pour plus d'informations sur les relations entre modèles, consultez [Accès aux objets liés](#). Pour en savoir plus sur la manière d'utiliser les doubles soulignements pour explorer les champs par l'API, consultez [Recherches par champs](#). Pour tous les détails sur l'API de base de données, consultez la [référence de l'API de base de données](#).

Introduction au site d'administration de Django

Philosophie

La génération de sites d'administration pour votre équipe ou vos clients pour ajouter, modifier et supprimer du contenu est un travail pénible qui ne requiert pas beaucoup de créativité. C'est pour cette raison que Django automatise entièrement la création des interfaces d'administration pour les modèles.

Django a été écrit dans un environnement éditorial, avec une très nette séparation entre les « éditeurs de contenu » et le site « public ». Les gestionnaires du site utilisent le système pour ajouter des nouvelles,

des histoires, des événements, des résultats sportifs, etc., et ce contenu est affiché sur le site public. Django résout le problème de création d'une interface uniforme pour les administrateurs du site qui éditent le contenu.

L'interface d'administration n'est pas destinée à être utilisée par les visiteurs du site ; elle est conçue pour les gestionnaires du site.

Création d'un utilisateur administrateur

Nous avons d'abord besoin de créer un utilisateur qui peut se connecter au site d'administration. Lancez la commande suivante :

```
$ python manage.py createsuperuser
```

Saisissez le nom d'utilisateur souhaité et appuyez sur retour.

Username: admin

On vous demande alors de saisir l'adresse de courriel souhaitée :

Email address: admin@example.com

L'étape finale est de saisir le mot de passe. On vous demande de le saisir deux fois, la seconde fois étant une confirmation de la première.

```
Password: *****  
Password (again): *****  
Superuser created successfully.
```

Démarrage du serveur de développement

Le site d'administration de Django est activé par défaut. Lançons le serveur de développement et explorons-le.

Si le serveur ne tourne pas encore, démarrez-le comme ceci :

```
$ python manage.py runserver
```

À présent, ouvrez un navigateur Web et allez à l'URL « /admin/ » de votre domaine local – par exemple, <http://127.0.0.1:8000/admin/>. Vous devriez voir l'écran de connexion à l'interface d'administration :

Django administration

Username:

Password:

Log in

Comme la [traduction](#) est active par défaut, l'écran de connexion pourrait s'afficher dans votre propre langue, en fonction des réglages de votre navigateur et pour autant qu'il existe une traduction de Django pour cette langue.

Entrée dans le site d'administration

Essayez maintenant de vous connecter avec le compte administrateur que vous avez créé à l'étape précédente. Vous devriez voir apparaître la page d'accueil du site d'administration de Django :

Django administration

WELCOME, ADMIN. [VIEW SITE](#)

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups	+ Add	Change
Users	+ Add	Change

Recent actions

My Actions

None available

Vous devriez voir quelques types de contenu éditable : groupes et utilisateurs. Ils sont fournis par [django.contrib.auth](#), le système d'authentification livré avec Django.

Rendre l'application de sondage modifiable via l'interface d'admin

Mais où est notre application de sondage ? Elle n'est pas affichée sur la page d'index de l'interface d'administration.

Juste une chose à faire : il faut indiquer à l'admin que les objets `Question` ont une interface d'administration. Pour ceci, ouvrez le fichier `polls/admin.py` et éditez-le de la manière suivante :

```
polls/admin.py
```

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

```
from .models import Question
```

```
admin.site.register(Question)
```

Exploration des fonctionnalités de l'interface d'administration

Maintenant que nous avons inscrit `Question` dans l'interface d'administration, Django sait que cela doit apparaître sur la page d'index :

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change
POLLS		
Questions	+ Add	Change


Recent

My Action

None available

Cliquez sur « Questions ». À présent, vous êtes sur la page « liste pour modification » des questions. Cette page affiche toutes les questions de la base de données et vous permet d'en choisir une pour la modifier. Il y a la question « Quoi de neuf ? » que nous avons créée précédemment :

Select question to change

Action:  0 of 1 selected

☐ QUESTION TEXT

☐ What's up?


1 question


Cliquez sur la question « Quoi de neuf ? » pour la modifier :

Change question

Question text:

Date published:

Date: Today 

Time: Now 

À noter ici :

- Le formulaire est généré automatiquement à partir du modèle `Question`.
- Les différents types de champs du modèle ([DateTimeField](#), [CharField](#)) correspondent au composant graphique d'entrée HTML approprié. Chaque type de champ sait comment s'afficher dans l'interface d'administration de Django.

- Chaque [DateTimeField](#) reçoit automatiquement des raccourcis Javascript. Les dates obtiennent un raccourci « Aujourd'hui » et un calendrier en popup, et les heures obtiennent un raccourci « Maintenant » et une popup pratique qui liste les heures couramment saisies.

La partie inférieure de la page vous propose une série d'opérations :

- Enregistrer – Enregistre les modifications et retourne à la page liste pour modification de ce type d'objet.
- Enregistrer et continuer les modifications – Enregistre les modifications et recharge la page d'administration de cet objet.
- Enregistrer et ajouter un nouveau – Enregistre les modifications et charge un nouveau formulaire vierge pour ce type d'objet.
- Supprimer – Affiche une page de confirmation de la suppression.

Si la valeur de « Date de publication » ne correspond pas à l'heure à laquelle vous avez créé cette question dans le [tutoriel 1](#), vous avez probablement oublié de définir la valeur correcte du paramètre [TIME_ZONE](#). Modifiez-le, rechargez la page et vérifiez que la bonne valeur s'affiche.

Modifiez la « Date de publication » en cliquant sur les raccourcis « Aujourd'hui » et « Maintenant ». Puis cliquez sur « Enregistrer et continuer les modifications ». Ensuite, cliquez sur « Historique » en haut à droite de la page. Vous verrez une page listant toutes les modifications effectuées sur cet objet via l'interface d'administration de Django, accompagnées des date et heure, ainsi que du nom de l'utilisateur qui a fait ce changement :

[Home](#) › [Polls](#) › [Questions](#) › [What's up?](#) › [History](#)

Change history: What's up?

DATE/TIME	USER	ACTION
Sept. 6, 2015, 9:21 p.m.	elky	Changed pub_date.

Lorsque vous serez à l'aise avec l'API des modèles et que vous vous serez familiarisé avec le site d'administration, lisez la [partie 3 de ce tutoriel](#) pour apprendre comment ajouter davantage de vues à notre application de sondage.

[Écriture de votre première application Django, 1ère partie](#)
[Écriture de votre première application Django, 3ème partie](#)

Écriture de votre première application Django,

3ème partie

Ce tutoriel commence là où le [tutoriel 2](#) s'achève. Nous continuons l'application de sondage Web et allons nous focaliser sur la création de l'interface publique – les « vues ».

Aperçu

Une vue est un « type » de page Web dans votre application Django qui sert généralement à une fonction précise et possède un gabarit spécifique. Par exemple, dans une application de blog, vous pouvez avoir les vues suivantes :

- La page d'accueil du blog – affiche quelques-uns des derniers billets.
- La page de « détail » d'un billet – lien permanent vers un seul billet.
- La page d'archives pour une année – affiche tous les mois contenant des billets pour une année donnée.
- La page d'archives pour un mois – affiche tous les jours contenant des billets pour un mois donné.
- La page d'archives pour un jour – affiche tous les billets pour un jour donné.
- Action de commentaire – gère l'écriture de commentaires sur un billet donné.

Dans notre application de sondage, nous aurons les quatre vues suivantes :

- La page de sommaire des questions – affiche quelques-unes des dernières questions.
- La page de détail d'une question – affiche le texte d'une question, sans les résultats mais avec un formulaire pour voter.
- La page des résultats d'une question – affiche les résultats d'une question particulière.
- Action de vote – gère le vote pour un choix particulier dans une question précise.

Dans Django, les pages Web et les autres contenus sont générés par des vues. Chaque vue est représentée par une simple fonction Python (ou une méthode dans le cas des vues basées sur des classes). Django choisit une vue en examinant l'URL demandée (pour être précis, la partie de l'URL après le nom de domaine).

Dans votre expérience sur le Web, vous avez certainement rencontré des perles comme par exemple « ME2/Sites/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D9B65B1B ». Vous serez certainement rassuré en sachant que Django permet des styles d'URL bien plus élégants que cela.

Un modèle d'URL est simplement la forme générale d'une URL ; par exemple :
`/archive/<année>/<mois>/`.

Pour passer de l'URL à la vue, Django utilise ce qu'on appelle des « URLconf ». Un URLconf associe des modèles d'URL (définis par des expressions régulières) à des vues.

Ce tutoriel fournit des instructions de base sur l'utilisation des URLconf, et vous pouvez consultez [django.core.urlresolvers](https://docs.djangoproject.com/en/1.11/topics/http/urls/) pour plus de détails.

Écriture de vues supplémentaires

Ajoutons maintenant quelques vues supplémentaires dans `polls/views.py`. Ces vues sont légèrement différentes, car elles acceptent un paramètre :

`polls/views.py`

```
def detail(request, question_id):
    return HttpResponse("You're looking at question %s." % question_id)

def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)

def vote(request, question_id):
    return HttpResponse("You're voting on question %s." % question_id)
```

Liez ces nouvelles vues avec leurs URL dans le module `polls.urls` en ajoutant les appels [`url\(\)`](#) suivants :

`polls/urls.py`

```
from django.conf.urls import url

from . import views

urlpatterns = [
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

Ouvrez votre navigateur à l'adresse « `/polls/34/` ». La méthode `detail()` sera exécutée et affichera l'ID fourni dans l'URL. Essayez aussi « `/polls/34/results/` » et « `/polls/34/vote/` », elles afficheront les pages modèles de résultats et de votes.

When somebody requests a page from your website – say, “`/polls/34/`”, Django will load the `mysite.urls` Python module because it's pointed to by the [`ROOT_URLCONF`](#) setting. It finds the

variable named `urlpatterns` and traverses the regular expressions in order. After finding the match at `^polls/`, it strips off the matching text (`"polls/"`) and sends the remaining text – `"34/"` – to the `'polls.urls'` `URLconf` for further processing. There it matches `r'^(?P<question_id>[0-9]+)/$'`, resulting in a call to the `detail()` view like so:

```
detail(request=<HttpRequest object>, question_id='34')
```

La partie `question_id='34'` vient de `(?P<question_id>[0-9]+)`. En utilisant des parenthèses autour d'un motif, cela « capture » le texte correspondant à ce motif et l'envoie en tant que paramètre à la fonction de la vue ; le terme `?P<question_id>` définit le nom qui va être utilisé pour identifier le motif trouvé, et `[0-9]+` est une expression régulière pour chercher une suite de chiffres (c'est-à-dire un nombre).

Comme les motifs d'URL sont des expressions régulières, il n'y a vraiment aucune limite à ce qu'ils vous permettent de faire. Et il n'y a pas besoin d'ajouter de fioritures aux URL tel que `.html` – sauf si vous le voulez vraiment, auquel cas vous pouvez faire quelque chose comme ça :

```
url(r'^polls/latest\.html$', views.index),
```

Mais ne le faites pas. C'est stupide.

Écriture de vues qui font réellement des choses

Chaque vue est responsable de faire une des deux choses suivantes : retourner un objet [`HttpResponse`](#) contenant le contenu de la page demandée, ou lever une exception, comme par exemple [`Http404`](#). Le reste, c'est votre travail.

Votre vue peut lire des entrées depuis une base de données, ou pas. Elle peut utiliser un système de gabarits comme celui de Django – ou un système de gabarits tiers – ou pas. Elle peut générer un fichier PDF, produire de l'XML, créer un fichier ZIP à la volée, tout ce que vous voulez, en utilisant les bibliothèques Python que vous voulez.

Voilà tout ce que veut Django : [`HttpResponse`](#) ou une exception.

Parce que c'est pratique, nous allons utiliser l'API de base de données de Django, que nous avons vu dans le [tutoriel 2](#). Voici une ébauche d'une nouvelle vue `index()`, qui affiche les 5 derniers sondages, séparés par des virgules et classés par date de publication :

```
polls/views.py
```

```
from django.http import HttpResponse
```

```
from .models import Question
```

```
def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    output = ', '.join([q.question_text for q in latest_question_list])
```

```
    return HttpResponse(output)

# Leave the rest of the views (detail, results, vote) unchanged
```

Cependant, il y a un problème : l'allure de la page est codée en dur dans la vue. Si vous voulez changer le style de la page, vous devrez modifier votre code python. Nous allons donc utiliser le système de gabarits de Django pour séparer le style du code Python en créant un gabarit que la vue pourra utiliser.

Tout d'abord, créez un répertoire nommé `templates` dans votre répertoire `polls`. C'est là que Django recherche les gabarits.

Le paramètre [`TEMPLATES`](#) de votre projet indique comment Django va charger et produire les gabarits. Le fichier de réglages par défaut configure un moteur `DjangoTemplates` dont l'option [`APP_DIRS`](#) est définie à `True`. Par convention, `DjangoTemplates` recherche un sous-répertoire « `templates` » dans chaque application figurant dans [`INSTALLED_APPS`](#).

Dans le répertoire `templates` que vous venez de créer, créez un autre répertoire nommé `polls` dans lequel vous placez un nouveau fichier `index.html`. Autrement dit, le chemin de votre gabarit doit être `polls/templates/polls/index.html`. Conformément au fonctionnement du chargeur de gabarit `app_directories` (cf. explication ci-dessus), vous pouvez désigner ce gabarit dans Django tout simplement par `polls/index.html`.

Espace de noms des gabarits

Il serait aussi *possible* de placer directement nos gabarits dans `polls/templates` (plutôt que dans un sous-répertoire `polls`), mais ce serait une mauvaise idée. Django choisit le premier gabarit qu'il trouve pour un nom donné et dans le cas où vous avez un gabarit de même nom dans une *autre* application, Django ne fera pas la différence. Il faut pouvoir indiquer à Django le bon gabarit, et la manière la plus simple de faire cela est d'utiliser des espaces de noms. C'est-à-dire que nous plaçons ces gabarits dans un *autre* répertoire portant le nom de l'application.

Insérez le code suivant dans ce gabarit :

`polls/templates/polls/index.html`

```
{% if latest_question_list %}
    <ul>
        {% for question in latest_question_list %}
            <li><a href="/polls/
{{ question.id }}/"/>{{ question.question_text }}</a></li>
        {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

Mettons maintenant à jour notre vue `index` dans `polls/views.py` pour qu'elle utilise le template :

`polls/views.py`

```

from django.http import HttpResponse
from django.template import loader

from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = {
        'latest_question_list': latest_question_list,
    }
    return HttpResponse(template.render(context, request))

```

Ce code charge le gabarit appelé `polls/index.html` et lui fournit un contexte. Ce contexte est un dictionnaire qui fait correspondre des objets Python à des noms de variables de gabarit.

Chargez la page en appelant l'URL « `/polls/` » dans votre navigateur et vous devriez voir une liste à puces contenant la question « What's up » du [tutoriel 2](#). Le lien pointe vers la page de détail de la question.

Un raccourci : [`render\(\)`](#)

Il est très courant de charger un gabarit, remplir un contexte et renvoyer un objet [`HttpResponse`](#) avec le résultat du gabarit interprété. Django fournit un raccourci. Voici la vue `index()` complète, réécrite :

```

polls/views.py

from django.shortcuts import render

from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list': latest_question_list}
    return render(request, 'polls/index.html', context)

```

Notez qu'une fois que nous avons fait ceci dans toutes nos vues, nous n'avons plus à importer [`loader`](#) et [`HttpResponse`](#) (il faut conserver `HttpResponse` tant que les méthodes initiales pour `detail`, `results` et `vote` sont présentes).

La fonction [`render\(\)`](#) prend comme premier paramètre l'objet requête, un nom de gabarit comme deuxième paramètre et un dictionnaire comme troisième paramètre facultatif. Elle retourne un objet [`HttpResponse`](#) composé par le gabarit interprété avec le contexte donné.

Les erreurs 404

Attaquons-nous maintenant à la vue du détail d'une question – la page qui affiche le texte de la question pour un sondage donné. Voici la vue :

polls/views.py

```
from django.http import Http404
from django.shortcuts import render

from .models import Question
# ...
def detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
    return render(request, 'polls/detail.html', {'question': question})
```

Le nouveau concept ici : la vue lève une exception de type [Http404](#) si une question avec l'ID demandé n'existe pas.

Nous parlerons un peu plus tard de ce que pourriez mettre dans le gabarit `polls/detail.html`, mais si vous voulez avoir rapidement un exemple qui fonctionne, écrivez simplement ceci :

polls/templates/polls/detail.html

```
{{ question }}
```

et vous obtiendrez un résultat élémentaire.

Un raccourci : [get_object_or_404\(\)](#)

Il est très courant d'utiliser [get\(\)](#) et de lever une exception [Http404](#) si l'objet n'existe pas. Django fournit un raccourci. Voici la vue `detail()` réécrite :

polls/views.py

```
from django.shortcuts import get_object_or_404, render

from .models import Question
# ...
def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/detail.html', {'question': question})
```

La fonction [get_object_or_404\(\)](#) prend un modèle Django comme premier paramètre et un nombre arbitraire de paramètres mots-clés, qu'il transmet à la méthode [get\(\)](#) du gestionnaire du modèle. Elle lève une exception [Http404](#) si l'objet n'existe pas.

Philosophie

Pourquoi utiliser une fonction auxiliaire [get_object_or_404\(\)](#) plutôt que d'intercepter automatiquement une exception [ObjectDoesNotExist](#) à un plus haut niveau, ou laisser l'API modèle lever une exception [Http404](#) à la place de [ObjectDoesNotExist](#) ?

Parce que cela couplerait la couche de gestion des modèles à la couche de vue. Un des buts principaux de la conception de Django et de garder un couplage le plus faible possible. Un peu de couplage contrôlé est introduit dans le module [django.shortcuts](#).

Il y a aussi une fonction [get_list_or_404\(\)](#), qui fonctionne comme [get_object_or_404\(\)](#), sauf qu'elle utilise [filter\(\)](#) au lieu de la méthode [get\(\)](#). Elle lève une exception [Http404](#) si la liste est vide.

Utilisation du système de gabarits

Revenons à la vue `detail()` de notre application de sondage. Étant donné la variable de contexte `question`, voici à quoi le gabarit `polls/detail.html` pourrait ressembler :

`polls/templates/polls/detail.html`

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

Le système de gabarits utilise une syntaxe d'accès aux attributs de variables à l'aide de points. Dans cet exemple avec `{{ question.question_text }}`, Django commence par rechercher un dictionnaire dans l'objet `question`. En cas d'échec, il cherche un attribut – qui fonctionne dans ce cas. Si la recherche d'attribut n'avait pas fonctionné, Django aurait essayé une recherche d'index sur une liste.

L'appel de méthode a lieu dans la boucle `{% for %}` : `question.choice_set.all` est interprété comme le code Python `question.choice_set.all()`, qui renvoie un itérable d'objets `Choice` et qui convient pour l'utilisation de la balise `{% for %}`.

Voir le [guide des gabarits](#) pour plus d'informations sur les gabarits.

Suppression des URL codés en dur dans les gabarits

Rappelez-vous, lorsque nous avons ajouté le lien vers la question dans le gabarit `polls/index.html`, le lien a été partiellement codé en dur comme ceci :

```
<li><a href="/polls/{{ question.id }}">{{ question.question_text }}</a></li>
```

Le problème de cette approche codée en dur et fortement couplée est qu'il devient fastidieux de modifier les URL dans des projets qui ont beaucoup de gabarits. Cependant, comme vous avez défini le paramètre « name » dans les fonctions [url\(\)](#) du module `polls.urls`, vous pouvez supprimer la dépendance en chemins d'URL spécifiques définis dans les configurations d'URL en utilisant la balise de gabarit `{% url %}` :

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

Le principe de ce fonctionnement est que l'URL est recherchée dans les définitions du module `polls.urls`. Ci-dessous, vous pouvez voir exactement où le nom d'URL de « detail » est défini :

```
...
# the 'name' value as called by the {% url %} template tag
url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
...
```

Si vous souhaitez modifier l'URL de détail des sondages, par exemple sur le modèle `polls/specifics/12/`, il suffit de faire la modification dans `polls/urls.py` au lieu de devoir toucher au contenu du ou des gabarits :

```
...
# added the word 'specifics'
url(r'^specifics/(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
...
```

Espaces de noms et noms d'URL

Le projet du tutoriel ne contient qu'une seule application, `polls`. Dans des projets Django réels, il peut y avoir cinq, dix, vingt applications ou plus. Comment Django arrive-t-il à différencier les noms d'URL entre elles ? Par exemple, l'application `polls` possède une vue `detail` et il se peut tout à fait qu'une autre application du même projet en possède aussi une. Comment peut-on indiquer à Django quelle vue d'application il doit appeler pour une URL lors de l'utilisation de la balise de gabarit `{% url %}` ?

La réponse est donnée par l'ajout d'espaces de noms à votre configuration d'URL. Dans le fichier `polls/urls.py`, ajoutez une variable `app_name` pour définir l'espace de nom de l'application :

```
polls/urls.py

from django.conf.urls import url

from . import views

app_name = 'polls'
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

Modifiez maintenant la partie suivante du gabarit `polls/index.html` :

```
polls/templates/polls/index.html
```

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

pour qu'elle pointe vers la vue « detail » à l'espace de nom correspondant :

polls/templates/polls/index.html

```
<li><a href="{% url 'polls:detail' question.id  
%}">{{ question.question_text }}</a></li>
```

Lorsque vous êtes à l'aise avec l'écriture des vues, lisez la [partie 4 de ce tutoriel](#) pour apprendre la gestion de formulaires simples et les vues génériques.

[Écriture de votre première application Django, 2ème partie](#)

[Écriture de votre première application Django, 4ème partie](#)

Écriture de votre première application Django, 4ème partie

Ce tutoriel commence là où le [tutoriel 3](#) s'achève. Nous continuons l'application de sondage Web et allons nous focaliser sur la gestion de formulaire simple et sur la réduction du code.

Écriture d'un formulaire simple

Nous allons mettre à jour le gabarit de la page de détail (« polls/details.html ») du tutoriel précédent, de manière à ce que le gabarit contienne une balise HTML `<form>` :

polls/templates/polls/detail.html

```
<h1>{{ question.question_text }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
{% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}"
value="{{ choice.id }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

Un résumé rapide :

- Ce gabarit affiche un bouton radio pour chaque choix de question. L'attribut `value` de chaque bouton radio correspond à l'ID du vote choisi. Le nom (`name`) de chaque bouton radio est `"choice"`. Cela signifie que lorsque quelqu'un sélectionne l'un des boutons radio et valide le formulaire, les données POST `choice=#` (où `#` est l'identifiant du choix sélectionné) seront envoyées. Ce sont les concepts de base des formulaires HTML.

- Nous avons défini `{% url 'polls:vote' question.id %}` comme attribut `action` du formulaire, et nous avons précisé `method="post"`. L'utilisation de `method="post"` (par opposition à `method="get"`) est très importante, puisque le fait de valider ce formulaire va entraîner des modifications de données sur le serveur. À chaque fois qu'un formulaire modifie des données sur le serveur, vous devez utiliser `method="post"`. Cela ne concerne pas uniquement Django ; c'est une bonne pratique à adopter en tant que développeur Web.
- `forloop.counter` indique combien de fois la balise `for` a exécuté sa boucle.
- Comme nous créons un formulaire POST (qui modifie potentiellement des données), il faut se préoccuper des attaques inter-sites. Heureusement, vous ne devez pas réfléchir trop longtemps car Django offre un moyen très simple à utiliser pour s'en protéger. En bref, tous les formulaires POST destinés à des URL internes doivent utiliser la balise de gabarit `{% csrf_token %}`.

Maintenant, nous allons créer une vue Django qui récupère les données envoyées pour nous permettre de les exploiter. Souvenez-vous, dans le [tutoriel 3](#), nous avons créé un URLconf pour l'application de sondage contenant cette ligne :

`polls/urls.py`

```
url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
```

Nous avons également créé une implémentation rudimentaire de la fonction `vote()`. Créons maintenant une version fonctionnelle. Ajoutez ce qui suit dans le fichier `polls/views.py`:

`polls/views.py`

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse

from .models import Choice, Question
# ...
def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
        selected_choice = question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.
        return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))
```

Ce code contient quelques points encore non abordés dans ce tutoriel :

- [`request.POST`](#) est un objet similaire à un dictionnaire qui vous permet d'accéder aux données envoyées par leurs clés. Dans ce cas, `request.POST['choice']` renvoie l'ID du choix sélectionné, sous forme d'une chaîne de caractères. Les valeurs dans [`request.POST`](#) sont toujours des chaînes de caractères.

Notez que Django dispose aussi de [`request.GET`](#) pour accéder aux données GET de la même manière – mais nous utilisons explicitement [`request.POST`](#) dans notre code, pour s'assurer que les données ne sont modifiées que par des requêtes POST.

- `request.POST['choice']` lèvera une exception [`KeyError`](#) si `choice` n'est pas spécifié dans les données POST. Le code ci-dessus vérifie qu'une exception [`KeyError`](#) n'est pas levée et réaffiche le formulaire de question avec un message d'erreur si `choice` n'est pas rempli.
- Après l'incrémentation du nombre de votes du choix, le code renvoie une [`HttpResponseRedirect`](#) plutôt qu'une [`HttpResponse`](#) normale. [`HttpResponseRedirect`](#) prend un seul paramètre : l'URL vers laquelle l'utilisateur va être redirigé (voir le point suivant pour la manière de construire cette URL dans ce cas).

Comme le commentaire Python l'indique, vous devez systématiquement renvoyer une [`HttpResponseRedirect`](#) après avoir correctement traité les données POST. Ceci n'est pas valable uniquement avec Django, c'est une bonne pratique du développement Web.

- Dans cet exemple, nous utilisons la fonction [`reverse\(\)`](#) dans le constructeur de [`HttpResponseRedirect`](#). Cette fonction nous évite de coder en dur une URL dans une vue. On lui donne en paramètre la vue vers laquelle nous voulons rediriger ainsi que la partie variable de l'URL qui pointe vers cette vue. Dans ce cas, en utilisant l'URLconf défini dans la [partie 3 de ce tutoriel](#), l'appel de la fonction [`reverse\(\)`](#) va renvoyer la chaîne de caractères :

```
'/polls/3/results/'
```

où 3 est la valeur de `question.id`. Cette URL de redirection va ensuite appeler la vue `'results'` pour afficher la page finale.

Comme expliqué dans la [partie 3 de ce tutoriel](#), `request` est un objet [`HttpRequest`](#). Pour plus d'informations sur les objets [`HttpRequest`](#), voir la [documentation des requêtes et réponses](#).

Après le vote d'une personne dans une question, la vue `vote()` redirige vers la page de résultats de la question. Écrivons cette vue :

```
polls/views.py
```

```
from django.shortcuts import get_object_or_404, render
```

```
def results(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/results.html', {'question': question})
```

C'est presque exactement la même que la vue `detail()` du [tutoriel 3](#). La seule différence est le nom du gabarit. Nous éliminerons cette redondance plus tard.

Écrivons maintenant le gabarit `polls/results.html` :

`polls/templates/polls/results.html`

```
<h1>{{ question.question_text }}</h1>

<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|
pluralize }}</li>
{% endfor %}
</ul>

<a href="{% url 'polls:detail' question.id %}">Vote again?</a>
```

Maintenant, rendez-vous à la page `/polls/1/` avec votre navigateur et votez pour la question proposée. Vous devriez voir une page de résultats qui sera mise à jour à chaque fois que vous voterez. Si vous validez le formulaire sans avoir coché votre choix, vous devriez voir le message d'erreur.

Note

Le code de notre vue `vote()` présente un petit problème. Il obtient d'abord l'objet `selected_choice` depuis la base de données, puis calcule la nouvelle valeur de `votes`, et enregistre ensuite le résultat dans la base de données. Si deux utilisateurs du site Web essaient de voter *exactement au même moment*, cela peut mal se passer : la même valeur, disons 42, sera obtenue pour `votes`. Puis, la nouvelle valeur calculée sera de 43 pour les deux utilisateurs qui enregistreront cette valeur, alors qu'elle devrait être de 44 au final.

On appelle cela une *situation de compétition*. Si cela vous intéresse, vous pouvez lire [Prévention des conflits de concurrence avec FQ](#) pour savoir comment il est possible d'éviter ce genre de situations.

Utilisation des vues génériques : moins de code, c'est mieux

Les vues `detail()` (cf. [tutoriel 3](#)) et `results()` sont triviales – et comme mentionné précédemment, redondantes. La vue `index()` qui affiche une liste de sondages est similaire.

Ces vues représentent un cas classique du développement Web : récupérer les données depuis la base de données suivant un paramètre contenu dans l'URL, charger un gabarit et renvoyer le gabarit interprété. Ce cas est tellement classique que Django propose un raccourci, appelé le système de « vues génériques ».

Les vues génériques permettent l'abstraction de pratiques communes, à un tel point que vous n'avez pas à écrire de code Python pour écrire une application.

Nous allons convertir notre application de sondage pour qu'elle utilise le système de vues génériques. Nous pourrions ainsi supprimer une partie de notre code. Nous avons juste quelques pas à faire pour faire cette conversion. Nous allons :

1. Convertir l'URLconf.
2. Supprimer quelques anciennes vues désormais inutiles.
3. Introduire de nouvelles vues basées sur les vues génériques de Django.

Lisez la suite pour plus de détails.

Pourquoi ces changements de code ?

En général, lorsque vous écrivez une application Django, vous devez estimer si les vues génériques correspondent bien à vos besoins et, le cas échéant, vous les utiliserez dès le début, plutôt que de réarranger votre code à mi-chemin. Mais ce tutoriel s'est concentré intentionnellement sur l'écriture des vues « à la dure » jusqu'à ce point, pour mettre l'accent sur les concepts de base.

Tout comme vous devez posséder des bases de maths avant de commencer à utiliser une calculatrice.

Correction de l'URLconf

Tout d'abord, ouvrez la configuration d'URL `polls/urls.py` et modifiez-la ainsi :

`polls/urls.py`

```
from django.conf.urls import url

from . import views

app_name = 'polls'
urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as_view(), name='detail'),
    url(r'^(?P<pk>[0-9]+)/results/$', views.ResultsView.as_view(), name='results'),
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

Notez que le nom du motif correspondant dans les expressions régulières des deuxième et troisième motifs a été modifié de `<question_id>` en `<pk>`.

Correction des vues

Ensuite, nous allons enlever les anciennes vues `index`, `detail` et `results` et utiliser à la place des vues génériques de Django. Pour cela, ouvrez le fichier `polls/views.py` et modifiez-le de cette façon :

`polls/views.py`

```

from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.views import generic

from .models import Choice, Question

class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question.objects.order_by('-pub_date')[:5]

class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/detail.html'

class ResultsView(generic.DetailView):
    model = Question
    template_name = 'polls/results.html'

def vote(request, question_id):
    ... # same as above, no changes needed.

```

Nous utilisons ici deux vues génériques : [ListView](#) et [DetailView](#). Respectivement, ces deux vues permettent l'abstraction des concepts « afficher une liste d'objets » et « afficher une page détaillée pour un type particulier d'objet ».

- Chaque vue générique a besoin de connaître le modèle sur lequel elle va agir. Cette information est fournie par l'attribut `model`.
- La vue générique [DetailView](#) s'attend à ce que la clé primaire capturée dans l'URL s'appelle "pk", nous avons donc changé `question_id` en `pk` pour les vues génériques.

Par défaut, la vue générique [DetailView](#) utilise un gabarit appelé `<nom app>/<nom modèle>_detail.html`. Dans notre cas, elle utiliserait le gabarit `"polls/question_detail.html"`. L'attribut `template_name` est utilisé pour signifier à Django d'utiliser un nom de gabarit spécifique plutôt que le nom de gabarit par défaut. Nous avons aussi indiqué le paramètre `template_name` pour la vue de liste `results`, ce qui permet de différencier l'apparence du rendu des vues « results » et « detail », même s'il s'agit dans les deux cas de vues [DetailView](#) à la base.

De la même façon, la vue générique [ListView](#) utilise par défaut un gabarit appelé `<nom app>/<nom modèle>_list.html` ; nous utilisons `template_name` pour indiquer à [ListView](#) d'utiliser notre gabarit existant `"polls/index.html"`.

Dans les parties précédentes de ce tutoriel, les templates ont été renseignés avec un contexte qui contenait les variables de contexte `question` et `latest_question_list`. Pour `DetailView`, la variable `question` est fournie automatiquement ; comme nous utilisons un modèle nommé `Question`, Django sait donner un nom approprié à la variable de contexte. Cependant, pour `ListView`, la variable de contexte générée automatiquement s'appelle `question_list`. Pour changer cela, nous fournissons l'attribut `context_object_name` pour indiquer que nous souhaitons plutôt la nommer `latest_question_list`. Il serait aussi possible de modifier les templates en utilisant les nouveaux nom de variables par défaut, mais il est beaucoup plus simple d'indiquer à Django les noms de variables que nous souhaitons.

Lancez le serveur et utilisez votre nouvelle application de sondage basée sur les vues génériques.

Pour plus de détails sur les vues génériques, voir la [documentation des vues génériques](#).

Lorsque vous êtes à l'aise avec les formulaires et les vues génériques, lisez la [5ème partie de ce tutoriel](#) pour apprendre comment tester notre application de sondage.

[Écriture de votre première application Django, 3ème partie](#)

[Écriture de votre première application Django, 5ème partie](#)

Écriture de votre première application Django, 5ème partie

Ce tutoriel commence là où le [tutoriel 4](#) s'est achevé. Nous avons construit une application Web de sondage et nous allons maintenant créer quelques tests automatisés pour cette application.

Introduction aux tests automatisés

Que sont les tests automatisés ?

Les tests sont de simples routines qui vérifient le fonctionnement de votre code.

Les tests peuvent se faire à différents niveaux. Certains tests s'appliquent à un petit détail (*est-ce que tel modèle renvoie les valeurs attendues ?*), alors que d'autres examinent le fonctionnement global du logiciel (*est-ce qu'une suite d'actions d'un utilisateur sur le site produit le résultat désiré ?*). C'est le même genre de test qui a été pratiqué précédemment dans le [tutoriel 2](#), en utilisant le [shell](#) pour examiner le comportement d'une méthode ou en lançant l'application et en saisissant des données pour contrôler son comportement.

Ce qui est différent dans les tests *automatisés*, c'est que le travail du test est fait pour vous par le système. Vous créez une seule fois un ensemble de tests, puis au fur et à mesure des modifications de votre application, vous pouvez contrôler que votre code fonctionne toujours tel qu'il devrait, sans devoir effectuer des tests manuels fastidieux.

Pourquoi faut-il créer des tests

Ainsi donc, pourquoi créer des tests, et pourquoi maintenant ?

Vous pouvez penser que vous avez déjà assez de chats à fouetter en apprenant Python/Django et que rajouter encore une nouvelle chose à apprendre et à faire est superflu et inutile. Après tout, notre application de sondage fonctionne maintenant à satisfaction ; se préoccuper encore de créer des tests automatisés ne va pas l'améliorer. Si la création de l'application de sondage est la dernière œuvre de programmation Django que vous entreprenez, alors oui, vous pouvez vous passer d'apprendre à créer des tests automatisés. Mais si ce n'est pas le cas, alors c'est maintenant l'occasion d'apprendre cela.

Les tests vous feront gagner du temps

Jusqu'à un certain point, « contrôler que cela fonctionne apparemment » est un test satisfaisant. Dans une application plus sophistiquée, vous pouvez rencontrer des dizaines d'interactions complexes entre les différents composants.

Une modification dans n'importe lequel de ces composants pourrait avoir des conséquences inattendues sur le comportement de l'application. Contrôler que « cela semble marcher » pourrait signifier la vérification de vingt combinaisons différentes de données de test simplement pour être sûr que vous n'avez rien cassé - vous avez certainement mieux à faire.

C'est particulièrement vrai alors que des tests automatisés pourraient faire cela pour vous en quelques secondes. Si quelque chose se passe mal, les tests vous aideront à identifier le code qui produit le comportement inapproprié.

Parfois, le fait de laisser de côté votre productif et créatif travail de programmation pour affronter la tâche ingrate et peu motivante de l'écriture de tests, peut ressembler à une corvée, spécialement lorsque vous savez que votre code fonctionne correctement.

Cependant, l'écriture de tests est bien plus rentable que de passer des heures à tester manuellement votre application ou à essayer d'identifier la cause d'un problème récemment découvert.

Les tests ne font pas qu'identifier les problèmes, ils les préviennent

C'est une erreur de penser que les tests ne sont que l'aspect négatif du développement.

Sans tests, le but ou le comportement attendu d'une application pourrait rester obscur. Même quand il s'agit de votre propre code, vous vous retrouverez parfois à fouiner un peu partout pour retrouver ce qu'il fait au juste.

Les tests changent cela ; ils éclairent votre code de l'intérieur et lorsque quelque chose va de travers, ils mettent en relief la partie qui coince, *même lorsque vous n'avez même pas réalisé que quelque chose n'allait pas*.

Les tests rendent votre code plus attractif

Vous avez peut-être créé un bout de logiciel extraordinaire, mais vous constaterez que beaucoup d'autres développeurs vont simplement refuser de l'examiner parce qu'il ne contient pas de tests ; sans tests, ils ne lui font pas confiance. Jacob Kaplan-Moss, l'un des développeurs initiaux de Django, a dit : « Du code sans tests est par définition du code cassé ».

Le fait que d'autres développeurs veulent voir des tests dans votre logiciel avant de le prendre au sérieux est une raison supplémentaire de commencer l'écriture de tests.

Les tests aident les équipes à travailler ensemble

Les points précédents sont écrits du point de vue d'un développeur isolé maintenant une application. Les applications complexes sont maintenues par des équipes. Les tests garantissent que les collègues ne cassent votre code par mégarde (et aussi que vous ne cassiez le leur sans le vouloir). Si vous voulez gagner votre vie en tant que programmeur Django, vous devez être bon dans l'écriture de tests !

Stratégies élémentaires pour les tests

Il existe de nombreuses approches pour écrire des tests.

Certains programmeurs suivent une discipline appelée « développement piloté par les tests » (« [test-driven development](#) »). Ils écrivent les tests avant de commencer à écrire le code. Cela peut paraître illogique, mais c'est un processus très semblable à ce que la plupart des gens font : ils décrivent un problème, puis ils écrivent du code pour le résoudre. Le développement piloté par les tests formalise simplement le problème dans un cas de test Python.

Plus souvent, un débutant dans les tests va créer du code, puis il décidera plus tard qu'il devrait ajouter des tests. Il aurait peut-être été préférable d'écrire des tests plus tôt, mais il n'est jamais trop tard pour commencer.

Il est parfois difficile de trouver où commencer en écrivant des tests. Si vous avez écrit plusieurs milliers de lignes de code Python, choisir quoi tester n'est pas simple. Dans un tel cas, il peut être utile d'écrire le premier test au moment où vous effectuez votre prochaine modification, soit pour ajouter une nouvelle fonctionnalité ou pour corriger un bogue.

Entrons dans le vif du sujet.

Écriture du premier test

Un bogue a été trouvé

Heureusement, il y a un petit bogue dans l'application `polls`, tout prêt à être corrigé : la méthode `Question.was_published_recently()` renvoie `True` si l'objet `Question` a été publié durant le jour précédent (ce qui est correct), mais également si le champ `pub_date` de `Question` est dans le futur (ce qui n'est évidemment pas juste).

Pour vérifier que le bogue existe réellement, créez une question avec une date dans le futur sur le site d'administration et testez la méthode en utilisant le [shell](#):

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the future
>>> future_question = Question(pub_date=timezone.now() +
datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.was_published_recently()
True
```

Étant donné que ce qui est dans le futur n'est pas « récent », c'est clairement une erreur.

Création d'un test pour révéler le bogue

Ce que nous venons de faire dans le [shell](#) pour tester le problème est exactement ce que nous pouvons faire dans un test automatisé ; transformons cette opération en un test automatisé.

Un endroit conventionnel pour placer les tests d'une application est le fichier `tests.py` dans le répertoire de l'application. Le système de test va automatiquement trouver les tests dans tout fichier dont le nom commence par `test`.

Placez ce qui suit dans le fichier `tests.py` de l'application `polls`:

```
polls/tests.py
```

```
import datetime
```

```
from django.utils import timezone
from django.test import TestCase
```

```
from .models import Question
```

```
class QuestionMethodTests(TestCase):
```

```
    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() should return False for questions whose
        pub_date is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
```

```
future_question = Question(pub_date=time)
self.assertEqual(future_question.was_published_recently(), False)
```

Nous venons ici de créer une sous-classe de [django.test.TestCase](#) contenant une méthode qui crée une instance `Question` en renseignant `pub_date` dans le futur. Nous vérifions ensuite le résultat de `was_published_recently()` qui *devrait* valoir `False`.

Lancement des tests

Dans le terminal, nous pouvons lancer notre test :

```
$ python manage.py test polls
```

et vous devriez voir quelque chose comme :

```
Creating test database for alias 'default'...
F
=====
FAIL: test_was_published_recently_with_future_question
(polls.tests.QuestionMethodTests)
-----
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in
test_was_published_recently_with_future_question
    self.assertEqual(future_question.was_published_recently(), False)
AssertionError: True != False
-----
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Voici ce qui s'est passé :

- La commande `python manage.py test polls` a cherché des tests dans l'application `polls` ;
- elle a trouvé une sous-classe de [django.test.TestCase](#) ;
- elle a créé une base de données spéciale uniquement pour les tests ;
- elle a recherché des méthodes de test, celles dont le nom commence par `test` ;
- dans `test_was_published_recently_with_future_question`, elle a créé une instance `Question` dont le champ `pub_date` est 30 jours dans le futur ;
- ... et à l'aide de la méthode `assertEqual()`, elle a découvert que sa méthode `was_published_recently()` renvoyait `True`, alors que nous souhaitons qu'elle renvoie `False`.

Le test nous indique le nom du test qui a échoué ainsi que la ligne à laquelle l'échec s'est produit.

Correction du bogue

Nous connaissons déjà le problème : `Question.was_published_recently()` devrait renvoyer `False` si sa `pub_date` est dans le futur. Corrigez la méthode dans `models.py` afin qu'elle ne renvoie `True` que si la date est aussi dans le passé :

`polls/models.py`

```
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

puis lancez à nouveau le test :

```
Creating test database for alias 'default'...
```

```
.
```

```
-----
Ran 1 test in 0.001s
```

```
OK
```

```
Destroying test database for alias 'default'...
```

Après avoir identifié un bogue, nous avons écrit un test qui le révèle et nous avons corrigé l'erreur dans le code pour que notre test réussisse.

Bien d'autres choses pourraient aller de travers avec notre application à l'avenir, mais nous pouvons être sûrs que nous n'allons pas réintroduire cette erreur par mégarde, car en lançant le test, nous serons immédiatement avertis. Nous pouvons considérer que cette petite partie de l'application est assurée de rester fonctionnelle pour toujours.

Des tests plus exhaustifs

Pendant que nous y sommes, nous pouvons assurer un peu plus le fonctionnement de la méthode `was_published_recently()` ; en fait, il serait réellement embarrassant si en corrigeant un bogue, nous en avons introduit un autre.

Ajoutez deux méthodes de test supplémentaires dans la même classe pour tester le comportement de la méthode de manière plus complète :

`polls/tests.py`

```
def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() should return False for questions whose
    pub_date is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=30)
    old_question = Question(pub_date=time)
    self.assertEqual(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() should return True for questions whose
```

```
pub_date is within the last day.
"""
time = timezone.now() - datetime.timedelta(hours=1)
recent_question = Question(pub_date=time)
self.assertEqual(recent_question.was_published_recently(), True)
```

Et maintenant nous disposons de trois tests qui confirment que `Question.was_published_recently()` renvoie des valeurs correctes pour des questions passées, récentes et futures.

Encore une fois, `polls` est une application simple, mais quelle que soit la complexité de son évolution ou le code avec lequel elle devra interagir, nous avons maintenant une certaine garantie que la méthode pour laquelle nous avons écrit des tests se comportera de façon cohérente.

Test d'une vue

L'application de sondage est peu regardante : elle publiera toute question, y compris celles dont le champ `pub_date` est situé dans le futur. C'est à améliorer. Définir `pub_date` dans le futur devrait signifier que la question sera publiée à ce moment, mais qu'elle ne doit pas être visible avant cela.

Un test pour une vue

En corrigeant le bogue ci-dessus, nous avons d'abord écrit le test, puis le code pour le corriger. En fait, c'était un exemple simple de développement piloté par les tests, mais l'ordre dans lequel se font les choses n'est pas fondamental.

Dans notre premier test, nous nous sommes concentrés étroitement sur le fonctionnement interne du code. Pour ce test, nous voulons contrôler son comportement tel qu'il se déroulerait avec un utilisateur depuis son navigateur.

Avant d'essayer de corriger quoi que ce soit, examinons les outils à notre disposition.

Le client de test de Django

Django fournit un [Client](#) de test pour simuler l'interaction d'un utilisateur avec le code au niveau des vues. On peut l'utiliser dans `tests.py` ou même dans le [shell](#).

Nous commencerons encore une fois par le [shell](#), où nous devons faire quelques opérations qui ne seront pas nécessaires dans `tests.py`. La première est de configurer l'environnement de test dans le [shell](#):

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

[setup_test_environment\(\)](#) installe un moteur de rendu de gabarit qui va nous permettre d'examiner certains attributs supplémentaires des réponses, tels que `response.context` qui n'est normalement pas disponible. Notez que cette méthode *ne crée pas* de base de données de test, ce qui

signifie que ce qui suit va être appliqué à la base de données existante et que par conséquent, le résultat peut légèrement différer en fonction des questions que vous avez déjà créées.

Ensuite, il est nécessaire d'importer la classe `Client` de test (plus loin dans `tests.py`, nous utiliserons la classe [django.test.TestCase](#) qui apporte son propre client, ce qui évitera cette étape) :

```
>>> from django.test import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

Ceci fait, nous pouvons demander au client de faire certaines tâches pour nous :

```
>>> # get a response from '/'
>>> response = client.get('/')
>>> # we should expect a 404 from that address
>>> response.status_code
404
>>> # on the other hand we should expect to find something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded URL
>>> from django.core.urlresolvers import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n\n\n    <p>No polls are available.</p>\n\n'
>>> # note - you might get unexpected results if your ``TIME_ZONE``
>>> # in ``settings.py`` is not correct. If you need to change it,
>>> # you will also need to restart your shell session
>>> from polls.models import Question
>>> from django.utils import timezone
>>> # create a Question and save it
>>> q = Question(question_text="Who is your favorite Beatle?",
pub_date=timezone.now())
>>> q.save()
>>> # check the response once again
>>> response = client.get('/polls/')
>>> response.content
b'\n\n\n    <ul>\n        \n        <li><a href="/polls/1/">Who is your favorite
Beatle?</a></li>\n    </ul>\n\n'
>>> # If the following doesn't work, you probably omitted the call to
>>> # setup_test_environment() described above
>>> response.context['latest_question_list']
[<Question: Who is your favorite Beatle?>]
```

Amélioration de la vue

La liste des sondages montre aussi les sondages pas encore publiés (c'est-à-dire ceux dont le champ `pub_date` est dans le futur). Corrigions cela.

Dans le [tutoriel 4](#), nous avons introduit une vue basée sur la classe [ListView](#):

`polls/views.py`

```
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'
```

```
def get_queryset(self):
    """Return the last five published questions."""
    return Question.objects.order_by('-pub_date')[:5]
```

Nous devons corriger la méthode `get_queryset()` pour qu'elle vérifie aussi la date en la comparant avec `timezone.now()`. Nous devons d'abord ajouter une importation :

`polls/views.py`

```
from django.utils import timezone
```

puis nous devons corriger la méthode `get_queryset` de cette façon :

`polls/views.py`

```
def get_queryset(self):
    """
    Return the last five published questions (not including those set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

`Question.objects.filter(pub_date__lte=timezone.now())` renvoie un queryset contenant les questions dont le champ `pub_date` est plus petit ou égal (c'est-à-dire plus ancien ou égal) à `timezone.now`.

Test de la nouvelle vue

Vous pouvez maintenant vérifier vous-même que tout fonctionne comme prévu en lançant `runserver` et en accédant au site depuis votre navigateur. Créez des questions avec des dates dans le passé et dans le futur et vérifiez que seuls celles qui ont été publiées apparaissent dans la liste. Mais vous ne voulez pas faire ce travail de test manuel *chaque fois que vous effectuez une modification qui pourrait affecter ce comportement*, créons donc aussi un test basé sur le contenu de notre session [shell](#) précédente.

Ajoutez ce qui suit à `polls/tests.py`:

`polls/tests.py`

```
from django.core.urlresolvers import reverse
```

et nous allons créer une fonction raccourci pour créer des questions, ainsi qu'une nouvelle classe de test :

`polls/tests.py`

```
def create_question(question_text, days):
    """
    Creates a question with the given `question_text` and published the
```

```

given number of `days` offset to now (negative for questions published
in the past, positive for questions that have yet to be published).
"""
time = timezone.now() + datetime.timedelta(days=days)
return Question.objects.create(question_text=question_text, pub_date=time)

```

```

class QuestionViewTests(TestCase):
    def test_index_view_with_no_questions(self):
        """
        If no questions exist, an appropriate message should be displayed.
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_a_past_question(self):
        """
        Questions with a pub_date in the past should be displayed on the
        index page.
        """
        create_question(question_text="Past question.", days=-30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_a_future_question(self):
        """
        Questions with a pub_date in the future should not be displayed on
        the index page.
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        should be displayed.
        """
        create_question(question_text="Past question.", days=-30)
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_two_past_questions(self):
        """
        The questions index page may display multiple questions.
        """
        create_question(question_text="Past question 1.", days=-30)
        create_question(question_text="Past question 2.", days=-5)
        response = self.client.get(reverse('polls:index'))

```

```

self.assertQuerysetEqual(
    response.context['latest_question_list'],
    ['<Question: Past question 2.>', '<Question: Past question 1.>']
)

```

Examinons plus en détails certaines de ces méthodes.

Tout d’abord, la fonction raccourci `create_question` permet d’éviter de répéter plusieurs fois le processus de création de questions.

`test_index_view_with_no_questions` ne crée aucune question, mais vérifie le message : « No polls are available. » et que la liste `latest_question_list` est vide. Notez que la classe [django.test.TestCase](#) fournit quelques méthodes d’assertion supplémentaires. Dans ces exemples, nous utilisons [assertContains\(\)](#) et [assertQuerysetEqual\(\)](#).

Dans `test_index_view_with_a_past_question`, nous créons une question et vérifions qu’elle apparaît dans la liste.

Dans `test_index_view_with_a_future_question`, nous créons une question avec `pub_date` dans le futur. La base de données est réinitialisée pour chaque méthode de test, ce qui explique que la première question n’est plus disponible et que la page d’index n’affiche plus de question.

Et ainsi de suite. En pratique, nous utilisons les tests pour raconter des histoires d’interactions entre des saisies dans l’interface d’administration et d’un utilisateur parcourant le site, en contrôlant qu’à chaque état ou changement d’état du système, les résultats attendus apparaissent.

Test de DetailView

Le code marche bien maintenant. Cependant, même si les questions futures n’apparaissent pas sur la page *index*, les utilisateurs peuvent toujours y accéder s’ils savent ou devinent la bonne URL. Nous avons donc besoin d’une contrainte semblable dans `DetailView` :

`polls/views.py`

```

class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excludes any questions that aren't published yet.
        """
        return Question.objects.filter(pub_date__lte=timezone.now())

```

Et bien évidemment, nous ajoutons quelques tests, pour contrôler qu’une question dont `pub_date` est dans le passé peut être affichée, mais que si `pub_date` est dans le futur, elle ne le sera pas :

`polls/tests.py`

```

class QuestionIndexDetailTests(TestCase):
    def test_detail_view_with_a_future_question(self):

```



```

"""
The detail view of a question with a pub_date in the future should
return a 404 not found.
"""
future_question = create_question(question_text='Future question.', days=5)
url = reverse('polls:detail', args=(future_question.id,))
response = self.client.get(url)
self.assertEqual(response.status_code, 404)

def test_detail_view_with_a_past_question(self):
    """
    The detail view of a question with a pub_date in the past should
    display the question's text.
    """
    past_question = create_question(question_text='Past Question.', days=-5)
    url = reverse('polls:detail', args=(past_question.id,))
    response = self.client.get(url)
    self.assertContains(response, past_question.question_text)

```

Idées pour d'autres tests

Nous devrions ajouter une méthode `get_queryset` similaire à `ResultsView` et créer une nouvelle classe de test pour cette vue. Elle sera très semblable à celle que nous venons de créer ; en fait, il y aura beaucoup de répétition.

Nous pourrions aussi améliorer notre application d'autres manières, en ajoutant des tests au fur et à mesure. Par exemple, il est stupide de pouvoir publier des questions sur le site sans choix. Nos vues pourraient donc vérifier cela et exclure de telles questions. Les tests créeraient une question sans choix et testeraient qu'elle n'est pas publiée ; de même, il s'agirait de créer une question *avec* des choix et tester qu'elle *est* bien publiée.

Peut-être que les utilisateurs connectés dans l'interface d'administration pourraient être autorisés à voir les questions non publiées, mais pas les visiteurs ordinaires. C'est toujours le même principe, tout ce qui est ajouté au logiciel devrait être accompagné par un test, que ce soit en écrivant d'abord le test puis en écrivant le code pour réussir le test, ou en travaillant d'abord sur la logique du code et en écrivant ensuite le test pour prouver le bon fonctionnement.

À un certain stade, vous allez regarder vos tests et vous demander si votre code ne souffre pas de surabondance de tests, ce qui nous amène à :

Pour les tests, abondance de biens ne nuit pas

En apparence, les tests peuvent avoir l'air de croître sans limites. À ce rythme, il y aura bientôt plus de code dans nos tests que dans notre application ; et la répétition est laide, comparée à la brièveté élégante du reste du code.

Ça n'a aucune importance. Laissez-les grandir. Dans la plupart des cas, vous pouvez écrire un test une fois et ne plus y penser. Il continuera à jouer son précieux rôle tout au long du développement de votre programme.

Les tests devront parfois être mis à jour. Supposons que nous corrigeons nos vues pour que seules les questions avec choix soient publiées. Dans ce cas, de nombreux tests existants vont échouer, *ce qui nous informera exactement au sujet des tests qui devront être mis à jour* ; dans cette optique, on peut dire que les tests prennent soin d’eux-même.

Au pire, au cours du développement, vous pouvez constater que certains tests deviennent redondants. Ce n’est même pas un problème. Dans les tests, la redondance est une *bonne* chose.

Tant que vos tests sont logiquement disposés, ils ne deviendront pas ingérables. Quelques bons principes à garder en tête :

- une classe de test séparée pour chaque modèle ou vue ;
- une méthode de test séparée pour chaque ensemble de conditions que vous voulez tester ;
- des noms de méthodes de test qui indiquent ce qu’elles font.

Encore plus de tests

Ce tutoriel ne fait qu’introduire à quelques concepts de base des tests. Il y a encore beaucoup plus à faire et d’autres outils très utiles à votre disposition pour effectuer des choses plus perfectionnées.

Par exemple, bien que les tests présentés ici couvrent une partie de la logique interne d’un modèle et la manière dont les vues publient de l’information, vous pouvez utiliser un système basé sur de vrais navigateurs comme [Selenium](#) pour tester la façon dont le code HTML est vraiment rendu dans un navigateur. Ces outils permettent de tester plus que le comportement du seul code Django, mais aussi, par exemple, du code JavaScript. C’est assez impressionnant de voir les tests lancer un navigateur et commencer d’interagir avec votre site, comme si un être humain invisible le pilotait ! Django propose la classe [LiveServerTestCase](#) pour faciliter l’intégration avec des outils comme Selenium.

Si votre application est complexe, il peut être utile de lancer automatiquement les tests lors de chaque commit dans l’optique d’une [intégration continue](#)), afin que le contrôle qualité puisse être lui-même automatisé, au moins partiellement.

Une bonne façon de détecter des parties d’application non couvertes par les tests est de contrôler la couverture du code. Cela aide aussi à identifier du code fragile, ou même mort. Si vous ne pouvez pas tester un bout de code, cela signifie généralement que le code doit être réarrangé ou supprimé. Le taux de couverture aide à identifier le code inutilisé. Consultez [Intégration avec coverage.py](#) pour plus de détails.

[Django et les tests](#) contient des informations complètes au sujet des tests.

Et ensuite ?

Pour des détails complets sur les tests, consultez [Django et les tests](#).

Lorsque vous êtes à l'aise avec les tests de vues Django, lisez la [6ème partie de ce tutoriel](#) pour en savoir plus sur la gestion des fichiers statiques.

[Écriture de votre première application Django, 4ème partie](#)
[Écriture de votre première application Django, 6ème partie](#)

Écriture de votre première application Django, 6ème partie

Ce tutoriel commence là où le [tutoriel 5](#) s'est achevé. Nous avons construit une application Web de sondage et nous allons maintenant ajouter une feuille de style et une image.

En plus du code HTML généré par le serveur, les applications Web doivent généralement servir des fichiers supplémentaires tels que des images, du JavaScript ou du CSS, utiles pour produire une page Web complète. Dans Django, on appelle ces fichiers des « fichiers statiques ».

Pour de petits projets, ce n'est pas un grand problème car il est possible de simplement placer les fichiers statiques à un endroit où le serveur Web peut les trouver. Cependant, dans des plus gros projets, surtout pour ceux qui contiennent plusieurs applications, la gestion de plusieurs groupes de fichiers statiques fournis par chaque application commence à devenir plus complexe.

C'est le travail de `django.contrib.staticfiles` : il collecte les fichiers statiques de chaque application (et de tout autre endroit que vous lui indiquez) pour les mettre dans un seul emplacement qui peut être facilement configuré pour servir les fichiers en production.

Personnalisation de l'apparence de votre application

Commencez par créer un répertoire nommé `static` dans votre répertoire `polls`. C'est là que Django va chercher les fichiers statiques, sur le même principe que la recherche de gabarits dans `polls/templates/`.

Le réglage [STATICFILES_FINDERS](#) de Django contient une liste de classes qui savent où aller chercher les fichiers statiques de différentes sources. Une de ces classes par défaut est `AppDirectoriesFinder` qui recherche un sous-répertoire « `static` » dans chaque application de [INSTALLED_APPS](#), comme l'application `polls` de notre tutoriel. Le site d'administration utilise la même structure de répertoires pour ses propres fichiers statiques.

À l'intérieur du répertoire `static` que vous venez de créer, créez un autre répertoire nommé `polls` dans lequel vous placez un fichier `style.css`. Autrement dit, votre feuille de style devrait se trouver

à `polls/static/polls/style.css`. En raison du fonctionnement de `AppDirectoriesFinder`, vous pouvez vous référer à ce fichier depuis Django avec la syntaxe `polls/style.css`, sur le même principe utilisé pour se référer aux chemins de gabarits.

Espaces de noms des fichiers statiques

Tout comme pour les gabarits, nous *pourrions* plus simplement placer nos fichiers statiques directement dans `polls/static` (au lieu de créer un sous-répertoire `polls`), mais ce serait une mauvaise idée. Django choisit le premier fichier statique trouvé correspondant au nom recherché, et si vous aviez un fichier de même nom dans une *autre* application, Django ne pourrait pas les distinguer. Nous devons pouvoir indiquer à Django le bon fichier et le moyen le plus simple de s'en assurer est d'utiliser les *espaces de noms*. C'est-à-dire en plaçant ces fichiers statiques dans un *autre* sous-répertoire nommé d'après l'application.

Écrivez le code suivant dans cette feuille de style (`polls/static/polls/style.css`):

```
polls/static/polls/style.css
```

```
li a {  
    color: green;  
}
```

Ensuite, ajoutez le contenu ci-dessous au début de `polls/templates/polls/index.html`:

```
polls/templates/polls/index.html
```

```
{% load staticfiles %}
```

```
<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}" />
```

`{% load staticfiles %}` charge la balise de gabarit `{% static %}` qui provient de la bibliothèque de balises de gabarit de `staticfiles`. La balise de gabarit `{% static %}` génère l'URL absolue du fichier statique.

C'est tout ce que vous avez à faire pour le développement. Rechargez

`http://localhost:8000/polls/` et vous devriez voir que les liens des questions sont verts (style Django !) ce qui signifie que votre feuille de style a été correctement chargée.

Ajout d'une image d'arrière-plan

Ensuite, nous allons créer un sous-répertoire pour les images. Créez un sous-répertoire `images` dans le répertoire `polls/static/polls/`. Dans le répertoire créé, placez une image nommée `background.gif`. Autrement dit, le chemin de votre image sera `polls/static/polls/images/background.gif`.

Puis, ajoutez ceci à votre feuille de style (`polls/static/polls/style.css`):

```
polls/static/polls/style.css
```

```
body {  
    background: white url("images/background.gif") no-repeat right bottom;  
}
```

Rechargez `http://localhost:8000/polls/` et vous devriez voir l'arrière-plan chargé dans le coin inférieur droit de l'écran.

Avertissement

Bien sûr, la balise de gabarit `{% static %}` n'est pas disponible dans les fichiers statiques comme votre feuille de style, puisqu'ils ne sont pas générés par Django. Vous devriez toujours utiliser des **chemins relatifs** pour lier vos fichiers statiques entre eux, car vous pouvez ensuite modifier [STATIC_URL](#) (utilisé par la balise de gabarit `static` pour générer les URL) sans devoir modifier tous les chemins dans les fichiers statiques.

Ce sont les **bases**. Pour plus de détails sur les réglages et les autres fonctionnalités incluses dans le module, lisez le [manuel des fichiers statiques](#) et la [référence des fichiers statiques](#). La section [déploiement des fichiers statiques](#) aborde l'utilisation des fichiers statiques sur un vrai serveur.

Lorsque vous serez à l'aise avec les fichiers statiques, lisez la [partie 7 de ce tutoriel](#) pour apprendre comment personnaliser l'interface d'administration automatique de Django.

[Écriture de votre première application Django, 5ème partie](#)

[Écriture de votre première application Django, 7ème partie](#)

Écriture de votre première application Django, 7ème partie

This tutorial begins where [Tutorial 6](#) left off. We're continuing the Web-poll application and will focus on customizing Django's automatically-generated admin site that we first explored in [Tutorial 2](#).

Personnalisation du formulaire d'administration

Quand vous avez inscrit le modèle `Question` avec `admin.site.register(Question)`, Django a été capable de représenter l'objet dans un formulaire par défaut. Il sera cependant souvent souhaitable de personnaliser l'affichage et le comportement du formulaire. Cela se fait en indiquant certaines options à Django lors de l'inscription de l'objet.

Voyons comment cela fonctionne, en réordonnant les champs sur le formulaire d'édition. Remplacez la ligne `admin.site.register(Question)` par :

```
polls/admin.py
```

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

```
from .models import Question

class QuestionAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question_text']

admin.site.register(Question, QuestionAdmin)
```

You'll follow this pattern – create a model admin class, then pass it as the second argument to `admin.site.register()` – any time you need to change the admin options for a model.

Cette modification fait que la « Date de publication » apparaît avant le champ « Question » :

Ce n'est pas spécialement impressionnant avec seulement deux champs, mais pour un formulaire d'administration avec des dizaines de champs, choisir un ordre intuitif est un détail d'utilisation important.

Et en parlant de formulaires avec des dizaines de champs, il peut être utile de partager le formulaire en plusieurs sous-ensembles (fieldsets) :

polls/admin.py

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

class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Question, QuestionAdmin)
```

Le premier élément de chaque tuple dans `fieldsets` est le titre du groupe de champs. Voici ce à quoi notre formulaire ressemble à présent :

Home › Polls › Questions › What's up?

Change question

Question text:

What's up?

Date information

Date published:


Date:

2015-09-06

Today | 

Time:

21:16:20

Now | 

Ajout d'objets liés

OK, nous avons notre page d'administration des questions. Mais une `Question` possède plusieurs choix `Choices`, et la page d'administration n'affiche aucun choix.

Pour le moment.

Il y a deux façons de résoudre ce problème. La première est d'inscrire `Choice` dans l'administration, comme nous l'avons fait pour `Question`. Voici ce que ça donnerait :

`polls/admin.py`

```
from django.contrib import admin

from .models import Choice, Question
# ...
admin.site.register(Choice)
```

Maintenant les choix “Choices” sont une option disponible dans l'interface d'administration de Django. Le formulaire « Add choice » ressemble à ceci :

Add choice

Question:	<div><div>-----</div><div>⬆ ⬇ ⬆</div><div> </div></div>
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>

Dans ce formulaire, le champ « Question » est une boîte de sélection contenant toutes les questions de la base de données. Django sait qu'une [ForeignKey](#) doit être représentée dans l'interface d'administration par une boîte `<select>`. Dans notre cas, seule une question existe à ce stade.

Notez également le lien « Ajouter un autre » à côté de « Question ». Chaque objet avec une relation `ForeignKey` vers un autre reçoit automatiquement cet outil. Quand vous cliquez sur « Ajouter un autre », vous obtenez une fenêtre surgissante contenant le formulaire « Add Question ». Si vous ajoutez une question dans cette fenêtre et que vous cliquez sur « Enregistrer », Django enregistre la question dans la base de données et l'ajoute dynamiquement comme choix sélectionné dans le formulaire « Add choice » que vous étiez en train de remplir.

Mais franchement, c'est une manière inefficace d'ajouter des objets `Choice` dans le système. Il serait préférable d'ajouter un groupe de choix « Choices » directement lorsque vous créez l'objet `Question`. Essayons de cette façon.

Enlevez l'appel `register()` pour le modèle `Choice`. Puis, modifiez le code d'inscription de `Question` comme ceci :

`polls/admin.py`

```
from django.contrib import admin
from .models import Choice, Question
```

```
class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3
```

```
class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
```



```
]
inline = [ChoiceInline]

admin.site.register(Question, QuestionAdmin)
```

Cela indique à Django : « les objets `Choice` sont édités dans la page d'administration de `Question`. Par défaut, fournir assez de place pour 3 choix ».

Chargez la page « Ajout Question » pour voir à quoi ça ressemble :

Add question

Question text:

Date information (Hide)

Date published:

Date:

Today | 

Time:

Now | 

CHOICES

Choice: #1

Choice text:

Votes:

0

Choice: #2

Choice text:

Votes:

0

Choice: #3

Choice text:

Votes:

0

 Add another Choice

Save and add another

Save and con

Ça marche comme ceci : il y a trois emplacements pour les choix « Choices » liés – comme indiqué par extra – et chaque fois que vous revenez sur la page de modification d’un objet déjà créé, vous obtenez trois emplacements supplémentaires.

Au bas des trois emplacements actuels, vous pouvez trouver un lien « Add another Choice ». Si vous cliquez dessus, un nouvel emplacement apparaît. Si vous souhaitez supprimer l’emplacement ajouté, vous pouvez cliquer sur le X en haut à droite de l’emplacement. Notez que vous ne pouvez pas enlever les trois emplacements de départ. Cette image montre l’emplacement ajouté :

CHOICES	
Choice: #1	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #2	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #3	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #4	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
+ Add another Choice	

Un petit problème cependant. Cela prend beaucoup de place d’afficher tous les champs pour saisir les objets Choice liés. C’est pour cette raison que Django offre une alternative d’affichage en tableau des objets liés ; il suffit de modifier la déclaration ChoiceInline comme ceci :

polls/admin.py

```
class ChoiceInline(admin.TabularInline):  
    #...
```

Avec ce `TabularInline` (au lieu de `StackedInline`), les objets liés sont affichés dans un format plus compact, tel qu'un tableau :

CHOICES	
CHOICE TEXT	VOTES
<input type="text"/>	<input type="text" value="0"/>
<input type="text"/>	<input type="text" value="0"/>
<input type="text"/>	<input type="text" value="0"/>

[+ Add another Choice](#)

[Save and add another](#) [Save and continue](#)

Remarquez la colonne supplémentaire « Delete? » qui permet de supprimer des lignes ajoutées à l'aide du bouton « Add Another Choice » ainsi que les lignes déjà enregistrées.

Personnalisation de la liste pour modification de l'interface d'administration

Maintenant que la page d'administration des `Questions` présente un peu mieux, améliorons la page « liste pour modification » – celle qui affiche toutes les questions du système.

Voici à quoi ça ressemble pour l'instant :

Select question to change

Action: 0 of 1 selected☐ QUESTION TEXT☐ What's up?

1 question

Par défaut, Django affiche le `str()` de chaque objet. Mais parfois, il serait plus utile d'afficher des champs particuliers. Dans ce but, utilisez l'option [list_display](#), qui est un tuple de noms de champs à afficher, en colonnes, sur la page liste pour modification de l'objet :

polls/admin.py

```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date')
```

Juste pour la démonstration, incluons également la méthode `was_published_recently()` du [tutoriel 2](#):

polls/admin.py

```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date', 'was_published_recently')
```

À présent, la page liste pour modification des questions ressemble à ceci :

Select question to change

Action: 0 of 1 selected

<input type="checkbox"/>	QUESTION TEXT	DATE PUBLISHED	WAS PUBLISHED RECENTLY
<input type="checkbox"/>	What's up?	Sept. 3, 2015, 9:16 p.m.	False

1 question

Vous pouvez cliquer sur les en-têtes de colonne pour trier selon ces valeurs – sauf dans le cas de l'en-tête `was_published_today`, parce que le tri selon le résultat d'une méthode arbitraire n'est pas pris en charge. Notez aussi que l'en-tête de la colonne pour `was_published_today` est, par défaut, le nom de la méthode (avec les soulignements remplacés par des espaces) et que chaque ligne contient la représentation textuelle du résultat.

Vous pouvez améliorer cela en donnant à cette méthode (dans `polls/models.py`) quelques attributs, comme ceci :

polls/models.py

```
class Question(models.Model):
    # ...
    def was_published_recently(self):
        now = timezone.now()
        return now - datetime.timedelta(days=1) <= self.pub_date <= now
    was_published_recently.admin_order_field = 'pub_date'
    was_published_recently.boolean = True
    was_published_recently.short_description = 'Published recently?'
```

Pour plus d'informations sur ces propriétés de méthodes, voir [list_display](#).

Modifiez encore une fois le fichier `polls/admin.py` et améliorez la page de liste pour modification des `Questions`: ajout de filtrage à l'aide de l'attribut [list_filter](#). Ajoutez la ligne suivante à `QuestionAdmin`:

```
list_filter = ['pub_date']
```

Cela ajoute une barre latérale « Filter » qui permet de filtrer la liste pour modification selon le champ `pub_date` :

Select question to change

Action: 0 of 1 selected

<input type="checkbox"/>	QUESTION TEXT	DATE PUBLISHED	PUBLISHED RECENTLY?
<input type="checkbox"/>	What's up?	Sept. 3, 2015, 9:16 p.m.	

1 question

FILTER

By date

Any date

Today

Past 7 d

This mon

This year

Le type de filtre affiché dépend du type de champ à filtrer. Comme `pub_date` est un champ [DateTimeField](#), Django sait fournir des options de filtrage appropriées : « Toutes les dates », « Aujourd'hui », « Les 7 derniers jours », « Ce mois-ci », « Cette année ».

Ça a meilleure forme. Ajoutons une fonctionnalité de recherche :

```
search_fields = ['question_text']
```

Cela ajoute une boîte de recherche en haut de la liste pour modification. Quand quelqu'un saisit des termes de recherche, Django va rechercher dans le champ `question_text`. Vous pouvez indiquer autant de champs que vous le désirez – néanmoins, en raison de l'emploi d'une requête `LIKE` en arrière-plan, il s'agit de rester raisonnable quant au nombre de champs de recherche, sinon la base de données risque de tirer la langue !

C'est maintenant le bon moment de noter que les listes pour modification vous laissent une grande liberté de mise en page. Par défaut, 100 éléments sont affichés par page. La [pagination](#) des listes pour modification, les [boîtes de recherche](#), les [filtres](#), les [hiérarchies](#) [calendaires](#) et le [tri selon l'en-tête de colonne](#), tout fonctionne ensemble pour une utilisation optimale.

Personnalisation de l'apparence de l'interface d'administration

Clairement, avoir « Django administration » en haut de chaque page d'administration est ridicule. C'est juste du texte de substitution.

Toutefois, c'est facile à modifier en utilisant le système de gabarits de Django. Le site d'administration de Django est fait en Django lui-même, et ses interfaces utilisent le système de gabarits propre à Django.

Personnalisation des gabarits de votre *projet*

Créez un répertoire s'appelant `templates` dans le répertoire de votre projet (celui qui contient `manage.py`). Les gabarits peuvent se trouver à n'importe quel endroit du système de fichiers, pourvu qu'ils soient accessibles par Django (Django utilise le même utilisateur que celui qui a démarré votre serveur). Cependant, par convention, il est recommandé de conserver les gabarits à l'intérieur du projet.

Ouvrez votre fichier de configuration (`mysite/settings.py`, souvenez-vous) et ajoutez une option [DIRS](#) dans le réglage [TEMPLATES](#):

`mysite/settings.py`

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

[DIRS](#) est une liste de répertoires du système de fichiers que Django parcourt lorsqu'il doit charger les gabarits ; il s'agit d'un chemin de recherche.

Organisation des gabarits

Just like the static files, we *could* have all our templates together, in one big templates directory, and it would work perfectly well. However, templates that belong to a particular application should be placed in that application's template directory (e.g. `polls/templates`) rather than the project's (`templates`). We'll discuss in more detail in the [reusable apps tutorial](#) why we do this.

Créez maintenant dans `templates` un répertoire nommé `admin` et copiez-y le gabarit `admin/base_site.html` à partir du répertoire par défaut des gabarits d'administration dans le code source de Django (`django/contrib/admin/templates`).

Où se trouvent les fichiers sources de Django ?

Si vous ne savez pas où les fichiers source de Django se trouvent sur votre système, lancez la commande suivante :

```
$ python -c "import django; print(django.__path__)"
```


Ensuite, éditez simplement le fichier et remplacez `{{ site_header|default:_('Django administration') }}` (y compris les accolades) par le nom de votre propre site. Cela devrait donner quelque chose comme :

```
{% block branding %}
<h1 id="site-name"><a href="{% url 'admin:index' %}">Polls Administration</a></h1>
{% endblock %}
```

Nous utilisons cette approche pour vous apprendre comment surcharger des gabarits. Dans un projet réel, vous auriez probablement utilisé l'attribut [django.contrib.admin.AdminSite.site_header](#) pour faire cette même modification de manière plus simple.

Ce fichier gabarit contient beaucoup de texte comme `{% block branding %}` et `{{ title }}`. Les balises `{%` et `{{` font partie du langage de gabarit de Django. Lorsque Django génère `admin/base_site.html`, le langage de gabarit est évalué afin de produire la page HTML finale, tout comme nous l'avons vu dans le [3ème tutoriel](#).

Notez que tous les gabarits de l'interface d'administration par défaut de Django peuvent être remplacés. Pour remplacer un gabarit, faites simplement la même chose qu'avec `base_site.html`, copiez-le depuis le répertoire par défaut dans votre répertoire personnalisé, et faites les modifications nécessaires.

Personnalisation des gabarits de votre *application*

Les lecteurs avisés demanderont : mais si [DIRS](#) était vide par défaut, comment Django trouvait-il les gabarits par défaut de l'interface d'administration ? La réponse est que dans la mesure où [APP_DIRS](#) est défini à `True`, Django regarde automatiquement dans un éventuel sous-répertoire `templates/` à l'intérieur de chaque paquet d'application, pour l'utiliser en dernier recours (n'oubliez pas que `django.contrib.admin` est aussi une application).

Notre application de sondage n'est pas très compliquée et ne nécessite pas de gabarits d'administration personnalisés. Mais si elle devenait plus sophistiquée et qu'il faille modifier les gabarits standards de l'administration de Django pour certaines fonctionnalités, il serait plus logique de modifier les gabarits de l'*application*, et non pas ceux du *projet*. De cette façon, vous pourriez inclure l'application « polls » dans tout nouveau projet en étant certain que Django trouve les gabarits personnalisés nécessaires.

Lisez la [documentation sur le chargement de gabarits](#) pour des informations complètes sur la manière dont Django trouve ses gabarits.

Personnalisation de la page d'accueil de l'interface d'administration

De la même manière, il peut être souhaitable de personnaliser l'apparence de la page d'index de l'interface d'administration de Django.

Par défaut, il affiche toutes les applications de [INSTALLED_APPS](#) qui ont été inscrites dans l'application admin, par ordre alphabétique. Il est possible de faire des modifications significatives sur la mise en page. Après tout, la page d'index est probablement la page la plus importante du site d'administration, donc autant qu'elle soit facile à utiliser.

Le gabarit à personnaliser est `admin/index.html` (faites la même chose qu'avec `admin/base_site.html` dans la précédente section – copiez-le depuis le répertoire par défaut vers votre répertoire de gabarits personnel). Éditez le fichier, et vous verrez qu'il utilise une variable de gabarit appelée `app_list`. Cette variable contient toutes les applications Django installées et inscrites. À la place, vous pouvez écrire en dur les liens vers les pages d'administration spécifiques aux objets de la manière qui vous convient.

Et ensuite ?

Le tutoriel d'introduction se termine ici. Dans l'intervalle, vous pouvez toujours consulter quelques ressources sur la [page des prochaines étapes](#).

Si vous êtes à l'aise avec la création de paquets Python et intéressé à apprendre comment faire de l'application de sondage une « application réutilisable », consultez le [Tutoriel avancé : comment écrire des applications réutilisables](#).

[Écriture de votre première application Django, 6ème partie](#)
[Tutoriel avancé : concevoir des applications réutilisables](#)

Écriture de votre première application Django, 6ème partie

Ce tutoriel commence là où le [tutoriel 5](#) s'est achevé. Nous avons construit une application Web de sondage et nous allons maintenant ajouter une feuille de style et une image.

En plus du code HTML généré par le serveur, les applications Web doivent généralement servir des fichiers supplémentaires tels que des images, du JavaScript ou du CSS, utiles pour produire une page Web complète. Dans Django, on appelle ces fichiers des « fichiers statiques ».

Pour de petits projets, ce n'est pas un grand problème car il est possible de simplement placer les fichiers statiques à un endroit où le serveur Web peut les trouver. Cependant, dans des plus gros projets, surtout pour ceux qui contiennent plusieurs applications, la gestion de plusieurs groupes de fichiers statiques fournis par chaque application commence à devenir plus complexe.

C'est le travail de `django.contrib.staticfiles` : il collecte les fichiers statiques de chaque application (et de tout autre endroit que vous lui indiquez) pour les mettre dans un seul emplacement qui peut être facilement configuré pour servir les fichiers en production.

Personnalisation de l'apparence de votre application

Commencez par créer un répertoire nommé `static` dans votre répertoire `polls`. C'est là que Django va chercher les fichiers statiques, sur le même principe que la recherche de gabarits dans `polls/templates/`.

Le réglage [STATICFILES_FINDERS](#) de Django contient une liste de classes qui savent où aller chercher les fichiers statiques de différentes sources. Une de ces classes par défaut est `AppDirectoriesFinder` qui recherche un sous-répertoire « `static` » dans chaque application de [INSTALLED_APPS](#), comme l'application `polls` de notre tutoriel. Le site d'administration utilise la même structure de répertoires pour ses propres fichiers statiques.

À l'intérieur du répertoire `static` que vous venez de créer, créez un autre répertoire nommé `polls` dans lequel vous placez un fichier `style.css`. Autrement dit, votre feuille de style devrait se trouver à `polls/static/polls/style.css`. En raison du fonctionnement de `AppDirectoriesFinder`, vous pouvez vous référer à ce fichier depuis Django avec la syntaxe `polls/style.css`, sur le même principe utilisé pour se référer aux chemins de gabarits.

Espaces de noms des fichiers statiques

Tout comme pour les gabarits, nous *pourrions* plus simplement placer nos fichiers statiques directement dans `polls/static` (au lieu de créer un sous-répertoire `polls`), mais ce serait une mauvaise idée. Django choisit le premier fichier statique trouvé correspondant au nom recherché, et si vous aviez un fichier de même nom dans une *autre* application, Django ne pourrait pas les distinguer. Nous devons pouvoir indiquer à Django le bon fichier et le moyen le plus simple de s'en assurer est d'utiliser les *espaces de noms*. C'est-à-dire en plaçant ces fichiers statiques dans un *autre* sous-répertoire nommé d'après l'application.

Écrivez le code suivant dans cette feuille de style (`polls/static/polls/style.css`) :

```
polls/static/polls/style.css
```

```
li a {  
    color: green;  
}
```

Ensuite, ajoutez le contenu ci-dessous au début de `polls/templates/polls/index.html` :

```
polls/templates/polls/index.html
```

```
{% load staticfiles %}
```

```
<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}" />
```

`{% load staticfiles %}` charge la balise de gabarit [{% static %}](#) qui provient de la bibliothèque de balises de gabarit de `staticfiles`. La balise de gabarit `{% static %}` génère l'URL absolue du fichier statique.

C'est tout ce que vous avez à faire pour le développement. Rechargez `http://localhost:8000/polls/` et vous devriez voir que les liens des questions sont verts (style Django !) ce qui signifie que votre feuille de style a été correctement chargée.

Ajout d'une image d'arrière-plan

Ensuite, nous allons créer un sous-répertoire pour les images. Créez un sous-répertoire `images` dans le répertoire `polls/static/polls/`. Dans le répertoire créé, placez une image nommée `background.gif`. Autrement dit, le chemin de votre image sera `polls/static/polls/images/background.gif`.

Puis, ajoutez ceci à votre feuille de style (`polls/static/polls/style.css`):

```
polls/static/polls/style.css

body {
    background: white url("images/background.gif") no-repeat right bottom;
}
```

Rechargez `http://localhost:8000/polls/` et vous devriez voir l'arrière-plan chargé dans le coin inférieur droit de l'écran.

Avertissement

Bien sûr, la balise de gabarit `{% static %}` n'est pas disponible dans les fichiers statiques comme votre feuille de style, puisqu'ils ne sont pas générés par Django. Vous devriez toujours utiliser des **chemins relatifs** pour lier vos fichiers statiques entre eux, car vous pouvez ensuite modifier [STATIC_URL](#) (utilisé par la balise de gabarit `static` pour générer les URL) sans devoir modifier tous les chemins dans les fichiers statiques.

Ce sont les **bases**. Pour plus de détails sur les réglages et les autres fonctionnalités incluses dans le module, lisez le [manuel des fichiers statiques](#) et la [référence des fichiers statiques](#). La section [déploiement des fichiers statiques](#) aborde l'utilisation des fichiers statiques sur un vrai serveur.

Lorsque vous serez à l'aise avec les fichiers statiques, lisez la [partie 7 de ce tutoriel](#) pour apprendre comment personnaliser l'interface d'administration automatique de Django.

[Écriture de votre première application Django, 5ème partie](#)
[Écriture de votre première application Django, 7ème partie](#)

Écriture de votre première application Django, 7ème partie

This tutorial begins where [Tutorial 6](#) left off. We're continuing the Web-poll application and will focus on customizing Django's automatically-generated admin site that we first explored in [Tutorial 2](#).

Personnalisation du formulaire d'administration

Quand vous avez inscrit le modèle `Question` avec `admin.site.register(Question)`, Django a été capable de représenter l'objet dans un formulaire par défaut. Il sera cependant souvent souhaitable de personnaliser l'affichage et le comportement du formulaire. Cela se fait en indiquant certaines options à Django lors de l'inscription de l'objet.

Voyons comment cela fonctionne, en réordonnant les champs sur le formulaire d'édition. Remplacez la ligne `admin.site.register(Question)` par :

```
polls/admin.py

from django.contrib import admin

from .models import Question

class QuestionAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question_text']

admin.site.register(Question, QuestionAdmin)
```


You'll follow this pattern – create a model admin class, then pass it as the second argument to `admin.site.register()` – any time you need to change the admin options for a model.


Cette modification fait que la « Date de publication » apparaît avant le champ « Question » :

Home › Polls › Questions › What's up?

Change question

Date published:

Date: 2015-09-06 Today | 

Time: 21:16:20 Now | 

Question text:

What's up?

Ce n'est pas spécialement impressionnant avec seulement deux champs, mais pour un formulaire d'administration avec des dizaines de champs, choisir un ordre intuitif est un détail d'utilisation important.

Et en parlant de formulaires avec des dizaines de champs, il peut être utile de partager le formulaire en plusieurs sous-ensembles (fieldsets) :

```
polls/admin.py
from django.contrib import admin
from .models import Question

class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Question, QuestionAdmin)
```

Le premier élément de chaque tuple dans [fieldsets](#) est le titre du groupe de champs. Voici ce à quoi notre formulaire ressemble à présent :

Home › Polls › Questions › What's up?


Change question


Question text:

What's up?

Date information

Date published:

Date: 2015-09-06 Today 

Time: 21:16:20 Now 

Ajout d'objets liés

OK, nous avons notre page d'administration des questions. Mais une `Question` possède plusieurs choix `Choices`, et la page d'administration n'affiche aucun choix.

Pour le moment.

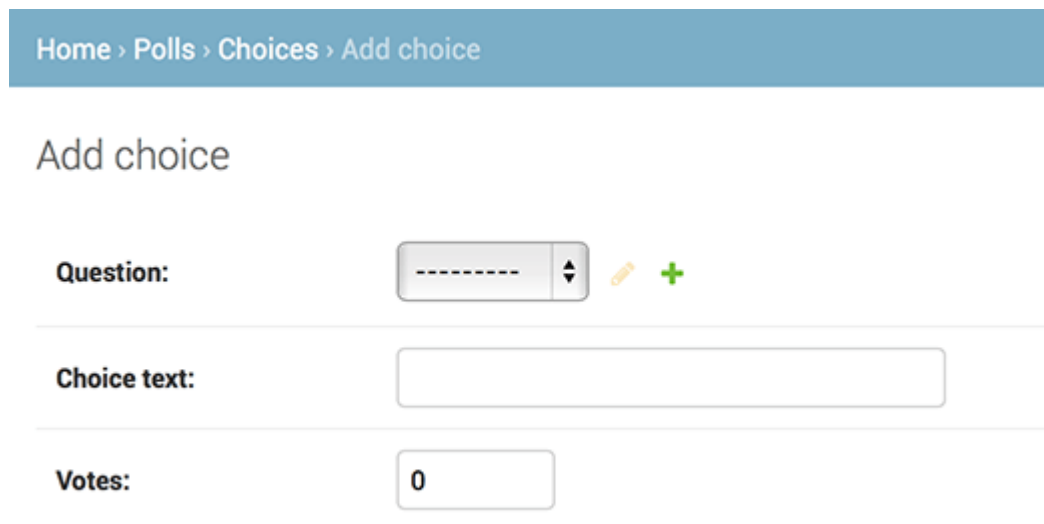
Il y a deux façons de résoudre ce problème. La première est d'inscrire `Choice` dans l'administration, comme nous l'avons fait pour `Question`. Voici ce que ça donnerait :

`polls/admin.py`

```
from django.contrib import admin

from .models import Choice, Question
# ...
admin.site.register(Choice)
```

Maintenant les choix "Choices" sont une option disponible dans l'interface d'administration de Django. Le formulaire « Add choice » ressemble à ceci :



Dans ce formulaire, le champ « Question » est une boîte de sélection contenant toutes les questions de la base de données. Django sait qu'une [ForeignKey](#) doit être représentée dans l'interface d'administration par une boîte `<select>`. Dans notre cas, seule une question existe à ce stade.

Notez également le lien « Ajouter un autre » à côté de « Question ». Chaque objet avec une relation `ForeignKey` vers un autre reçoit automatiquement cet outil. Quand vous cliquez sur « Ajouter un autre », vous obtenez une fenêtre surgissante contenant le formulaire « Add Question ». Si vous ajoutez une question dans cette fenêtre et que vous cliquez sur « Enregistrer », Django enregistre la question dans la base de données et l'ajoute dynamiquement comme choix sélectionné dans le formulaire « Add choice » que vous étiez en train de remplir.

Mais franchement, c'est une manière inefficace d'ajouter des objets `Choice` dans le système. Il serait préférable d'ajouter un groupe de choix « Choices » directement lorsque vous créez l'objet `Question`. Essayons de cette façon.

Enlevez l'appel `register()` pour le modèle `Choice`. Puis, modifiez le code d'inscription de `Question` comme ceci :

`polls/admin.py`

```
from django.contrib import admin

from .models import Choice, Question

class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
    inlines = [ChoiceInline]

admin.site.register(Question, QuestionAdmin)
```

Cela indique à Django : « les objets `Choice` sont édités dans la page d'administration de `Question`. Par défaut, fournir assez de place pour 3 choix ».

Chargez la page « Ajout Question » pour voir à quoi ça ressemble :

Add question

Question text:

Date information (Hide)

Date published:

Date:

Today | 

Time:

Now | 

CHOICES

Choice: #1

Choice text:

Votes:

0

Choice: #2

Choice text:

Votes:

0

Choice: #3

Choice text:

Votes:

0

 Add another Choice

Save and add another

Save and con

Ça marche comme ceci : il y a trois emplacements pour les choix « Choices » liés – comme indiqué par extra – et chaque fois que vous revenez sur la page de modification d’un objet déjà créé, vous obtenez trois emplacements supplémentaires.

Au bas des trois emplacements actuels, vous pouvez trouver un lien « Add another Choice ». Si vous cliquez dessus, un nouvel emplacement apparaît. Si vous souhaitez supprimer l’emplacement ajouté, vous pouvez cliquer sur le X en haut à droite de l’emplacement. Notez que vous ne pouvez pas enlever les trois emplacements de départ. Cette image montre l’emplacement ajouté :

CHOICES	
Choice: #1	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #2	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #3	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #4	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
+ Add another Choice	

Un petit problème cependant. Cela prend beaucoup de place d’afficher tous les champs pour saisir les objets Choice liés. C’est pour cette raison que Django offre une alternative d’affichage en tableau des objets liés ; il suffit de modifier la déclaration ChoiceInline comme ceci :

polls/admin.py

```
class ChoiceInline(admin.TabularInline):  
    #...
```

Avec ce `TabularInline` (au lieu de `StackedInline`), les objets liés sont affichés dans un format plus compact, tel qu'un tableau :

CHOICES	
CHOICE TEXT	VOTES
<input type="text"/>	<input type="text" value="0"/>
<input type="text"/>	<input type="text" value="0"/>
<input type="text"/>	<input type="text" value="0"/>

[+ Add another Choice](#)

[Save and add another](#) [Save and cont](#)

Remarquez la colonne supplémentaire « Delete? » qui permet de supprimer des lignes ajoutées à l'aide du bouton « Add Another Choice » ainsi que les lignes déjà enregistrées.

Personnalisation de la liste pour modification de l'interface d'administration

Maintenant que la page d'administration des `Questions` présente un peu mieux, améliorons la page « liste pour modification » – celle qui affiche toutes les questions du système.

Voici à quoi ça ressemble pour l'instant :

Select question to change

Action: 0 of 1 selected☐ QUESTION TEXT☐ What's up?

1 question

Par défaut, Django affiche le `str()` de chaque objet. Mais parfois, il serait plus utile d'afficher des champs particuliers. Dans ce but, utilisez l'option [list_display](#), qui est un tuple de noms de champs à afficher, en colonnes, sur la page liste pour modification de l'objet :

polls/admin.py

```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date')
```

Juste pour la démonstration, incluons également la méthode `was_published_recently()` du [tutoriel 2](#):

polls/admin.py

```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date', 'was_published_recently')
```

À présent, la page liste pour modification des questions ressemble à ceci :

Select question to change

Action: 0 of 1 selected

<input type="checkbox"/>	QUESTION TEXT	DATE PUBLISHED	WAS PUBLISHED RECENTLY
<input type="checkbox"/>	What's up?	Sept. 3, 2015, 9:16 p.m.	False

1 question

Vous pouvez cliquer sur les en-têtes de colonne pour trier selon ces valeurs – sauf dans le cas de l'en-tête `was_published_today`, parce que le tri selon le résultat d'une méthode arbitraire n'est pas pris en charge. Notez aussi que l'en-tête de la colonne pour `was_published_today` est, par défaut, le nom de la méthode (avec les soulignements remplacés par des espaces) et que chaque ligne contient la représentation textuelle du résultat.

Vous pouvez améliorer cela en donnant à cette méthode (dans `polls/models.py`) quelques attributs, comme ceci :

polls/models.py

```
class Question(models.Model):
    # ...
    def was_published_recently(self):
        now = timezone.now()
        return now - datetime.timedelta(days=1) <= self.pub_date <= now
    was_published_recently.admin_order_field = 'pub_date'
    was_published_recently.boolean = True
    was_published_recently.short_description = 'Published recently?'
```

Pour plus d'informations sur ces propriétés de méthodes, voir [list_display](#).

Modifiez encore une fois le fichier `polls/admin.py` et améliorez la page de liste pour modification des `Questions`: ajout de filtrage à l'aide de l'attribut [list_filter](#). Ajoutez la ligne suivante à `QuestionAdmin`:

```
list_filter = ['pub_date']
```

Cela ajoute une barre latérale « Filter » qui permet de filtrer la liste pour modification selon le champ `pub_date` :

Select question to change

Action: 0 of 1 selected

<input type="checkbox"/>	QUESTION TEXT	DATE PUBLISHED	PUBLISHED RECENTLY?
<input type="checkbox"/>	What's up?	Sept. 3, 2015, 9:16 p.m.	

1 question

FILTER

By date

Any date

Today

Past 7 d

This mon

This year

Le type de filtre affiché dépend du type de champ à filtrer. Comme `pub_date` est un champ [DateTimeField](#), Django sait fournir des options de filtrage appropriées : « Toutes les dates », « Aujourd'hui », « Les 7 derniers jours », « Ce mois-ci », « Cette année ».

Ça a meilleure forme. Ajoutons une fonctionnalité de recherche :

```
search_fields = ['question_text']
```

Cela ajoute une boîte de recherche en haut de la liste pour modification. Quand quelqu'un saisit des termes de recherche, Django va rechercher dans le champ `question_text`. Vous pouvez indiquer autant de champs que vous le désirez – néanmoins, en raison de l'emploi d'une requête `LIKE` en arrière-plan, il s'agit de rester raisonnable quant au nombre de champs de recherche, sinon la base de données risque de tirer la langue !

C'est maintenant le bon moment de noter que les listes pour modification vous laissent une grande liberté de mise en page. Par défaut, 100 éléments sont affichés par page. La [pagination](#) des listes pour modification, les [boîtes de recherche](#), les [filtres](#), les [hiérarchies](#), les [calendaires](#) et le [tri selon l'en-tête de colonne](#), tout fonctionne ensemble pour une utilisation optimale.

Personnalisation de l'apparence de l'interface d'administration

Clairement, avoir « Django administration » en haut de chaque page d'administration est ridicule. C'est juste du texte de substitution.

Toutefois, c'est facile à modifier en utilisant le système de gabarits de Django. Le site d'administration de Django est fait en Django lui-même, et ses interfaces utilisent le système de gabarits propre à Django.

Personnalisation des gabarits de votre *projet*

Créez un répertoire s'appelant `templates` dans le répertoire de votre projet (celui qui contient `manage.py`). Les gabarits peuvent se trouver à n'importe quel endroit du système de fichiers, pourvu qu'ils soient accessibles par Django (Django utilise le même utilisateur que celui qui a démarré votre serveur). Cependant, par convention, il est recommandé de conserver les gabarits à l'intérieur du projet.

Ouvrez votre fichier de configuration (`mysite/settings.py`, souvenez-vous) et ajoutez une option [DIRS](#) dans le réglage [TEMPLATES](#):

`mysite/settings.py`

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

[DIRS](#) est une liste de répertoires du système de fichiers que Django parcourt lorsqu'il doit charger les gabarits ; il s'agit d'un chemin de recherche.

Organisation des gabarits

Just like the static files, we *could* have all our templates together, in one big templates directory, and it would work perfectly well. However, templates that belong to a particular application should be placed in that application's template directory (e.g. `polls/templates`) rather than the project's (`templates`). We'll discuss in more detail in the [reusable apps tutorial](#) why we do this.

Créez maintenant dans `templates` un répertoire nommé `admin` et copiez-y le gabarit `admin/base_site.html` à partir du répertoire par défaut des gabarits d'administration dans le code source de Django (`django/contrib/admin/templates`).

Où se trouvent les fichiers sources de Django ?

Si vous ne savez pas où les fichiers source de Django se trouvent sur votre système, lancez la commande suivante :

```
$ python -c "import django; print(django.__path__)"
```

Ensuite, éditez simplement le fichier et remplacez `{{ site_header|default:_('Django administration') }}` (y compris les accolades) par le nom de votre propre site. Cela devrait donner quelque chose comme :

```
{% block branding %}
<h1 id="site-name"><a href="{% url 'admin:index' %}">Polls Administration</a></h1>
{% endblock %}
```

Nous utilisons cette approche pour vous apprendre comment surcharger des gabarits. Dans un projet réel, vous auriez probablement utilisé l'attribut [django.contrib.admin.AdminSite.site_header](#) pour faire cette même modification de manière plus simple.

Ce fichier gabarit contient beaucoup de texte comme `{% block branding %}` et `{{ title }}`. Les balises `{%}` et `{{}}` font partie du langage de gabarit de Django. Lorsque Django génère `admin/base_site.html`, le langage de gabarit est évalué afin de produire la page HTML finale, tout comme nous l'avons vu dans le [3ème tutoriel](#).

Notez que tous les gabarits de l'interface d'administration par défaut de Django peuvent être remplacés. Pour remplacer un gabarit, faites simplement la même chose qu'avec `base_site.html`, copiez-le depuis le répertoire par défaut dans votre répertoire personnalisé, et faites les modifications nécessaires.

Personnalisation des gabarits de votre *application*

Les lecteurs avisés demanderont : mais si [DIRS](#) était vide par défaut, comment Django trouvait-il les gabarits par défaut de l'interface d'administration ? La réponse est que dans la mesure où [APP_DIRS](#) est défini à `True`, Django regarde automatiquement dans un éventuel sous-répertoire `templates/` à l'intérieur de chaque paquet d'application, pour l'utiliser en dernier recours (n'oubliez pas que `django.contrib.admin` est aussi une application).

Notre application de sondage n'est pas très compliquée et ne nécessite pas de gabarits d'administration personnalisés. Mais si elle devenait plus sophistiquée et qu'il faille modifier les gabarits standards de l'administration de Django pour certaines fonctionnalités, il serait plus logique de modifier les gabarits de l'*application*, et non pas ceux du *projet*. De cette façon, vous pourriez inclure l'application « polls » dans tout nouveau projet en étant certain que Django trouve les gabarits personnalisés nécessaires.

Lisez la [documentation sur le chargement de gabarits](#) pour des informations complètes sur la manière dont Django trouve ses gabarits.

Personnalisation de la page d'accueil de l'interface d'administration

De la même manière, il peut être souhaitable de personnaliser l'apparence de la page d'index de l'interface d'administration de Django.

Par défaut, il affiche toutes les applications de [INSTALLED_APPS](#) qui ont été inscrites dans l'application admin, par ordre alphabétique. Il est possible de faire des modifications significatives sur la mise en page. Après tout, la page d'index est probablement la page la plus importante du site d'administration, donc autant qu'elle soit facile à utiliser.

Le gabarit à personnaliser est `admin/index.html` (faites la même chose qu'avec `admin/base_site.html` dans la précédente section – copiez-le depuis le répertoire par défaut vers votre répertoire de gabarits personnel). Éditez le fichier, et vous verrez qu'il utilise une variable de gabarit appelée `app_list`. Cette variable contient toutes les applications Django installées et inscrites. À la place, vous pouvez écrire en dur les liens vers les pages d'administration spécifiques aux objets de la manière qui vous convient.

Et ensuite ?

Le tutoriel d'introduction se termine ici. Dans l'intervalle, vous pouvez toujours consulter quelques ressources sur la [page des prochaines étapes](#).

Si vous êtes à l'aise avec la création de paquets Python et intéressé à apprendre comment faire de l'application de sondage une « application réutilisable », consultez le [Tutoriel avancé : comment écrire des applications réutilisables](#).

[Écriture de votre première application Django, 6ème partie](#)
[Tutoriel avancé : concevoir des applications réutilisables](#)

Tutoriel avancé : concevoir des applications réutilisables

This advanced tutorial begins where [Tutorial 7](#) left off. We'll be turning our Web-poll into a standalone Python package you can reuse in new projects and share with other people.

If you haven't recently completed Tutorials 1–7, we encourage you to review these so that your example project matches the one described below.

Importance du recyclage

C'est un gros travail de concevoir, construire, tester et maintenir une application Web. Beaucoup de projets Python et Django ont des problématiques communes. Ne serait-il pas merveilleux si nous pouvions économiser un peu de ce travail répétitif ?

Le recyclage de code est un principe de vie en Python. Le site [Python Package Index \(PyPI\)](#) recense une très large variété de paquets qu'il est possible d'utiliser dans ses propres programmes Python.

Parcourez le site [Django Packages](#) pour trouver des applications réutilisables qu'il est possible d'intégrer dans votre projet. Django lui-même n'est qu'un paquet Python. Cela signifie que vous pouvez prendre des paquets Python ou des applications Django existantes et les composer dans votre propre projet Web. Vous n'avez plus qu'à écrire les parties qui font de votre projet un projet unique.

Disons que vous débutez un nouveau projet qui nécessite une application de sondage telle que celle sur laquelle nous avons travaillé. Comme pouvez-vous recycler cette application ? Heureusement, vous êtes déjà sur la bonne voie. Dans le [tutoriel 3](#), nous avons vu comment découpler les sondages de l'URLconf de base du projet en utilisant un `include`. Dans ce tutoriel, nous irons un peu plus loin pour rendre l'application simple à utiliser dans de nouveaux projets et prête à être publiée pour que d'autres puissent l'installer et l'exploiter.

Paquet ? Application ?

A Python [package](#) provides a way of grouping related Python code for easy reuse. A package contains one or more files of Python code (also known as “modules”).

Un paquet peut être importé avec `import foo.bar` ou `from foo import bar`. Pour qu'un répertoire (comme `polls`) constitue un paquet, il doit contenir un fichier spécial `__init__.py`, même si ce dernier est vide.

Une *application* Django n'est qu'un paquet Python qui est spécialement prévu pour fonctionner dans un projet Django. Une application peut utiliser des conventions Django partagées, comme la présence de sous-modules `models`, `tests`, `urls` et `views`.

Par la suite, nous utilisons le terme *empaquetage* (packaging) pour décrire le processus de création d'un paquet Python facile à installer par tout le monde. Cela peut prêter à confusion, effectivement.

Votre projet et votre application recyclable

Après les tutoriels précédents, notre projet devrait ressembler à ceci :

```
mysite/
  manage.py
  mysite/
    __init__.py
    settings.py
    urls.py
    wsgi.py
  polls/
    __init__.py
    admin.py
    migrations/
      __init__.py
      0001_initial.py
    models.py
    static/
      polls/
        images/
          background.gif
        style.css
```

```
templates/  
  polls/  
    detail.html  
    index.html  
    results.html  
tests.py  
urls.py  
views.py  
templates/  
  admin/  
    base_site.html
```

Vous avez créé `mysite/templates` dans le [tutoriel 7](#) et `polls/templates` dans le [tutoriel 3](#). Il est peut-être maintenant plus facile de comprendre pourquoi nous avons séparé les répertoires de gabarits du projet de ceux de l'application : tout ce qui fait partie de l'application de sondage se trouve dans `polls`. L'application forme un tout par elle-même et est ainsi plus facile à intégrer dans un nouveau projet.

Le répertoire `polls` pourrait maintenant être copié dans un nouveau projet Django pour être immédiatement utilisé. L'application n'est cependant pas encore tout à fait prête pour être publiée. Pour cela, il faut encore emballer l'application afin que d'autres puissent facilement l'installer.

Installation de quelques prérequis

L'état actuel de la construction de paquets Python est un peu embrouillé par l'utilisation de plusieurs outils. Dans ce tutoriel, nous allons utiliser [setuptools](#) pour construire nos paquets. C'est l'outil d'emballage que nous recommandons (fusionné avec le dérivé `distribute`). Nous utiliserons aussi [pip](#) pour l'installer et le désinstaller. Vous devriez installer maintenant ces deux paquets. Si vous avez besoin d'aide, vous pouvez vous référer à [comment installer Django avec pip](#). Vous pouvez installer `setuptools` de la même manière.

Emballage de l'application

L'*emballage* Python se réfère à la préparation d'une application dans un format spécifique qui peut être facilement installé et utilisé. Django lui-même est emballé de manière très semblable. Pour une petite application comme la nôtre, ce processus n'est pas trop difficile.

1. Premièrement, créez un répertoire parent pour `polls`, en dehors de votre projet Django. Nommez ce répertoire `django-polls`.

Choix d'un nom pour votre application

Au moment de choisir un nom pour votre paquet, consultez les ressources comme PyPI pour éviter des conflits de nom avec des paquets existants. Il est souvent utile de préfixer votre nom de module avec `django-` lors de la création d'un paquet à distribuer. Les personnes qui cherchent des applications Django peuvent ainsi identifier plus facilement votre application comme étant spécifique à Django.

Les étiquettes d'applications (c'est-à-dire la partie finale du chemin pointé vers le paquet de l'application) figurant dans [INSTALLED_APPS](#) doivent être uniques. Évitez d'utiliser la même étiquette que l'une des [applications contribuéées](#) de Django, par exemple `auth`, `admin` ou `messages`.

2. Déplacez le répertoire `polls` dans le répertoire `django-polls`.
3. Créez un fichier `django-polls/README.rst` contenant ceci :

```
django-polls/README.rst
```

```
=====  
Polls  
=====
```

```
Polls is a simple Django app to conduct Web-based polls. For each  
question, visitors can choose between a fixed number of answers.
```

```
Detailed documentation is in the "docs" directory.
```

```
Quick start  
-----
```

1. Add "polls" to your `INSTALLED_APPS` setting like this::

```
INSTALLED_APPS = [  
    'polls',  
]
```

2. Include the polls URLconf in your project `urls.py` like this::

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

3. Run ``python manage.py migrate`` to create the polls models.
4. Start the development server and visit `http://127.0.0.1:8000/admin/` to create a poll (you'll need the Admin app enabled).
5. Visit `http://127.0.0.1:8000/polls/` to participate in the poll.

4. Créez un fichier `django-polls/LICENSE`. Le choix d'une licence n'est pas dans le thème de ce tutoriel, mais nous vous rendons attentif au fait que du code publié sans licence est *inutile*. Django et de nombreuses applications compatibles Django sont distribués sous licence BSD ; vous êtes bien sûr libre d'utiliser la licence de votre choix. Sachez seulement que le choix de la licence a un impact sur la quantité d'utilisateurs potentiels de votre code.
5. Next we'll create a `setup.py` file which provides details about how to build and install the app. A full explanation of this file is beyond the scope of this tutorial, but the [setuptools docs](#) have a good explanation. Create a file `django-polls/setup.py` with the following contents:

```
django-polls/setup.py
```

```

import os
from setuptools import find_packages, setup

with open(os.path.join(os.path.dirname(__file__), 'README.rst')) as readme:
    README = readme.read()

# allow setup.py to be run from any path
os.chdir(os.path.normpath(os.path.join(os.path.abspath(__file__),
os.pardir)))

setup(
    name='django-polls',
    version='0.1',
    packages=find_packages(),
    include_package_data=True,
    license='BSD License', # example license
    description='A simple Django app to conduct Web-based polls.',
    long_description=README,
    url='https://www.example.com/',
    author='Your Name',
    author_email='yourname@example.com',
    classifiers=[
        'Environment :: Web Environment',
        'Framework :: Django',
        'Framework :: Django :: X.Y', # replace "X.Y" as appropriate
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License', # example license
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        # Replace these appropriately if you are stuck on Python 2.
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.4',
        'Programming Language :: Python :: 3.5',
        'Topic :: Internet :: WWW/HTTP',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
    ],
)

```

6. Seuls des modules et paquets Python sont inclus par défaut dans le paquet. Pour ajouter des fichiers supplémentaires, nous devons créer un fichier `MANIFEST.in`. La documentation de `setuptools` mentionnée à l'étape précédente présente ce fichier en détails. Pour inclure les gabarits et les fichiers `README.rst` et `LICENSE`, créez un fichier `django-polls/MANIFEST.in` avec le contenu suivant :

```

django-polls/MANIFEST.in

include LICENSE
include README.rst
recursive-include polls/static *
recursive-include polls/templates *

```

7. Il est facultatif, mais recommandé, d'inclure de la documentation détaillée dans votre application. Créez un répertoire vide `django-polls/docs` en prévision de contenu documentaire. Ajoutez une ligne supplémentaire dans `django-polls/MANIFEST.in`:

```
recursive-include docs *
```

Notez que le répertoire `docs` ne sera pas inclus dans votre paquet tant que vous n’y ajoutez pas de fichier. Beaucoup d’applications Django offrent aussi un version en ligne de leur documentation par le moyen de sites comme readthedocs.org.

8. Essayez de construire votre paquet avec `python setup.py sdist` (lancé depuis `django-polls`). Cela va créer un répertoire nommé `dist` et construire votre nouveau paquet, `django-polls-0.1.tar.gz`.

Pour plus d’informations sur l’empaquetage, consultez (en anglais) le [Tutoriel sur l’empaquetage et la distribution de projets](#).

Utilisation de son propre paquet

Comme nous avons déplacé le répertoire `polls` hors du projet, celui-ci ne fonctionne plus. Nous allons maintenant corriger ça en installant notre nouveau paquet `django-polls`.

Installation en tant que bibliothèque utilisateur

Les étapes suivantes installent `django-polls` comme une bibliothèque utilisateur. Les installations « utilisateur » comportent bien des avantages sur l’installation de paquets au niveau système, comme de pouvoir être utilisables sur des systèmes dont vous n’êtes pas administrateur ou pour éviter que le paquet ne perturbe des services du système ou d’autres utilisateurs de la machine.

Notez que les installations « utilisateur » peuvent tout de même influencer le comportement d’outils système qui tournent sous l’identité de cet utilisateur, il est donc plus robuste d’utiliser `virtualenv` (voir ci-dessous).

1. Pour installer le paquet, utilisez `pip` (vous l’avez déjà [installé](#), n’est-ce pas ?) :

```
pip install --user django-polls/dist/django-polls-0.1.tar.gz
```

2. Si tout va bien, votre projet Django devrait de nouveau fonctionner correctement. Lancez à nouveau le serveur pour le vérifier.
3. Pour désinstaller le paquet, utilisez `pip` :

```
pip uninstall django-polls
```

Publication de l’application

Après avoir empaqueté et testé `django-polls`, il est prêt à être partagé avec le monde entier ! Si ce n’était pas qu’un exemple, vous pourriez maintenant :

- Envoyer le paquet à un ami par courriel.
- Envoyer le paquet sur votre site Web.

- Publier le paquet dans un dépôt public, tel que le [Python Package Index \(PyPI\)](https://pypi.org/).
packaging.python.org dispose d'un [bon tutoriel](#) (en anglais) qui explique comment faire cela.

Installation de paquets Python avec virtualenv

Précédemment, nous avons installé l'application de sondage comme bibliothèque utilisateur. Cela comporte quelques désavantages :

- La modification de bibliothèques utilisateurs peut influencer d'autres logiciels Python sur votre système.
- Vous ne pourrez pas faire fonctionner plusieurs versions de ce paquet en parallèle (ou avec d'autres portant le même nom).

Typiquement, ces situations ne se produisent que lorsque vous maintenez plusieurs projets Django. Dans ces cas, la meilleure solution est d'utiliser [virtualenv](#). Cet outil permet de maintenir plusieurs environnements Python isolés les uns des autres, ayant chacun leur propre copie des bibliothèques et de l'espace de nom des paquets.

[Écriture de votre première application Django, 7ème partie](#)
[Quelle lecture pour la suite](#)

Quelle lecture pour la suite

Vous avez donc lu toute la [documentation d'introduction](#) et vous avez décidé de continuer à utiliser Django. Nous n'avons fait qu'effleurer la surface dans cette introduction (même en ayant lu chaque mot de l'introduction, vous n'avez parcouru qu'environ 5 % de toute la documentation).

Comment continuer ?

Nous avons toujours été partisans de l'apprentissage par l'action. À ce stade, vous devriez en savoir assez pour commencer votre propre projet en explorant la technologie. Dès que vous ressentez le besoin d'apprendre de nouveaux concepts, revenez à la documentation.

Nous avons consacré beaucoup d'énergie pour rendre la documentation de Django utile, facile à lire et aussi complète que possible. La suite de ce document explique un peu plus comment est structurée cette documentation afin que vous puissiez en tirer le meilleur parti.

(Oui, c'est une documentation sur la documentation. Soyez assuré que nous n'avons aucun projet d'écrire un document sur la manière de lire le document au sujet de la documentation.)

Recherche dans la documentation

La documentation de Django est *volumineuse* (presque 450 000 mots), il est donc parfois périlleux de trouver ce que l'on cherche. Deux bons points de départ sont la [Page de recherche](#) et l'[Index](#).

Ou vous pouvez tout aussi bien naviguer dans les contenus.

Organisation de la documentation

La documentation principale de Django est segmentée en sections prévues pour répondre à différents besoins :

- Le [contenu d'introduction](#) est conçu pour les débutants avec Django, ou avec le développement Web en général. Il ne traite pas de tout en détails, mais donne plutôt un aperçu de haut niveau sur l'approche générale du développement en Django.
- Les [guides thématiques](#), d'un autre côté, traitent en profondeur de domaines spécifiques de Django. Il existe des guides complets sur le [système des modèles](#), le [moteur de gabarits](#), l'[architecture des formulaires](#), et bien plus encore.

C'est probablement l'endroit où vous passerez le plus de temps ; si vous faites l'effort de parcourir ces différents guides, vous connaîtrez presque tout ce qu'il y a à savoir sur Django.

- Le développement Web est à large spectre, mais pas profond (les problématiques recouvrent de nombreux domaines). Nous avons rédigé un ensemble de [marches à suivre](#) qui répondent aux « Comment dois-je faire pour... ? » les plus courants. Vous trouverez là des informations sur [la génération de PDF avec Django](#), [l'écriture de balises de gabarits personnalisées](#), et d'autres sujets encore.

Les réponses aux questions vraiment très souvent posées peuvent aussi être trouvées dans la [FAQ](#).

- Les guides et les marches à suivre ne traitent pas de chaque classe, fonction ou méthode disponible dans Django, car ce serait décourageant pour ceux qui cherchent à apprendre. Par contre, tous les détails sur les classes, fonctions, méthodes et modules sont donnés dans la [référence](#). C'est là que vous devrez chercher pour obtenir des détails sur une fonction particulière ou tout autre objet.
- Si vous êtes intéressé à déployer un projet publiquement, notre documentation contient [quelques guides](#) présentant diverses stratégies de déploiement ainsi qu'une [liste de contrôle de déploiement](#) pour vous rendre attentifs à certains aspects importants.
- Pour finir, il existe aussi une documentation « spécialisée » qui n'est habituellement pas destinée à la majorité des développeurs. Ceci inclut les [notes de publication](#) et de la [documentation interne](#) pour ceux qui se destinent à ajouter du code dans Django lui-même, ainsi que [quelques autres documents](#) qui n'entrent dans aucune autre catégorie.

Mise à jour de la documentation

Tout comme le code Django est développé et amélioré chaque jour, cette documentation s'améliore aussi constamment. Nous améliorons la documentation pour plusieurs raisons :

- Pour corriger des contenus, comme des corrections grammaticales et orthographiques.
- Pour ajouter des informations ou des exemples dans des sections existantes mais qui ont besoin d'être complétées.
- Pour documenter des fonctionnalités Django qui ne figurent pas encore dans la documentation (la liste des éléments concernés diminue, mais il y en a encore).
- Pour ajouter de la documentation pour de nouvelles fonctions lorsqu'elles sont ajoutées, ou quand l'API ou le comportement de Django est modifié.

La documentation de Django est conservée dans le même système de gestion de version que le code. Elle se trouve dans le répertoire [docs](#) de notre dépôt Git. Chaque page de cette documentation est un fichier texte distinct dans le dépôt.

Où l'obtenir

Vous pouvez lire la documentation de Django de plusieurs manières. Les voici, par ordre de préférence :

Sur le Web

La version la plus récente de la documentation de Django se trouve à l'adresse <https://docs.djangoproject.com/en/dev/>. Ces pages HTML sont générées automatiquement à partir des fichiers texte dans le gestionnaire de versions. Cela signifie qu'elles reflètent la toute dernière version de Django, comprenant les toutes dernières corrections et ajouts, et qu'elles abordent les fonctionnalités les plus récentes de Django, qui ne sont parfois disponibles que pour ceux qui utilisent la version de développement de Django (voir « Différences entre versions » ci-dessous).

Nous vous encourageons à aider à l'amélioration de la documentation en proposant des modifications, des corrections et des suggestions dans le [système de tickets](#). Les développeurs Django surveillent de façon attentive le système de tickets et exploitent vos contributions afin d'améliorer la documentation au profit de tous.

Notez toutefois que les tickets doivent être liés explicitement à la documentation, et non pas à des questions d'aide de portée générale. Si vous avez besoin d'aide avec une configuration particulière de Django, adressez-vous plutôt aux canaux d'aide appropriés ([liste de diffusion |django-users|](#) ou [canal IRC #django](#)).

En mode texte

Pour une lecture hors ligne ou simplement par préférence, il est possible de lire la documentation de Django en mode texte pur.

Si vous utilisez une version officielle de Django, le paquet compressé du code contient un répertoire `docs/` comprenant toute la documentation correspondant à cette version.

Si vous utilisez la version de développement de Django (« trunk »), notez que le répertoire `docs/` contient toute la documentation. Vous pouvez mettre à jour votre extraction Git pour obtenir les dernières modifications.

Une astuce pratique pour exploiter la documentation brute de Django est d'utiliser l'utilitaire Unix « `grep` » pour rechercher du contenu dans toute la documentation. Par exemple, ceci vous indiquera toutes les mentions de l'expression « `max_length` » dans la documentation :

```
$ grep -r max_length /path/to/django/docs/
```

En HTML, localement

Vous pouvez produire une version locale de la documentation HTML en suivant ces quelques étapes :

- La documentation de Django utilise un système appelé [Sphinx](#) pour convertir le texte brut en HTML. Vous devez donc installer Sphinx soit en le téléchargeant et en installant le paquet à partir du site Web de Sphinx, ou à l'aide de `pip` :

```
$ pip install Sphinx
```

- Puis, il suffit d'utiliser le fichier `Makefile` inclus pour transformer la documentation en HTML :

```
$ cd path/to/django/docs
$ make html
```

[GNU Make](#) doit être installé pour que ça fonctionne.

Si vous êtes sur Windows, il est aussi possible d'utiliser le fichier batch inclus :

```
cd path\to\django\docs
make.bat html
```

- La documentation HTML sera placée dans `docs/_build/html`.

Note

La génération de la documentation Django fonctionne avec toute version de Sphinx à partir de la 0.6, mais nous recommandons d'utiliser la version 1.0.2 ou plus récente de Sphinx.

Différences entre versions

Comme mentionné précédemment, la documentation au format texte dans le dépôt Git contient toutes les modifications les plus récentes. Ces modifications comprennent souvent de la documentation sur des nouvelles fonctionnalités ajoutées dans la version de développement de Django (la version dite « trunk »). Pour cette raison, il est utile d'expliquer notre politique de conservation de la documentation spécifique pour chaque version de Django.

Nous appliquons la politique suivante :

- La documentation principale sur le site djangoproject.com est une version HTML de la toute dernière version dans Git. Cette documentation correspond toujours à la dernière version officielle de Django, augmentée de toutes les fonctionnalités ajoutées ou modifiées *depuis* la dernière publication.
- Au rythme de l'ajout de fonctionnalités dans la version de développement de Django, nous essayons de mettre à jour la documentation dans la même transaction Git que les modifications de code.
- Pour mettre en évidence les modifications ou ajouts de fonctionnalités dans la documentation, nous utilisons la phrase « New in version X.Y », où X.Y correspond à la prochaine version qui sera publiée (donc celle qui est en cours de développement).
- Les corrections et améliorations de documentation peuvent être reportées dans la branche de la version stable, selon l'appréciation de la personne chargée du commit. Cependant, dès qu'une version de Django n'est plus prise en charge, cette version de la documentation n'est plus mise à jour.
- La [page Web principale de la documentation](#) contient des liens vers la documentation de toutes les versions précédentes. Prenez soin de bien utiliser la version de la documentation qui correspond à la version de Django que vous utilisez !

[Tutoriel avancé : concevoir des applications réutilisables](#)
[Écriture de votre premier correctif pour Django](#)

Écriture de votre premier correctif pour Django

Introduction

Intéressé à redonner un peu de votre temps à la communauté ? Vous avez peut-être trouvé un bogue dans Django et vous souhaiteriez le corriger, ou alors vous aimeriez ajouter une petite fonctionnalité dans Django.

La meilleure manière de réaliser ces objectifs est de contribuer à Django lui-même. Si cela peut paraître une tâche effrayante au premier abord, ce n'est finalement pas si difficile que cela. Nous allons vous accompagner tout au long de ce processus afin que vous puissiez apprendre avec un exemple.

À qui est destiné ce tutoriel ?

Voir aussi

Si vous cherchez une référence sur la façon de soumettre des correctifs, consultez la documentation [Submitting patches](#).

Pour ce tutoriel, il s'agit d'avoir au moins une compréhension générale de la manière dont Django fonctionne. Cela implique que vous devez être à l'aise en parcourant les tutoriaux existants sur [l'écriture de votre première application Django](#). De plus, vous devriez bien connaître Python lui-même. Si ce n'est pas le cas, [Dive Into Python](#) est un livre en ligne fantastique (et gratuit) pour les programmeurs Python débutants.

Ceux d'entre vous qui ne sont pas familiers avec les systèmes de gestion de versions et avec Trac trouveront dans ce tutoriel et ses liens les bonnes informations pour bien démarrer. Toutefois, il serait souhaitable d'en apprendre un peu plus sur ces différents outils si vous pensez contribuer régulièrement à Django.

Mais pour la plus grande partie, ce tutoriel vise à expliquer le plus possible de choses pour qu'il puisse convenir au plus grand nombre.

Où obtenir de l'aide :

Si vous avez des problèmes en suivant ce tutoriel, vous pouvez écrire (en anglais) à [django-developers](#) ou utiliser le canal IRC [#django-dev sur irc.freenode.net](#) (ou #django-fr en français) pour discuter avec d'autres utilisateurs Django qui pourront peut-être vous aider.

Quels sont les contenus de ce tutoriel ?

Nous allons vous guider dans votre première contribution d'un correctif pour Django. À la fin de ce tutoriel, vous aurez acquis une compréhension basique à la fois des outils et des processus nécessaires. Plus particulièrement, nous aborderons les thèmes suivants :

- installation de Git,

- le téléchargement d'une copie de développement de Django,
- le lancement de la suite de tests de Django,
- l'écriture d'un test pour votre correctif,
- l'écriture du code du correctif,
- le test de votre correctif,
- la génération d'un fichier correctif contenant vos modifications,
- les sources d'informations supplémentaires.

Une fois le tutoriel terminé, vous pouvez consulter le reste de la [documentation de Django sur la façon de contribuer](#). Elle contient de nombreuses informations utiles et constitue une lecture indispensable pour tout contributeur en herbe à Django. Si vous avez des questions, vous y trouverez probablement les réponses.

Python 3 indispensable !

Ce tutoriel suppose que vous utilisez Python 3. Obtenez la dernière version sur la [page des téléchargements de Python](#) ou par l'intermédiaire du gestionnaire des paquets de votre système.

Pour les utilisateurs Windows

Lors de l'installation de Python sur Windows, prenez soin de cocher l'option « Ajouter python.exe au chemin PATH » afin qu'il soit accessible en tout temps dans la ligne de commande.

Code de conduite

En tant que contributeur, vous pouvez nous aider à maintenir la communauté Django ouverte et accueillante. Lisez et respectez notre [Code de conduite](#).

Installation de Git

Pour ce tutoriel, Git doit être installé afin de pouvoir télécharger la version actuelle de développement de Django et pour générer les fichiers correctifs contenant vos modifications.

Pour savoir si Git est installé, saisissez `git` sur une ligne de commande. Si vous obtenez un message vous indiquant que la commande n'a pas pu être trouvée, c'est que vous devez l'installer, voir [la page de téléchargement de Git](#).

Pour les utilisateurs Windows

Lors de l'installation de Git pour Windows, il est recommandé de choisir l'option « Git Bash » afin que Git tourne dans son propre shell. Ce tutoriel part du principe que vous l'avez installé de cette façon.

Si vous ne connaissez pas bien Git, vous pouvez toujours obtenir des informations sur les commandes (une fois installé) en tapant `git help` sur une ligne de commande.

Téléchargement d'une copie de la version de développement de Django

La première étape pour contribuer à Django est d'obtenir une copie de son code source. À partir de la ligne de commande, utilisez la commande `cd` pour vous placer dans le répertoire où vous souhaitez mettre votre copie locale de Django.

Téléchargez le dépôt du code source de Django en utilisant la commande suivante :

```
$ git clone https://github.com/django/django.git
```

Maintenant que vous disposez d'une copie locale de Django, vous pouvez l'installer comme tout autre paquet à l'aide de `pip`. La façon la plus simple de le faire est d'utiliser un *environnement virtuel* (ou `virtualenv`) qui est une fonctionnalité de Python permettant d'isoler dans un répertoire dédié les paquets installés pour chacun de vos projets afin qu'ils n'interfèrent pas entre eux.

Il est conseillé de conserver tous les environnements virtuels au même endroit, par exemple dans `.virtualenvs/` dans votre dossier personnel. Créez-le s'il n'existe pas encore :

```
$ mkdir ~/.virtualenvs
```

Créez maintenant un nouveau `virtualenv` en exécutant :

```
$ python3 -m venv ~/.virtualenvs/djangodev
```

Le chemin correspond à l'endroit où le nouvel environnement sera enregistré sur votre ordinateur.

Pour les utilisateurs Windows

L'utilisation du module intégré `venv` ne fonctionnera pas si vous utilisez aussi le shell Git Bash sur Windows, car les scripts d'activation ne sont créés que pour le shell système (`.bat`) et PowerShell (`.ps1`). Utilisez plutôt le paquet `virtualenv`:

```
$ pip install virtualenv
$ virtualenv ~/.virtualenvs/djangodev
```

Pour les utilisateurs d'Ubuntu

Sur certaines versions d'Ubuntu, la commande ci-dessous pourrait échouer. Utilisez alors plutôt le paquet `virtualenv`, en vous assurant d'abord de disposer de `pip3`:

```
$ sudo apt-get install python3-pip
$ # Prefix the next command with sudo if it gives a permission denied error
$ pip3 install virtualenv
$ virtualenv --python=`which python3` ~/.virtualenvs/djangodev
```

La dernière étape de préparation de l'environnement virtuel est de l'activer :

```
$ source ~/.virtualenvs/djangodev/bin/activate
```

Si la commande `SOURCE` n'est pas disponible, vous pouvez essayer d'utiliser un point à la place :

```
$ . ~/.virtualenvs/djangodev/bin/activate
```

Pour les utilisateurs Windows

Pour activer votre virtualenv sur Windows, lancez :

```
$ source ~/virtualenvs/djangodev/Scripts/activate
```

Vous devez activer le virtualenv chaque fois que vous ouvrez une nouvelle fenêtre de terminal.

[virtualenvwrapper](#) est un outil utile pour rendre cette opération plus directe.

À partir de cet instant, tout ce que vous installez avec `pip` sera installé dans votre nouvel environnement virtuel, isolé des autres environnements et des paquets du système. De plus, le nom de l'environnement actuellement actif est affiché sur la ligne de commande pour vous aider à voir d'un coup d'œil lequel est activé. Poursuivez en installant la copie de Django précédemment chargée :

```
$ pip install -e /path/to/your/local/clone/django/
```

La version installée de Django correspond maintenant à votre copie locale. Vous constaterez immédiatement les effets de toute modification, ce qui est très utile lorsqu'on se met à écrire des correctifs.

Retour à une révision précédente de Django

Pour ce tutoriel, nous allons utiliser le ticket [#24788](#) comme étude de cas ; nous allons donc revenir en arrière dans l'historique Git de Django jusqu'au point où le correctif de ce ticket n'était pas encore appliqué. Cela nous permettra de parcourir toutes les étapes nécessaires à l'écriture de ce correctif à partir de zéro, y compris en lançant la suite de tests de Django.

Gardez à l'esprit que même si nous allons utiliser une révision plus ancienne de Django dans l'optique de ce tutoriel, vous devriez toujours utiliser la révision la plus actuelle du développement de Django lorsque vous travaillez sur le correctif réel d'un ticket !

Note

Le patch de ce ticket a été écrit par Paweł Marczewski et a été appliqué à Django dans le [commit 4df7e8483b2679fc1cba3410f08960bac6f51115](#). Par conséquent, nous utiliserons la révision de Django qui précède immédiatement, c'est-à-dire le [commit 4ccfc4439a7add24f8db4ef3960d02ef8ae09887](#).

Placez-vous dans le répertoire racine de Django (celui qui contient *django*, *docs*, *tests*, *AUTHORS*, etc.). Vous pouvez ensuite extraire l'ancienne révision de Django qui nous sera utile pour le tutoriel ci-dessous :

```
$ git checkout 4ccfc4439a7add24f8db4ef3960d02ef8ae09887
```

Lancement de la suite de test de Django pour la première fois

En contribuant à Django, il est très important que vos modifications de code n'introduisent pas de bogues dans d'autres parties de Django. Une des façons de contrôler que Django fonctionne toujours après vos modifications est de lancer la suite de tests de Django. Si tous les tests passent, il est raisonnable de penser que vous n'avez pas complètement cassé Django. Si vous n'avez encore jamais lancé la suite de tests de Django, il est recommandé de le faire une fois simplement pour vous familiariser avec ce que l'on obtient d'habitude quand tout va bien.

Avant de lancer la suite de tests, installez ses dépendances en vous plaçant d'abord dans le répertoire *tests/* de Django et en exécutant :

```
$ pip install -r requirements/py3.txt
```

Nous sommes maintenant prêts à lancer la suite de tests. Si vous utilisez GNU/Linux, Mac OS X ou une autre version de Unix, lancez :

```
$ ./runtests.py
```

Maintenant prenez place et détendez-vous. La suite complète de tests Django contient plus de 9600 tests différents, ce qui peut prendre entre 5 et 15 minutes en fonction de la rapidité de votre ordinateur.

Pendant que la suite de tests de Django s'exécute, vous voyez un flux de caractères représentant le statut de chaque test exécuté. E indique qu'une erreur est survenue durant le test et F indique qu'une assertion de test a échoué. Les deux sont considérés comme des échecs. Quant à eux, x et S indiquent respectivement des échecs attendus et des tests omis volontairement. Chaque point signifie qu'un test a réussi.

Les tests omis volontairement sont typiquement dus à des bibliothèques externes manquantes et nécessaires pour exécuter ces tests ; voir [Running all the tests](#) pour une liste de dépendances. Prenez soin d'installer toute dépendance de test qui est liée aux modifications que vous appliquez (nous n'en utiliserons pas pour ce tutoriel).

Once the tests complete, you should be greeted with a message informing you whether the test suite passed or failed. Since you haven't yet made any changes to Django's code, the entire test suite **should** pass. If you get failures or errors make sure you've followed all of the previous steps properly. See [Running the unit tests](#) for more information. If you're using Python 3.5+, there will be a couple failures related to deprecation warnings that you can ignore. These failures have since been fixed in Django.

Il faut savoir que la version « trunk » (la plus actuelle) de Django n'est pas toujours stable. Lorsque vous développez sur cette version, vous pouvez vérifier les [compilations d'intégration continue de Django](#) pour savoir si les échecs que vous obtenez sont spécifiques à votre environnement ou s'ils sont aussi présents dans les compilations officielles de Django. Si vous cliquez sur une compilation particulière, vous voyez la « matrice de configurations » qui indique les échecs précisément par version de Python et par moteur de base de données.

Note

Pour ce tutoriel et le ticket sur lequel nous travaillons, il est suffisant de tester avec SQLite. Cependant, il est possible (et parfois nécessaire) de [lancer les tests en utilisant une base de données différente](#).

Écriture de quelques tests pour le ticket

Dans la majorité des cas, un correctif doit contenir des tests pour être accepté dans Django. Pour les correctifs de correction de bogue, cela implique la rédaction d'un test de régression pour s'assurer que le bogue ne soit plus réintroduit plus tard dans Django. Un test de régression doit être écrit de façon à échouer tant que le bogue est présent et à réussir lorsque le bogue a été corrigé. En ce qui concerne les correctifs contenant de nouvelles fonctionnalités, vous devrez inclure des tests qui assurent que les nouvelles fonctionnalités fonctionnent correctement. Ces tests devraient aussi échouer avant que la fonctionnalité ne soit développée, et réussir après qu'elle a été implémentée.

Une bonne manière de faire cela est d'écrire d'abord les nouveaux tests, avant de toucher au code. Ce style de développement est appelé [développement piloté par les tests](#) et peut être appliqué aussi bien à des projets complets qu'à des correctifs individuels. Après avoir écrit les tests, vous les lancez pour être sûr qu'ils échouent effectivement (puisque vous n'avez pas encore corrigé l'erreur ou ajouté la fonctionnalité). Si vos nouveaux tests n'échouent pas, vous devez les corriger jusqu'à ce qu'ils échouent. Après tout, un test de régression qui passe que le bogue soit présent ou non n'est pas très utile pour éviter que ce bogue ne réapparaisse à l'avenir.

Entrons maintenant dans le vif du sujet.

Écriture de tests pour le ticket #24788

Le ticket [#24788](#) propose l'ajout d'une petite fonctionnalité : la possibilité de définir l'attribut de classe `prefix` pour les classes de formulaires, afin que :

```
[...] forms which ship with apps could effectively namespace themselves such
that N overlapping form fields could be POSTed at once and resolved to the
correct form.
```

Dans le but de résoudre ce ticket, nous allons ajouter un attribut `prefix` à la classe `BaseForm`. Lors de la création d'instances de cette classe, la transmission d'un préfixe à la méthode `__init__()` va toujours définir ce préfixe pour l'instance créée. Mais si aucun préfixe n'est transmis (ou `None`), c'est le préfixe de classe qui sera utilisé. Cependant, avant d'effectuer ces changements, nous allons écrire

quelques tests pour vérifier que notre modification fonctionne et qu'elle continue à fonctionner correctement à l'avenir.

Placez-vous dans le dossier `tests/forms_tests/tests/` et ouvrez le fichier `test_forms.py`. Ajoutez le code suivant à la ligne 1674, juste avant la fonction `test_forms_with_null_boolean`:

```
def test_class_prefix(self):
    # Prefix can be also specified at the class level.
    class Person(Form):
        first_name = CharField()
        prefix = 'foo'

    p = Person()
    self.assertEqual(p.prefix, 'foo')

    p = Person(prefix='bar')
    self.assertEqual(p.prefix, 'bar')
```

Ce nouveau test contrôle que la définition d'un préfixe au niveau de la classe fonctionne comme attendu et que la transmission d'un paramètre `prefix` au moment de la création d'une instance continue aussi de fonctionner.

Mais cette histoire de tests semble plutôt compliquée...

Si vous n'avez jamais eu affaire à des tests précédemment, ils peuvent paraître un peu difficiles à écrire au premier abord. Heureusement, les tests sont un sujet *très* important en programmation, il existe donc beaucoup d'informations à leur sujet :

- Une première approche conseillée dans l'écriture de tests pour Django est de parcourir la documentation [Écriture et lancement de tests](#).
- Dive Into Python (un livre en ligne gratuit pour les développeurs débutants en Python) contient une très bonne section sur l'[introduction aux tests unitaires](#).
- After reading those, if you want something a little meatier to sink your teeth into, there's always the Python [unittest](#) documentation.

Lancement des nouveaux tests

Rappelez-vous que nous n'avons encore effectué aucune modification à `BaseForm`, ce qui fait que nos tests vont échouer. Lançons tous les tests dans le dossier `forms_tests` pour être certain que c'est vraiment ce qui va se passer. À partir de la ligne de commande, placez-vous dans le répertoire `tests/` de Django et lancez :

```
$ ./runtests.py forms_tests
```

Si les tests se sont déroulés correctement, vous devriez voir un échec correspondant à la méthode de test que nous avons ajoutée. Si tous les tests ont réussi, vous devriez alors vérifier que vous avez bien ajouté les nouveaux tests ci-dessus dans le bon répertoire et la bonne classe.

Écriture du code pour le ticket

Nous allons maintenant ajouter la fonctionnalité Django décrite dans le ticket [#24788](#).

Écriture du code pour le ticket #24788

Placez-vous dans le dossier `django/django/forms/` et ouvrez le fichier `forms.py`. Trouvez la classe `BaseForm` à la ligne 72 et ajoutez l'attribut de classe `prefix` juste après l'attribut `field_order`:

```
class BaseForm(object):
    # This is the main implementation of all the Form logic. Note that this
    # class is different than Form. See the comments by the Form class for
    # more information. Any improvements to the form API should be made to
    # *this* class, not to the Form class.
    field_order = None
    prefix = None
```

Vérification du succès des tests

Après avoir terminé de modifier Django, nous devons être sûrs que les tests précédemment écrits réussissent, afin de vérifier que le code que nous venons d'écrire fonctionne correctement. Pour lancer les tests du dossier `forms_tests`, placez-vous dans le répertoire `tests/` de Django et lancez :

```
$ ./runtests.py forms_tests
```

Oups, heureusement que nous avons écrit ces tests ! Vous devriez toujours voir un échec avec l'exception suivante :

```
AssertionError: None != 'foo'
```

Nous avons oublié d'ajouter l'instruction conditionnelle dans la méthode `__init__`. Poursuivez en modifiant la ligne `self.prefix = prefix` qui se trouve maintenant à la ligne 87 de `django/forms/forms.py`, en y ajoutant l'instruction conditionnelle :

```
if prefix is not None:
    self.prefix = prefix
```

Relancez les tests et tout devrait réussir. Si ce n'est pas le cas, assurez-vous d'avoir bien modifié la classe `BaseForm` tel qu'indiqué ci-dessus et que vous avez copié le nouveau test correctement.

Lancement de la suite de tests de Django pour la seconde fois

Après avoir vérifié que votre correctif et les tests fonctionnent correctement, il est recommandé de relancer toute la suite de tests de Django, simplement pour contrôler que vos modifications n'ont pas introduit des bogues dans d'autres parties de Django. Même si quand la suite de tests passe entièrement, ce n'est toujours pas une garantie que votre code ne contient pas d'erreurs, cela aide tout de même à révéler de nombreux bogues et régressions qui pourraient être passés inaperçus.

Pour lancer la suite complète de tests de Django, placez-vous dans le répertoire `tests/` de Django et lancez :

```
$ ./runtests.py
```

Tant que vous ne voyez pas apparaître d'erreurs, tout va pour le mieux.

Écriture de la documentation

Ceci est une nouvelle fonctionnalité, elle doit donc être documentée. Ajoutez la section ci-après à la ligne 1068 (en fin de fichier) de `django/docs/ref/forms/api.txt`:

The prefix can also be specified on the form class::

```
>>> class PersonForm(forms.Form):
...     ...
...     prefix = 'person'
```

```
.. versionadded:: 1.9
```

The ability to specify ``prefix`` on the form class was added.

Comme cette nouvelle fonctionnalité sera présente dans une version à venir, nous la mentionnons aussi dans les notes de publication de Django 1.9, à la ligne 164 en-dessous de la section « Formulaires » dans le fichier `docs/releases/1.9.txt`:

```
* A form prefix can be specified inside a form class, not only when
  instantiating a form. See :ref:`form-prefix` for details.
```

Pour plus d'informations sur l'écriture de documentation, y compris une explication sur la signification de `versionadded`, consultez [Writing documentation](#). Cette page contient aussi une explication sur la manière de produire une copie locale de la documentation afin de pouvoir prévisualiser le code HTML qui sera généré.

Création du correctif contenant les modifications

C'est maintenant le moment de générer un fichier correctif qui pourra être envoyé à Trac ou appliqué à une autre copie de Django. Pour jeter un œil sur le correctif, lancez la commande suivante :

```
$ git diff
```

L'affichage montre les différences entre votre copie actuelle de Django (incluant vos modifications) et la révision que vous avez initialement extraite précédemment dans ce tutoriel.

Pour quitter l'affichage des différences, appuyez sur la touche `q` pour revenir à la ligne de commande. Si le contenu du correctif vous a semblé correct, vous pouvez lancer la commande suivante pour enregistrer le fichier correctif dans votre répertoire de travail actuel :

```
$ git diff > 24788.diff
```

Vous devriez maintenant voir un fichier nommé `24788.diff` dans le répertoire racine de Django. Ce fichier correctif contient toutes vos modifications et devrait ressembler à ceci :

```
diff --git a/django/forms/forms.py b/django/forms/forms.py
index 509709f..d1370de 100644
--- a/django/forms/forms.py
+++ b/django/forms/forms.py
@@ -75,6 +75,7 @@ class BaseForm(object):
     # information. Any improvements to the form API should be made to *this*
     # class, not to the Form class.
     field_order = None
+    prefix = None

     def __init__(self, data=None, files=None, auto_id='id_%s', prefix=None,
                  initial=None, error_class=ErrorList, label_suffix=None,
@@ -83,7 +84,8 @@ class BaseForm(object):
     self.data = data or {}
     self.files = files or {}
     self.auto_id = auto_id
-    self.prefix = prefix
+    if prefix is not None:
+        self.prefix = prefix
     self.initial = initial or {}
     self.error_class = error_class
     # Translators: This is the default suffix added to form field labels
diff --git a/docs/ref/forms/api.txt b/docs/ref/forms/api.txt
index 3bc39cd..008170d 100644
--- a/docs/ref/forms/api.txt
+++ b/docs/ref/forms/api.txt
@@ -1065,3 +1065,13 @@ You can put several Django forms inside one ``<form>`` tag.
To give each
    >>> print(father.as_ul())
    <li><label for="id_father-first_name">First name:</label> <input type="text"
name="father-first_name" id="id_father-first_name" /></li>
    <li><label for="id_father-last_name">Last name:</label> <input type="text"
name="father-last_name" id="id_father-last_name" /></li>
+
+The prefix can also be specified on the form class::
+
+    >>> class PersonForm(forms.Form):
+    ...     ...
+    ...     prefix = 'person'
+
+.. versionadded:: 1.9
+
+    The ability to specify ``prefix`` on the form class was added.
diff --git a/docs/releases/1.9.txt b/docs/releases/1.9.txt
index 5b58f79..f9bb9de 100644
```

```

--- a/docs/releases/1.9.txt
+++ b/docs/releases/1.9.txt
@@ -161,6 +161,9 @@ Forms
:attr:`~django.forms.Form.field_order` attribute, the ``field_order``
constructor argument , or the :meth:`~django.forms.Form.order_fields` method.

+* A form prefix can be specified inside a form class, not only when
+ instantiating a form. See :ref:`form-prefix` for details.
+
Generic Views
^^^^^^^^^^^^^^^^

diff --git a/tests/forms_tests/tests/test_forms.py
b/tests/forms_tests/tests/test_forms.py
index 690f205..e07fae2 100644
--- a/tests/forms_tests/tests/test_forms.py
+++ b/tests/forms_tests/tests/test_forms.py
@@ -1671,6 +1671,18 @@ class FormsTestCase(SimpleTestCase):
    self.assertEqual(p.cleaned_data['last_name'], 'Lennon')
    self.assertEqual(p.cleaned_data['birthday'], datetime.date(1940, 10, 9))

+    def test_class_prefix(self):
+        # Prefix can be also specified at the class level.
+        class Person(Form):
+            first_name = CharField()
+            prefix = 'foo'
+
+        p = Person()
+        self.assertEqual(p.prefix, 'foo')
+
+        p = Person(prefix='bar')
+        self.assertEqual(p.prefix, 'bar')
+
    def test_forms_with_null_boolean(self):
        # NullBooleanField is a bit of a special case because its presentation
(widget)
        # is different than its data. This is handled transparently, though.

```

Quelle est la prochaine étape ?

Félicitations, vous avez produit votre tout premier correctif pour Django ! En ayant franchi cette étape avec succès, vous pouvez désormais mettre en œuvre votre nouvelle compétence en aidant à améliorer le code de Django. La production de correctifs et leur envoi dans des tickets Trac est utile ; cependant, comme nous utilisons Git, nous recommandons l'adoption d'un [processus plus orienté sur Git](#).

Comme nous n'avons pas commité localement nos modifications, effectuez ce qui suit pour faire revenir votre branche git à un bon point de départ :

```

$ git reset --hard HEAD
$ git checkout master

```

Plus d'informations pour les nouveaux contributeurs

Avant de vous impliquer plus intensément dans l'écriture de correctifs pour Django, il est souhaitable de parcourir un peu plus d'informations destinées aux contributeurs :

- Il est important de lire la documentation de Django sur l'[appropriation de tickets et la soumission de correctifs](#). Cela comprend les règles de comportement sur Trac, la façon de vous assigner un ticket, le style attendu du code des correctifs et encore bien d'autres détails importants.
- Les contributeurs débutants devraient aussi lire la [documentation de Django pour les contributeurs débutants](#). Elle contient de nombreux bons conseils pour ceux d'entre nous qui débutent dans la contribution à Django.
- Après cela, si vous êtes toujours avide de plus d'informations sur la contribution, vous pouvez toujours parcourir le reste de la [documentation de Django sur la contribution](#). Elle contient plein d'informations utiles et devrait constituer votre source principale de réponses aux questions que vous vous posez.

Recherche de votre premier vrai ticket

Après avoir consulté une partie de ces informations, vous êtes fin prêt pour partir à la recherche d'un ticket pour lequel vous allez écrire un correctif. Commencez en priorité par les tickets marqués comme « Easy pickings » (résolution facile). Ces tickets sont souvent plus simples par nature et sont une cible idéale pour les contributeurs débutants. Lorsque vous serez à l'aise avec la contribution à Django, vous pourrez alors passer à la rédaction de correctifs pour des tickets plus difficiles.

Si vous voulez commencer tout de suite (et personne ne vous le reprochera), jetez un coup d'œil à la liste des [tickets simples qui ont besoin d'un correctif](#) et des [tickets simples qui ont des correctifs nécessitant une amélioration](#). Si vous êtes à l'aise dans l'écriture de tests, vous pouvez aussi examiner la liste des [tickets simples qui ont besoin de tests](#). N'oubliez pas de suivre les instructions sur l'attribution de tickets qui étaient mentionnées dans le lien sur la documentation de Django sur l'[appropriation de tickets et la soumission de correctifs](#).

Et ensuite ?

Quand un ticket possède un correctif, il doit être relu par une deuxième paire d'yeux. Après l'envoi d'un correctif ou la soumission d'une « pull request », prenez soin de mettre à jour les métadonnées du ticket en définissant les drapeaux du ticket « has patch », « doesn't need tests », etc, pour que d'autres puissent les retrouver en vue d'une relecture. Contribuer ne signifie pas nécessairement écrire un correctif à partir de zéro. La relecture de correctifs existants est également une contribution très utile. Consultez [Triaging tickets](#) pour plus de détails.

[Quelle lecture pour la suite](#)
[Utilisation de Django](#)

Utilisation de Django

Introductions à toutes les parties essentielles de Django qu'il faut connaître :

- [Comment installer Django](#)
- [Modèles et bases de données](#)
- [Gestion des requêtes HTTP](#)
- [Utilisation des formulaires](#)
- [Gabarits](#)
- [Vues fondées sur les classes](#)
- [Migrations](#)
- [Gestion des fichiers](#)
- [Les tests dans Django](#)
- [Authentification des utilisateurs dans Django](#)
- [L'infrastructure de cache dans Django](#)
- [Traitement conditionnel de vue](#)
- [Signatures cryptographiques](#)
- [Envoi de messages électroniques](#)
- [Internationalisation et régionalisation](#)
- [Journalisation](#)
- [Pagination](#)
- [Passage à Python 3](#)
- [La sécurité dans Django](#)
- [Performance et optimisations](#)
- [Sérialisation d'objets Django](#)
- [Les réglages de Django](#)
- [Signaux](#)
- [Infrastructure de contrôle du système](#)
- [Paquets externes](#)

[Écriture de votre premier correctif pour Django](#)
[Comment installer Django](#)

Comment installer Django

Ce document a pour but de vous préparer à faire fonctionner Django.

Installation de Python

Django étant un applicatif Web Python, il a donc besoin de Python. Voir [Quelle version de Python puis-je utiliser avec Django ?](#) pour plus de détails.

Obtenez la dernière version de Python à l'adresse <https://www.python.org/download/> ou par l'intermédiaire du gestionnaire des paquets de votre système.

Django sur Jython

Si vous utilisez [Jython](#) (une implémentation de Python pour machine Java), vous devrez suivre quelques étapes supplémentaires. Voyez [Fonctionnement de Django sur Jython](#) pour les détails.

Python sur Windows

Si vous êtes débutant avec Django et que vous utilisez Windows, cette documentation peut être utile : [Comment installer Django avec Windows](#).

Installation de Apache et mod_wsgi

Si vous voulez juste faire quelques expériences avec Django, passez directement à la section suivante ; Django inclut un serveur Web léger utilisable pour les tests, il n'est donc pas utile de configurer Apache tant que vous ne souhaitez pas déployer Django en production.

Si vous souhaitez utiliser Django sur un site en production, utilisez [Apache](#) avec [mod_wsgi](#). mod_wsgi peut fonctionner dans deux modes : un mode intégré et un mode démon. Dans le mode intégré, mod_wsgi est comparable à mod_perl, il intègre Python dans Apache et charge le code Python en mémoire au démarrage du serveur. Le code reste en mémoire tout au long du cycle de vie du processus Apache, ce qui apporte des améliorations de performance significatives en comparaison d'autres configurations. En mode démon, mod_wsgi délègue la gestion des requêtes à un processus démon indépendant. Ce processus peut fonctionner sous un nom d'utilisateur différent de celui du serveur Web, ce qui peut améliorer la sécurité ; d'autre part, le processus peut être redémarré sans devoir redémarrer tout le serveur Web Apache, ce qui peut simplifier la mise à jour de code. Consultez la documentation de mod_wsgi pour déterminer le mode qui vous conviendra le mieux. Assurez-vous que Apache est installé et que son module mod_wsgi est activé. Django fonctionne avec toute version d'Apache qui prend en charge mod_wsgi.

Consultez [Comment utiliser Django avec mod_wsgi](#) pour plus d'informations sur la façon de configurer mod_wsgi après l'avoir installé.

Si vous n'avez pas la possibilité d'utiliser mod_wsgi, ne désespérez pas : Django prend en charge beaucoup d'autres scénarios de déploiement. L'un d'entre eux est [uWSGI](#) ; il fonctionne très bien avec

[nginx](#). De plus, Django respecte la spécification WSGI ([PEP 3333](#)) ce qui permet de le faire fonctionner sur diverses plates-formes serveurs.

Mise en route de la base de données

Si vous prévoyez d'utiliser la fonctionnalité base de données de Django, vous devez faire le nécessaire pour qu'une base de données soit fonctionnelle. Django sait gérer plusieurs serveurs de bases de données différents ; officiellement, il prend en charge [PostgreSQL](#), [MySQL](#), [Oracle](#) et [SQLite](#).

Si vous développez un projet simple ou quelque chose que vous ne prévoyez pas de déployer dans un environnement de production, SQLite est généralement l'option la plus simple car elle n'exige pas de serveur séparé. Cependant, SQLite comporte beaucoup de différences en comparaison d'autres bases de données et si vous travaillez sur un projet un peu conséquent, il est recommandé de développer avec la même base de données qui sera utilisée en production.

En plus des bases de données prises en charge officiellement, il existe des [moteurs externes à Django](#) qui permettent d'utiliser d'autres bases de données avec Django.

En plus du moteur de base de données, il est aussi nécessaire de vérifier que les liaisons Python pour la base de données sont installées.

- Si vous utilisez PostgreSQL, vous avez besoin du paquet [psycopg2](#). Vous pouvez consulter les [notes sur PostgreSQL](#) pour plus de détails.
- Si vous utilisez MySQL, vous aurez besoin d'une [API de driver de BDD](#) comme `mysqlclient`. Voir [notes pour le backend MySQL](#) pour plus de détails.
- Si vous utilisez SQLite, il est recommandé de lire les [notes du moteur SQLite](#).
- Si vous utilisez Oracle, vous aurez besoin d'une copie de [cx Oracle](#), mais veuillez lire les [notes spécifiques pour le moteur de base de données Oracle](#) pour les détails concernant les versions prises en charge d'Oracle et de `cx_Oracle`.
- Si vous utilisez un moteur externe non officiel, consultez la documentation qui l'accompagne pour d'éventuels contraintes supplémentaires.

Si vous pensez utiliser la commande de Django `manage.py migrate` pour créer automatiquement les tables de base de données pour vos modèles (après la fin de l'installation de Django et la création d'un projet), vous devez vous assurer que Django possède les permissions de créer et modifier les tables dans la base de données que vous utilisez ; si vous pensez créer manuellement les tables, il suffit alors d'attribuer à Django les permissions `SELECT`, `INSERT`, `UPDATE` et `DELETE`. Après avoir créé un utilisateur de base de données possédant ces permissions, il s'agira d'indiquer les détails de connexion dans le fichier de réglages de votre projet, voir [DATABASES](#) pour les détails.

Si vous utilisez l'[infrastructure de test](#) de Django pour tester les requêtes de bases de données, Django aura besoin des permissions nécessaires pour créer une base de données de test.

Désinstallation des anciennes versions de Django

Si vous mettez à jour votre installation de Django à partir d'une version précédente, il est nécessaire de d'abord désinstaller l'ancienne version de Django avant d'installer la nouvelle.

Si l'ancienne version de Django a été installée avec [pip](#) ou `easy_install`, l'installation de la nouvelle version en utilisant le même outil ([pip](#) ou `easy_install`) provoquera automatiquement la désinstallation de la version précédente, vous n'avez donc rien à faire de particulier.

Si l'ancienne version de Django a été installée avec `python setup.py install`, sa désinstallation se résume à supprimer le répertoire `django` du répertoire `site-packages` de votre installation Python. Pour trouver le répertoire à supprimer, vous pouvez exécuter la commande suivante dans le shell d'un terminal (pas dans un shell Python interactif) :

```
$ python -c "import django; print(django.__path__)"
```

Installation du code de Django

Les instructions d'installation divergent légèrement selon que vous installez un paquet de votre distribution, que vous téléchargez la dernière version officielle ou que vous récupériez la dernière version en développement.

C'est facile, quelle que soit la méthode choisie.

Installation d'une version officielle avec

C'est la façon recommandée d'installer Django.

1. Installez [pip](#). La manière la plus simple est d'utiliser l'[installateur pip autonome](#). Si votre distribution contient déjà une version installée de `pip`, il pourrait être nécessaire de la mettre à jour si elle est trop ancienne. Dans ce cas, vous le verrez bien car l'installation ne fonctionnera pas.
2. Jetez un œil à [virtualenv](#) et [virtualenvwrapper](#). Ces outils fournissent des environnements Python isolés qui sont bien plus pratiques que d'installer des paquets au niveau de tout le système. Ils permettent aussi d'installer des paquets sans avoir besoin de privilèges administrateurs. Le [tutoriel de contribution](#) vous guide pour la création d'un environnement `virtualenv` avec Python 3.
3. Après avoir créé et activé un environnement virtuel, saisissez la commande `pip install Django` dans le terminal.

Installation d'un paquet de la distribution

Parcourez les [notes spécifiques aux distributions](#) pour vérifier que votre plate-forme/distribution propose des paquets ou installateurs officiels de Django. Les paquets fournis par les distributions

permettent généralement de profiter de l'installation automatique des dépendances et de mises à niveau plus faciles ; toutefois, ces paquets correspondent rarement aux dernières versions de Django.

Installation de la version de développement

Suivi du développement de Django

Si vous décidez d'utiliser la dernière version de développement de Django, il est recommandé de surveiller attentivement l'[activité de développement](#) de même que les [notes de publication de la version à venir](#). Cela vous aidera à vous tenir au courant de toute nouvelle fonctionnalité que vous pourriez exploiter ou des modifications que vous devrez effectuer dans votre code lors de la mise à jour de votre copie de Django (pour les versions stables, toute modification nécessaire est documentée dans les notes de publication).

Si vous aimeriez pouvoir mettre à jour occasionnellement votre version de Django avec les dernières corrections et améliorations, suivez ces instructions :

1. Vérifiez que [Git](#) est installé et que vous pouvez lancer ses commandes depuis un terminal (saisissez `git help` à l'invite de commande pour le tester).
2. Créez une copie de travail de la branche principale de développement de Django comme ceci :

```
$ git clone git://github.com/django/django.git
```

Un répertoire `django` sera créé dans le répertoire actuel.

3. Vérifiez que l'interpréteur Python peut charger le code de Django. La façon la plus pratique de faire cela est d'utiliser [virtualenv](#), [virtualenvwrapper](#), et [pip](#). Le [tutoriel de contribution](#) vous guide pour la création d'un environnement `virtualenv` avec Python 3.
4. Après avoir configuré et activé l'environnement virtuel, exécutez la commande suivante :

```
$ pip install -e django/
```

Ceci rendra le code Django importable et mettra aussi à disposition la commande utilitaire `django-admin`. En d'autres mots, vous serez fin prêt !

Lorsque vous souhaitez mettre à jour votre copie du code source de Django, lancez simplement la commande `git pull` à partir du répertoire `django`. Quand vous faites cela, Git télécharge automatiquement toutes les modifications.

[Utilisation de Django](#)

[Modèles et bases de données](#)

Modèles et bases de données

Un modèle est la source de données unique et définitive à propos de vos données. Il contient les champs et le comportement essentiels des données que vous stockez. Généralement, chaque modèle correspond à une seule table de base de données.

- [Modèles](#)
- [Création de requêtes](#)
- [Agrégation](#)
- [Gestionnaires](#)
- [Lancement de requêtes SQL brutes](#)
- [Transactions de base de données](#)
- [Bases de données multiples](#)
- [Espaces de tables](#)
- [Optimisation de l'accès à la base de données](#)
- [Exemples d'utilisation de l'API de relations entre modèles](#)

[Comment installer Django](#)
[Modèles](#)

Modèles

Un modèle est la source d'information unique et définitive à propos de vos données. Il contient les champs et le comportement essentiels des données que vous stockez. Généralement, chaque modèle correspond à une seule table de base de données.

Les bases :

- Chaque modèle est une classe Python qui hérite de [django.db.models.Model](#).
- Chaque attribut du modèle représente un champ de base de données.
- Avec tout ça, Django vous offre une API d'accès à la base de données générée automatiquement ; voir [Création de requêtes](#).

Exemple rapide

Cet exemple de modèle définit une personne (Person) avec un prénom (`first_name`) et un nom (`last_name`) :

```
from django.db import models
```

```
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` et `last_name` sont des [champs](#) du modèle. Chaque champ est défini comme un attribut de classe, et chaque attribut correspond à une colonne de base de données.

Le modèle `Person` ci-dessus va créer une table de base de données comme celle-ci :

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Quelques notes techniques :

- Le nom de la table, `myapp_person`, est automatiquement dérivé de certaines métadonnées du modèle mais peut être surchargé. Consultez [Noms de tables](#) pour plus de détails.
- Un champ `id` est ajouté automatiquement, mais ce comportement peut être adapté. Voir [Champs clé primaire automatiques](#).
- Le code SQL `CREATE TABLE` de cet exemple est mis en forme avec la syntaxe PostgreSQL, mais il est utile de relever que Django utilise du code SQL adapté au moteur de base de données indiqué dans votre [fichier de réglages](#).

Utilisation des modèles

Après avoir défini les modèles, il faut indiquer à Django que vous souhaitez *utiliser* ces modèles. Vous pouvez le faire en éditant votre fichier de réglages et en modifiant le réglage [INSTALLED_APPS](#) pour y ajouter le nom du module qui contient `models.py`.

Par exemple, si le modèle de votre application se trouve dans le module `myapp.models` (la structure de paquet créée pour une application par le script [manage.py startapp](#)), [INSTALLED_APPS](#) devrait contenir :

```
INSTALLED_APPS = [
    #...
    'myapp',
    #...
]
```

Lorsque vous ajoutez de nouvelles applications à [INSTALLED_APPS](#), n'oubliez pas ensuite de lancer [manage.py migrate](#), en créant éventuellement d'abord des migrations pour ces applications avec [manage.py makemigrations](#).

Champs

La partie la plus importante d'un modèle (et la seule qui soit obligatoire) est la liste des champs de base de données qu'il définit. Les champs sont définis par des attributs de classe. Faites attention à ne pas choisir des noms de champs qui entrent en conflit avec l'[API des modèles](#) comme `clean`, `save` ou `delete`.

Exemple :

```
from django.db import models

class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

Types de champs

Chaque champ de votre modèle doit être une instance de la classe [Field](#) appropriée. Django utilise les types des classes de champs pour déterminer un certain nombre de choses :

- Le type de la colonne, qui indique à la base de données le genre de données à stocker (par ex. `INTEGER`, `VARCHAR`, `TEXT`).
- Le [composant HTML](#) par défaut à utiliser lors de la création d'un champ de formulaire (par ex. : `<input type="text">`, `<select>`).
- Les exigences minimales de validation, utilisées dans l'administration de Django et dans les formulaires générés automatiquement.

Django est fourni avec des dizaines de types de champs intégrés ; vous trouverez la liste complète dans la [référence des champs de modèle](#). Vous pouvez facilement écrire vos propres champs si ceux proposés par Django ne font pas l'affaire ; voir [Écriture de champs de modèles personnalisés](#).

Options des champs

Chaque champ accepte un certain nombre de paramètres spécifiques (documentés dans la [référence des champs de modèle](#)). Par exemple, [CharField](#) (et ses sous-classes) exige un paramètre [max_length](#) indiquant la taille du champ de base de données `VARCHAR` qui stockera les données.

Il existe aussi un ensemble de paramètres communs à tous les types de champs. Ils sont tous facultatifs. Ils sont décrits en détails dans la [référence](#), mais voici un résumé rapide de ceux qui sont le plus souvent utilisés :

null

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

blank

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

Notez que ce n'est pas la même chose que null. null est purement lié à la base de données alors que blank est lié à la validation. Si un champ possède blank=True, la validation de formulaire permet la saisie d'une valeur vide. Si un champ possède blank=False, le champ est obligatoire.

choices

Un itérable (une liste ou un tuple) de tuples à 2 valeurs à utiliser comme liste de choix pour ce champ. Quand ce paramètre est présent, le composant de formulaire par défaut est une liste déroulante au lieu du champ de texte standard et seules les valeurs proposées dans la liste seront acceptées.

Une liste de choix ressemble à ceci :

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

Le premier élément de chaque tuple est la valeur qui sera stockée dans la base de données. Le second élément est celui qui est affiché dans le composant de formulaire par défaut ou dans un ModelChoiceField. Étant donné une instance de modèle, la valeur d'affichage d'un champ comportant des choix peut être accédée en utilisant la méthode `get_FOO_display()`. Par exemple :

```
from django.db import models  
  
class Person(models.Model):  
    SHIRT_SIZES = (  
        ('S', 'Small'),  
        ('M', 'Medium'),  
        ('L', 'Large'),  
    )  
    name = models.CharField(max_length=60)  
    shirt_size = models.CharField(max_length=1, choices=SHIRT_SIZES)  
  
>>> p = Person(name="Fred Flintstone", shirt_size="L")  
>>> p.save()  
>>> p.shirt_size
```



```
'L'  
>>> p.get_shirt_size_display()  
'Large'
```

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.

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.

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 `IntegerField` 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](#).

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. Par exemple :

```
from django.db import models  
  
class Fruit(models.Model):  
    name = models.CharField(max_length=100, primary_key=True)  
  
>>> fruit = Fruit.objects.create(name='Apple')  
>>> fruit.name = 'Pear'  
>>> fruit.save()  
>>> Fruit.objects.values_list('name', flat=True)  
['Apple', 'Pear']
```

unique

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

Répétons encore une fois que ce n'était que de courtes descriptions des options de champs les plus courantes. Des détails complets peuvent être obtenus dans la [référence des options communes des champs de modèle](#).

Champs clé primaire automatiques

Par défaut, Django ajoute à chaque modèle le champ suivant :

```
id = models.AutoField(primary_key=True)
```

C'est une clé primaire avec incrémentation automatique.

Si vous souhaitez spécifier une clé primaire personnalisée, il suffit d'ajouter `primary_key=True` à l'un de vos champs. Si Django voit que vous avez explicitement défini `Field.primary_key`, il n'ajoutera pas de colonne `id` automatique.

Pour chaque modèle, il est obligatoire d'avoir un et un seul champ ayant le paramètre `primary_key=True` (qu'il soit déclaré explicitement ou ajouté automatiquement).

Noms de champs verbeux

Chaque type de champ, à l'exception de `ForeignKey`, `ManyToManyField` et `OneToOneField`, accepte un premier paramètre positionnel facultatif, un nom verbeux. Si le nom verbeux n'est pas défini, Django le crée automatiquement en se basant sur le nom d'attribut du champ, remplaçant les soulignements par des espaces.

Dans cet exemple, le nom verbeux est "person's first name":

```
first_name = models.CharField("person's first name", max_length=30)
```

Dans cet exemple, le nom verbeux est "first name":

```
first_name = models.CharField(max_length=30)
```

`ForeignKey`, `ManyToManyField` et `OneToOneField` exigent que le premier paramètre soit une classe de modèle, il est donc nécessaire d'utiliser le paramètre nommé `verbose_name`:

```
poll = models.ForeignKey(
    Poll,
    on_delete=models.CASCADE,
    verbose_name="the related poll",
)
sites = models.ManyToManyField(Site, verbose_name="list of sites")
place = models.OneToOneField(
    Place,
    on_delete=models.CASCADE,
    verbose_name="related place",
)
```

La convention est de ne pas mettre en majuscule la première lettre de `verbose_name`. Django le fera automatiquement là où il pense que c'est nécessaire.

Relations

Clairement, la puissance des bases de données relationnelles se trouve dans les liaisons entre tables. Django propose des méthodes pour définir les trois types de relations les plus courantes : plusieurs-à-un, plusieurs-à-plusieurs et un-à-un.

Relations plusieurs-à-un

Pour définir une relation plusieurs-à-un, utilisez [django.db.models.ForeignKey](#). Son utilisation est pareille à celle des autres types de champs [Field](#) : il s'agit d'un attribut de classe d'un modèle.

[ForeignKey](#) exige un paramètre positionnel : la classe à laquelle le modèle est lié.

Par exemple, si un modèle `Car` (voiture) possède un `Manufacturer` (fabriquant), c'est-à-dire qu'un `Manufacturer` peut produire plusieurs voitures mais chaque `Car` n'a qu'un seul `Manufacturer`, voici le code correspondant :

```
from django.db import models

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

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

Il est aussi possible de créer des [relations récursives](#) (un objet avec une relation plusieurs-à-un avec lui-même) et des [relations vers des modèles non encore définis](#) ; consultez la [référence des champs de modèle](#) pour plus de détails.

Il est recommandé mais non obligatoire que le nom d'un champ [ForeignKey](#) (manufacturer dans l'exemple ci-dessus) soit égal au nom du modèle en minuscules. Vous pouvez évidemment nommer le champ de la manière qui vous convient. Par exemple :

```
class Car(models.Model):
    company_that_makes_it = models.ForeignKey(
        Manufacturer,
        on_delete=models.CASCADE,
    )
    # ...
```

Voir aussi

Les champs [ForeignKey](#) acceptent quelques paramètres supplémentaires qui sont présentés dans la [référence des champs de modèle](#). Ces options aident à spécifier le fonctionnement de la relation ; toutes sont facultatives.

Pour plus de détails sur l'accès aux objets par référence liée inverse, consultez l'[exemple de l'accès aux relations inverses](#).

Pour des exemples de code, consultez les [exemples de relations plusieurs-à-un entre modèles](#).

Relations plusieurs-à-plusieurs

Pour définir une relation plusieurs-à-plusieurs, utilisez [`django.db.models.ManyToManyField`](#). Son utilisation est pareille à celle des autres types de champs [`Field`](#) : il s'agit d'un attribut de classe d'un modèle.

[`ManyToManyField`](#) exige un paramètre positionnel : la classe à laquelle le modèle est lié.

Par exemple, si une `Pizza` possède plusieurs objets `Topping` (garniture), c'est-à-dire qu'un `Topping` peut se trouver sur plusieurs pizzas et chaque `Pizza` possède plusieurs garnitures, voici comment ce cas de figure serait représenté :

```
from django.db import models

class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

Comme pour [`ForeignKey`](#), il est aussi possible de créer des [relations récursives](#) (un objet avec une relation plusieurs-à-plusieurs avec lui-même) et des [relations vers des modèles non encore définis](#).

Il est recommandé mais non obligatoire que le nom d'un champ [`ManyToManyField`](#) (`toppings` dans l'exemple ci-dessus) soit un pluriel décrivant l'ensemble des objets modèles liés.

Que le champ [`ManyToManyField`](#) soit placé sur un modèle ou l'autre ne change pas grand chose, mais il est essentiel de ne le mettre que dans un des modèles, pas dans les deux.

Généralement, les instances [`ManyToManyField`](#) devraient être placées dans l'objet qui va être modifié par un formulaire. Dans l'exemple ci-dessus, les *toppings* sont dans `Pizza` (plutôt que ce soit `Topping` qui possède un champ [`ManyToManyField`](#) de pizzas) parce qu'il est plus logique d'imaginer une pizza ayant plusieurs garnitures qu'une garniture se trouvant dans plusieurs pizzas. De la manière dont les choses ont été définies ci-dessus, le formulaire `Pizza` permettra de choisir des garnitures.

Voir aussi

Consultez les [exemples de relations plusieurs-à-plusieurs entre modèles](#) pour voir un exemple complet.

Les champs [`ManyToManyField`](#) acceptent aussi quelques paramètres supplémentaires qui sont présentés dans la [référence des champs de modèle](#). Ces options aident à spécifier le fonctionnement de la relation ; toutes sont facultatives.

Champs supplémentaires dans les relations plusieurs-à-plusieurs

Lorsque vous devez gérer des relations plusieurs-à-plusieurs simples comme le mélange et la combinaison de pizzas et de garnitures, un [ManyToManyField](#) standard suffit. Cependant, il est parfois nécessaire d'associer des données à la relation entre deux modèles.

Par exemple, considérez le cas d'une application faisant le lien entre des musiciens et les groupes musicaux auxquels ils appartiennent. Une relation plusieurs-à-plusieurs existe entre une personne et les groupes dont elle est membre, il est donc possible d'utiliser un champ [ManyToManyField](#) pour représenter cette relation. Cependant, il y a de nombreux détails au sujet de l'appartenance au groupe qu'il peut être intéressant de conserver, comme par exemple la date à laquelle la personne a rejoint le groupe.

Pour ces situations, Django permet d'indiquer le modèle qui sera utilisé pour spécifier la relation plusieurs-à-plusieurs. Il est alors possible de définir des champs supplémentaires dans le modèle intermédiaire. Ce dernier est associé au champ [ManyToManyField](#) en utilisant le paramètre [through](#) qui va pointer vers le modèle agissant comme intermédiaire. Dans notre exemple de musique, le code pourrait ressembler à ceci :

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):
        # __unicode__ on Python 2
        return self.name

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

    def __str__(self):
        # __unicode__ on Python 2
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

Lorsque vous définissez le modèle intermédiaire, vous indiquez explicitement les clés étrangères vers les modèles impliqués dans la relation plusieurs-à-plusieurs. Cette déclaration explicite définit comment les deux modèles sont reliés.

Il y a quelques restrictions concernant le modèle intermédiaire :

- Le modèle intermédiaire doit contenir une et une seule clé étrangère vers le modèle source (qui correspondrait à `Group` dans notre exemple), ou alors vous devez explicitement indiquer à Django les clés étrangères à employer pour les relations en utilisant [ManyToManyField.through_fields](#). S'il y a plus d'une clé étrangère et que

`through_fields` n'est pas précisé, une erreur de validation est produite. C'est exactement le même procédé pour la clé étrangère vers le modèle cible (qui correspondrait à `Person` dans notre exemple).

- Pour un modèle ayant une relation plusieurs-à-plusieurs avec lui-même au travers d'un modèle intermédiaire, deux clés étrangères vers le même modèle sont autorisées, mais elles seront considérées comme les deux (différentes) parties de la relation plusieurs-à-plusieurs. Mais s'il y a *plus* de deux clés étrangères, vous devez aussi indiquer `through_fields` comme ci-dessus, sinon une erreur de validation sera générée.
- Lors de la définition d'une relation plusieurs-à-plusieurs d'un modèle vers lui-même au travers d'un modèle intermédiaire, vous devez employer le paramètre [`symmetrical=False`](#) (voir la [référence des champs de modèle](#)).

Après avoir configuré le champ [`ManyToManyField`](#) afin qu'il utilise le modèle intermédiaire (`Membership`, dans ce cas), tout est prêt pour commencer à créer des relations plusieurs-à-plusieurs. Cela s'effectue en créant des instances du modèle intermédiaire :

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason="Needed a new drummer.")
>>> m1.save()
>>> beatles.members.all()
[<Person: Ringo Starr>]
>>> ringo.group_set.all()
[<Group: The Beatles>]
>>> m2 = Membership.objects.create(person=paul, group=beatles,
...     date_joined=date(1960, 8, 1),
...     invite_reason="Wanted to form a band.")
>>> beatles.members.all()
[<Person: Ringo Starr>, <Person: Paul McCartney>]
```

Au contraire des champs plusieurs-à-plusieurs normaux, il n'est *pas* possible d'utiliser `add`, `create` ou l'attribution (par ex. `beatles.members = [...]`) pour créer des relations :

```
# THIS WILL NOT WORK
>>> beatles.members.add(john)
# NEITHER WILL THIS
>>> beatles.members.create(name="George Harrison")
# AND NEITHER WILL THIS
>>> beatles.members = [john, paul, ringo, george]
```

Pourquoi ? Il n'est pas possible de simplement créer une relation entre une `Person` et un `Group`, car il est nécessaire d'indiquer tous les détails de la relation imposés par le modèle `Membership`. Les appels simples `add`, `create` ainsi que l'attribution ne permettent pas d'indiquer les détails supplémentaires. En conséquence, ils sont désactivés pour les relations plusieurs-à-plusieurs qui

utilisent un modèle intermédiaire. La seule manière de créer ce type de relation est de créer des instances du modèle intermédiaire.

La méthode [remove\(\)](#) est désactivée pour les mêmes raisons. Toutefois, la méthode [clear\(\)](#) peut être utilisée pour enlever toutes les relations plusieurs-à-plusieurs d'une instance :

```
>>> # Beatles have broken up
>>> beatles.members.clear()
>>> # Note that this deletes the intermediate model instances
>>> Membership.objects.all()
[]
```

Après avoir établi des relations plusieurs-à-plusieurs en créant des instances du modèle intermédiaire, il est possible d'effectuer des requêtes. Comme pour une relation plusieurs-à-plusieurs normale, les requêtes peuvent utiliser les attributs du modèle lié à la relation plusieurs-à-plusieurs :

```
# Find all the groups with a member whose name starts with 'Paul'
>>> Group.objects.filter(members__name__startswith='Paul')
[<Group: The Beatles>]
```

Comme vous utilisez un modèle intermédiaire, la requête peut aussi exploiter ses attributs :

```
# Find all the members of the Beatles that joined after 1 Jan 1961
>>> Person.objects.filter(
...     group__name='The Beatles',
...     membership__date_joined__gt=date(1961,1,1))
[<Person: Ringo Starr>]
```

Si vous avez besoin d'accéder aux informations d'appartenance au groupe, c'est possible de le faire en interrogeant directement le modèle `Membership`:

```
>>> ringos_membership = Membership.objects.get(group=beatles, person=ringo)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

Une autre façon d'accéder à la même information est d'interroger la [relation plusieurs-à-plusieurs inverse](#) à partir d'un objet `Person`:

```
>>> ringos_membership = ringo.membership_set.get(group=beatles)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

Relations un-à-un

Pour définir une relation un-à-un, utilisez [OneToOneField](#). Son utilisation est pareille à celle des autres types de champs `Field`: il s'agit d'un attribut de classe d'un modèle.

Son utilisation principale concerne la clé primaire d'un objet lorsque cet objet « complète » un autre objet d'une certaine manière.

[OneToOneField](#) exige un paramètre positionnel : la classe à laquelle le modèle est lié.

Par exemple, si vous construisiez une base de données d'« emplacements » (Place dans l'exemple), il s'agirait de mettre en place des éléments assez standards comme l'adresse, le numéro de téléphone, etc. dans la base de données. Ensuite, si vous souhaitez construire une base de données de restaurants au-dessus de la base des emplacements, au lieu de vous répéter et de répliquer tous ces champs dans le modèle Restaurant, il serait possible de concevoir Restaurant avec un champ [OneToOneField](#) vers Place (parce qu'un restaurant « est un » emplacement ; en fait, pour gérer cela, vous feriez probablement appel à de l'[héritage](#), ce qui implique une relation un-à-un implicite).

Comme pour [ForeignKey](#), il est aussi possible de créer des [relations récursives](#) et des [relations vers des modèles non encore définis](#).

Voir aussi

Consultez les [exemples de relations un-à-un entre modèles](#) pour voir un exemple complet.

Les champs [OneToOneField](#) acceptent aussi un paramètre [parent_link](#) facultatif.

Précédemment, les classes [OneToOneField](#) devenaient automatiquement la clé primaire d'un modèle. Cela n'est plus le cas (même si il est toujours possible de passer manuellement le paramètre [primary_key](#) si vous le voulez). Il est donc possible d'avoir plusieurs champs [OneToOneField](#) dans un seul modèle.

Modèles dans plusieurs fichiers

Il est parfaitement admis de lier un modèle à celui d'une autre application. Pour cela, importez le modèle à lier au sommet du fichier où votre modèle est défini. Puis, il suffit de faire référence à cette autre classe de modèle là où c'est nécessaire. Par exemple :

```
from django.db import models
from geography.models import ZipCode

class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(
        ZipCode,
        on_delete=models.SET_NULL,
        blank=True,
        null=True,
    )
```

Restrictions sur les noms de champs

Django ne soumet les noms de champs de modèles qu'à deux restrictions :

1. Un nom de champ ne peut pas être un mot réservé de Python, car il en résulterait une erreur de syntaxe Python. Par exemple :

```
class Example(models.Model):
```



```
pass = models.IntegerField() # 'pass' is a reserved word!
```

2. Un nom de champ ne peut pas contenir deux soulignements suivis, en raison du fonctionnement de la syntaxe Django pour les requêtes. Par exemple :

```
class Example(models.Model):  
    foo__bar = models.IntegerField() # 'foo__bar' has two underscores!
```

Ces limites peuvent être contournées, car le nom de champ n'est pas nécessairement égal au nom de la colonne de base de données. Voir l'option [db_column](#).

Les mots réservés SQL, comme `join`, `where` ou `select` *sont* autorisés comme noms de champs de modèle, car Django échappe tous les noms de tables ou de colonnes de base de données dans toutes les requêtes SQL sous-jacentes. Il utilise la syntaxe de guillemets propre au moteur de base de données utilisé.

Types de champs personnalisés

Si aucun des champs de modèle existants ne convient à vos besoins, ou si vous souhaitez tirer profit d'un type de colonne de base de données plus spécialisé, vous pouvez créer votre propre classe de champ. Une documentation complète sur la création de champs personnalisés se trouve dans [Écriture de champs de modèles personnalisés](#).

Options de Meta

Vous pouvez attribuer des métadonnées à votre modèle en utilisant une classe `Meta` imbriquée, comme ceci :

```
from django.db import models  
  
class Ox(models.Model):  
    horn_length = models.IntegerField()  
  
    class Meta:  
        ordering = ["horn_length"]  
        verbose_name_plural = "oxen"
```

Les métadonnées de modèles sont « tout ce qui n'est pas un champ », comme les options de tri ([ordering](#)), le nom de la table de base de données ([db_table](#)) ou des noms verbeux singulier et pluriel ([verbose_name](#) et [verbose_name_plural](#)). Aucune n'est obligatoire et la présence de `class Meta` dans un modèle est entièrement facultative.

Une liste complète de toutes les options `Meta` possibles se trouve dans la [référence des options de modèle](#).

Les attributs de modèle

objects

L'attribut le plus important d'un modèle est le [Manager](#). Il s'agit de l'interface par laquelle les modèles Django ont accès aux opérations de requêtes vers la base de données et par laquelle les [instances de modèles sont construites](#) à partir de la base de données. Si aucun objet Manager (gestionnaire) personnalisé n'est indiqué, le nom par défaut est [objects](#). Les gestionnaires ne sont accessibles qu'au travers des classes de modèle, et pas des instances de modèle.

Méthodes des modèles

Pour ajouter des fonctionnalités de « niveau ligne » à vos objets, définissez des méthodes personnalisées dans le modèle. Alors que les méthodes de [Manager](#) sont prévues pour agir au niveau des tables, les méthodes de modèles agissent plutôt sur une instance particulière d'un modèle.

C'est une technique importante pour conserver la logique métier à un seul endroit, le modèle.

Par exemple, ce modèle possède quelques méthodes personnalisées :

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def baby_boomer_status(self):
        "Returns the person's baby-boomer status."
        import datetime
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        elif self.birth_date < datetime.date(1965, 1, 1):
            return "Baby boomer"
        else:
            return "Post-boomer"

    def _get_full_name(self):
        "Returns the person's full name."
        return '%s %s' % (self.first_name, self.last_name)
    full_name = property(_get_full_name)
```

La dernière méthode de cet exemple est une [propriété](#).

La [référence des instances de modèles](#) présente une liste complète des [méthodes automatiquement héritées par chaque modèle](#). Vous pouvez surcharger la plupart d'entre elles, voir [surcharge des méthodes de modèles prédéfinies](#) ci-dessous, mais vous souhaitez presque toujours définir certaines dans vos modèles :

[__str__\(\)](#) (Python 3)

Une « méthode magique » de Python qui renvoie une « représentation » unicode d'un objet. C'est ce que Python et Django utilisent chaque fois qu'une instance de modèle doit être transformée en une simple chaîne en vue d'être affichée. C'est par exemple le cas lorsqu'un objet doit être affiché dans une console interactive ou dans l'interface d'administration.

Cette méthode devrait toujours être définie ; la valeur par défaut n'est vraiment pas très utile.

`__unicode__()` (Python 2)

Équivalent Python 2 de `__str__()`.

[`get_absolute_url\(\)`](#)

Cette méthode indique à Django comment produire l'URL d'un objet. Django l'utilise dans son interface d'administration ainsi qu'à chaque fois qu'il a besoin de connaître l'URL d'un objet.

Chaque objet possédant un URL permettant de l'identifier de manière unique devrait définir cette méthode.

Surcharge des méthodes de modèles prédéfinies

Un autre groupe de [méthodes de modèles](#) recouvrent un ensemble de comportements liés à la base de données et sont susceptibles d'être personnalisées. En particulier, il est assez fréquent de vouloir modifier le fonctionnement de [save\(\)](#) et de [delete\(\)](#).

Vous êtes libre de surcharger ces méthodes (et toute autre méthode de modèle) pour modifier leur comportement.

Un cas d'utilisation classique de surcharge des méthodes intégrées est lorsque vous souhaitez effectuer une action lors de l'enregistrement d'un objet. Par exemple (voir [save\(\)](#) pour de la documentation sur les paramètres acceptés) :

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        do_something()
        super(Blog, self).save(*args, **kwargs) # Call the "real" save() method.
        do_something_else()
```

Vous pouvez aussi empêcher l'enregistrement :

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        if self.name == "Yoko Ono's blog":
            return # Yoko shall never have her own blog!
        else:
            super(Blog, self).save(*args, **kwargs) # Call the "real" save()
method.
```

Il est important de ne pas oublier d'appeler la méthode de la classe parente (c'est l'affaire de `super(Blog, self).save(*args, **kwargs)`) pour s'assurer que l'objet soit bien enregistré dans la base de données. Si vous omettez d'appeler la méthode parente, le comportement par défaut n'est pas appliqué et la base de données ne sera pas affectée.

Il est aussi important de transmettre les paramètres qui peuvent être acceptés par la méthode du modèle, c'est le rôle de `*args`, `**kwargs`. De temps à autre, Django étend les possibilités des méthodes de modèles intégrées en ajoutant de nouveaux paramètres. Si vous utilisez `*args`, `**kwargs` dans vos définitions de méthodes, vous êtes certain que votre code gérera automatiquement ces paramètres quand ils seront ajoutés.

Les méthodes de modèle surchargées ne sont pas appelées dans les opérations groupées

Notez que la méthode [`delete\(\)`](#) d'un objet n'est pas nécessairement appelée lorsque les [objets sont supprimés en vrac en utilisant un QuerySet](#) ou par le truchement d'une [suppression en cascade](#). Pour être certain que la logique de suppression personnalisée soit exécutée, vous pouvez utiliser les signaux [`pre_delete`](#) ou [`post_delete`](#).

Malheureusement, il n'y a pas de solution possible lors de la [création](#) ou de la [mise à jour](#) d'objets en vrac, puisque ni [`save\(\)`](#), ni [`pre_save`](#), ni [`post_save`](#) ne sont appelés.

Exécution de SQL personnalisé

Un autre usage assez fréquent est d'écrire des commandes SQL personnalisées dans les méthodes de modèles et les méthodes au niveau module. Pour plus de détails sur l'utilisation de SQL brut, consultez la documentation sur l'[utilisation de SQL brut](#).

Héritage de modèle

Dans Django, l'héritage de modèle fonctionne de manière presque identique à l'héritage des classes tel qu'il se pratique en Python, mais les éléments de base présentés au début de cette page sont toujours de mise. Cela signifie que la classe de base doit être une sous-classe de [`django.db.models.Model`](#).

La seule décision qui vous revient est de savoir si vous voulez que les modèles parents soient des modèles à part entière (avec leur propre table de base de données) ou si les parents ne sont que des conteneurs d'informations partagées qui ne seront visibles qu'au travers de leurs modèles enfants.

Il existe trois types d'héritage possibles avec Django.

1. Souvent, vous voulez simplement que la classe parente contienne des informations que vous ne souhaitez pas ressaisir dans chaque modèle enfant. Cette classe ne sera jamais utilisée pour elle-même, il s'agit donc de [Classes de base abstraites](#).
2. Si vous héritez d'un modèle existant (provenant peut-être d'une toute autre application) et que vous souhaitez que chaque modèle dispose de sa propre table de base de données, il s'agit de [Héritage multi-table](#).

3. Finalement, si vous souhaitez uniquement modifier le comportement d'un modèle dans son code Python sans toucher aux champs de modèle, il s'agit de [Modèles mandataires](#).

Classes de base abstraites

Les classes de base abstraites sont utiles lorsque vous souhaitez regrouper certaines informations communes à un ensemble de modèles. Vous rédigez la classe de base et indiquez `abstract=True` dans sa classe [Meta](#). Aucune table de base de données ne sera créée pour ce modèle. Par contre, lorsqu'elle sera utilisée comme classe de base pour d'autres modèles, ses champs seront ajoutés à ceux de la classe enfant. Il est considéré comme une erreur que d'avoir des champs de la classe de base abstraite ayant le même nom que ceux de la classe enfant (et Django lèvera une exception).

Un exemple :

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

Le modèle `Student` aura trois champs : `name`, `age` et `home_group`. Le modèle `CommonInfo` ne peut pas être utilisé comme un modèle Django normal, car c'est une classe de base abstraite. Ce modèle ne génère pas de table de base de données et ne possède pas de gestionnaire ; il ne peut pas être instancié ni enregistré directement.

Dans beaucoup de situations, c'est ce type d'héritage de modèle qui convient. Il offre la possibilité de regrouper les informations communes au niveau Python, tout en ne créant qu'une table de base de données par modèle enfant.

Héritage de Meta

Lorsqu'une classe de base abstraite est créée, Django rend disponible toute classe imbriquée [Meta](#) déclarée dans la classe de base comme attribut. Si une classe enfant ne déclare pas sa propre classe [Meta](#), elle hérite de la classe [Meta](#) de son parent. Si la classe enfant souhaite étendre la classe [Meta](#) du parent, elle peut en hériter. Par exemple :

```
from django.db import models

class CommonInfo(models.Model):
    # ...
    class Meta:
        abstract = True
        ordering = ['name']

class Student(CommonInfo):
```

```
# ...
class Meta(CommonInfo.Meta):
    db_table = 'student_info'
```

Django effectue une seule modification à la classe [Meta](#) d'une classe de base abstraite : avant d'installer l'attribut [Meta](#), il définit `abstract=False`. Cela signifie que les enfants des classes de base abstraites ne deviennent pas automatiquement des classes abstraites elles-mêmes. Il est bien sûr possible de créer une classe de base abstraite qui hérite d'une autre classe abstraite. Il faut simplement se rappeler de définir explicitement `abstract=True` à chaque fois.

Certains attributs n'ont pas de bonne raison d'être intégrés dans la classe [Meta](#) d'une classe de base abstraite. Par exemple, la présence de `db_table` signifierait que toutes les classes enfants (celles qui ne contiennent pas leur propre classe [Meta](#)) utiliseraient la même table de base de données, ce qui ne serait certainement pas le comportement souhaité.

Soyez prudent avec `related_name`

Si vous utilisez l'attribut [related_name](#) d'un champ `ForeignKey` ou `ManyToManyField`, vous devez toujours indiquer un nom inverse *unique* pour le champ. Cela produirait normalement un problème dans les classes de base abstraites, puisque les champs de cette classe sont inclus dans toutes les classes enfants avec exactement les mêmes valeurs d'attributs (y compris [related_name](#)) à chaque fois.

Pour contourner ce problème, lorsque vous utilisez [related_name](#) dans une classe de base abstraite (et dans ce cas seulement), des parties du nom doivent contenir `'%(app_label)s'` et `'%(class)s'`.

- `'%(class)s'` est remplacé par le nom en minuscules de la classe enfant dans laquelle est utilisé le champ.
- `'%(app_label)s'` est remplacé par le nom en minuscules de l'application dans laquelle la classe enfant est contenue. Chaque nom d'application installée doit être unique et les noms de classes de modèles dans chaque application doivent également être uniques, il est donc garanti que le nom résultant sera chaque fois différent.

Par exemple, étant donné une application `common/models.py`:

```
from django.db import models

class Base(models.Model):
    m2m = models.ManyToManyField(OtherModel, related_name="% (app_label)s_%
(class)s_related")

    class Meta:
        abstract = True

class ChildA(Base):
    pass
```

```
class ChildB(Base):
    pass
```

Accompagnée d'une autre application `rare/models.py`:

```
from common.models import Base

class ChildB(Base):
    pass
```

Le nom inverse du champ `common.ChildA.m2m` sera `common_childa_related`, alors que le nom inverse du champ `common.ChildB.m2m` sera `common_childb_related`, et finalement, le nom inverse du champ `rare.ChildB.m2m` sera `rare_childb_related`. Vous êtes libre de choisir comment utiliser les portions `'%(class)s'` et `'%(app_label)s'` pour construire le nom lié, mais si vous oubliez de les inclure, Django signalera des erreurs lors du lancement des vérifications systèmes (ou en lançant [migrate](#)).

Si vous n'indiquez pas d'attribut [related_name](#) pour un champ d'une classe de base abstraite, le nom inverse par défaut sera le nom de la classe enfant suivi par `'_set'`, comme cela se serait produit si vous aviez directement déclaré le champ dans la classe enfant. Par exemple, dans le code ci-dessus, si l'attribut [related_name](#) avait été omis, le nom inverse du champ `m2m` aurait été `childa_set` dans le cas `ChildA` et `childb_set` pour le champ `ChildB`.

Héritage multi-table

Le deuxième type d'héritage de modèle pris en charge par Django est lorsque chaque modèle d'une hiérarchie est lui-même un modèle à part entière. À chaque modèle correspond une table de base de données qui peut être interrogée et créée individuellement. Cette relation d'héritage introduit des liens entre le modèle enfant et chacun de ses parents (via un champ [OneToOneField](#) automatique). Par exemple :

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

Tous les champs de `Place` seront aussi disponible dans `Restaurant`, même si les données se trouveront dans des tables de base de données différentes. Ainsi, ces deux formes sont possibles :

```
>>> Place.objects.filter(name="Bob's Cafe")
>>> Restaurant.objects.filter(name="Bob's Cafe")
```

Si vous avez une `Place` qui est aussi un `Restaurant`, vous pouvez accéder à l'objet `Restaurant` depuis l'objet `Place` en utilisant le nom du modèle en minuscules :

```
>>> p = Place.objects.get(id=12)
# If p is a Restaurant object, this will give the child class:
>>> p.restaurant
<Restaurant: ...>
```

Cependant, si dans l'exemple ci-dessus *p* n'était pas un `Restaurant` (soit parce qu'il a été créé directement comme un objet `Place` ou qu'il est le parent d'une autre classe), l'accès à `p.restaurant` aurait généré une exception `Restaurant.DoesNotExist`.

Meta et l'héritage multi-table

Dans le cas de l'héritage multi-table, l'héritage de la classe [Meta](#) du parent n'a aucun intérêt pour la classe enfant. Toutes les options [Meta](#) ont déjà été appliquées à la classe parente et une nouvelle application n'amènerait qu'à des comportements contradictoires (a contrario du cas de la classe de base abstraite où la classe de base n'existe pas pour elle-même).

Ainsi, un modèle enfant n'a pas accès à la classe [Meta](#) de son parent. Cependant, il existe quelques cas limités où l'enfant hérite du comportement de son parent : si l'enfant n'indique pas d'attribut [ordering](#) ou [get_latest_by](#), il hérite des attributs correspondants de son parent.

Si le parent possède un ordre de tri mais que vous ne souhaitez pas que l'enfant ait un ordre de tri quelconque, vous pouvez le désactiver explicitement :

```
class ChildModel(ParentModel):
    # ...
    class Meta:
        # Remove parent's ordering effect
        ordering = []
```

Héritage et relations inverses

Comme l'héritage multi-table utilise un champ [OneToOneField](#) implicite pour lier l'enfant à son parent, il est possible d'accéder à l'enfant depuis le parent comme dans l'exemple ci-dessus. Cependant, il est fait usage du nom correspondant à la valeur par défaut de [related_name](#) pour les relations [ForeignKey](#) and [ManyToManyField](#). Si vous ajoutez ces types de relations dans la sous-classe du modèle parent, vous **devez** renseigner l'attribut [related_name](#) pour chacun de ces champs. Si vous l'oubliez, Django génère une erreur de validation.

Par exemple, en se basant toujours sur la classe `Place` ci-dessus, créons une autre sous-classe avec un champ [ManyToManyField](#):

```
class Supplier(Place):
    customers = models.ManyToManyField(Place)
```

Il en résulte une erreur :

```
Reverse query name for 'Supplier.customers' clashes with reverse query
name for 'Supplier.place_ptr'.
```

HINT: Add or change a `related_name` argument to the definition for

'Supplier.customers' or 'Supplier.place_ptr'.

L'ajout de `related_name` au champ `customers` comme ci-après permettrait de résoudre l'erreur : `models.ManyToManyField(Place, related_name='provider')`.

Indication du champ de lien vers le parent

Comme déjà mentionné, Django crée automatiquement une relation [OneToOneField](#) de la classe enfant vers tout modèle parent non abstrait. Si vous souhaitez contrôler le nom du champ de liaison vers le parent, vous pouvez ajouter votre propre champ [OneToOneField](#) et définir [parent_link=True](#) pour indiquer que ce champ est le champ de liaison vers la classe parente.

Modèles mandataires

Lorsqu'on utilise l'[héritage multi-table](#), une nouvelle table de base de données est créée pour chaque sous-classe d'un modèle. C'est normalement le comportement souhaité, dans la mesure où la sous-classe doit pouvoir stocker les champs de données supplémentaires qui ne figurent pas dans la classe de base. Cependant, dans certains cas, seul le comportement Python d'un modèle a besoin d'être modifié, peut-être pour changer le gestionnaire par défaut ou pour ajouter une nouvelle méthode.

C'est l'objectif de l'héritage de modèle mandataire : créer un mandataire (« proxy ») du modèle d'origine. Vous pouvez créer, supprimer et mettre à jour des instances du modèle mandataire et toutes les données seront enregistrées comme si vous utilisiez le modèle d'origine (non mandataire). La différence est que vous pouvez modifier certaines choses dans le modèle mandataire comme le tri par défaut des modèles ou le gestionnaire par défaut, sans devoir toucher au modèle original.

Les modèles mandataires sont déclarés comme des modèles normaux. Vous indiquez à Django qu'il s'agit de modèles mandataires en définissant l'attribut [proxy](#) de la classe `Meta` à `True`.

Par exemple, supposons que vous vouliez ajouter une méthode au modèle `Person`. Vous pouvez le faire de cette façon :

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

class MyPerson(Person):
    class Meta:
        proxy = True

    def do_something(self):
        # ...
        pass
```

La classe `MyPerson` opère sur la même table de base de données que sa classe parente `Person`. En particulier, toute nouvelle instance de `Person` sera aussi accessible au travers de `MyPerson` et inversement :

```
>>> p = Person.objects.create(first_name="foobar")
>>> MyPerson.objects.get(first_name="foobar")
<MyPerson: foobar>
```

Il est aussi possible d'utiliser un modèle mandataire pour définir un autre tri par défaut sur un modèle. Par exemple, vous ne voulez pas toujours trier le modèle `Person`, mais souvent le trier par `last_name` lorsque que vous utilisez le mandataire. C'est simple :

```
class OrderedPerson(Person):
    class Meta:
        ordering = ["last_name"]
        proxy = True
```

Dorénavant les requêtes sur `Person` ne seront pas triées mais les requêtes sur `OrderedPerson` seront triées par leur champ `last_name`.

Les requêtes `QuerySet` renvoient toujours le modèle interrogé

Il n'y a aucun moyen de demander à Django de renvoyer par exemple un objet `MyPerson` chaque fois que vous faites une requête sur des objets `Person`. Une requête sur des objets ``Person`` renvoie toujours des objets de ce même type. L'idée principale des objets mandataires est que le code se basant sur l'objet `Person` original utilise ces objets-là et que votre propre code peut utiliser les extensions que vous avez rajoutées (et qu'aucun autre code ne peut utiliser de toute manière). Ce n'est pas une façon de remplacer partout le modèle `Person` (ou un autre) par un autre modèle de votre conception.

Restrictions des classes de base

Un modèle mandataire ne peut hériter que d'une seule classe de modèle non abstraite. Il n'est pas possible d'hériter de plusieurs modèles non abstraits car le modèle mandataire ne fournit aucune connexion entre les lignes de différentes tables de base de données. Un modèle mandataire peut hériter d'autant de classes de modèle abstraites que nécessaire pourvu qu'elles ne définissent *pas* de champ de modèle.

Gestionnaires des modèles mandataires

Si vous n'indiquez aucun gestionnaire de modèle pour un modèle mandataire, il hérite des gestionnaires de ses modèles parents. Si vous définissez un gestionnaire sur le modèle mandataire, il devient le gestionnaire par défaut, bien que d'éventuels gestionnaires définis dans les classes parentes seront aussi disponibles.

En poursuivant l'exemple ci-dessus, vous pourriez modifier le gestionnaire par défaut utilisé lors des requêtes sur le modèle `Person` comme ceci :

```
from django.db import models

class NewManager(models.Manager):
    # ...
    pass
```

```
class MyPerson(Person):
    objects = NewManager()

    class Meta:
        proxy = True
```

Si vous souhaitez ajouter un nouveau gestionnaire au modèle mandataire sans remplacer celui qui est défini par défaut, vous pouvez employer les techniques décrites dans la documentation des [gestionnaires personnalisés](#) : créez une classe de base contenant les nouveaux gestionnaires et héritez de celle-ci après la classe de base principale :

```
# Create an abstract class for the new manager.
class ExtraManagers(models.Model):
    secondary = NewManager()

    class Meta:
        abstract = True

class MyPerson(Person, ExtraManagers):
    class Meta:
        proxy = True
```

Il est assez rare d'avoir besoin de le faire, mais si besoin est, c'est possible.

Differences between proxy inheritance and unmanaged models

L'héritage des modèles mandataires est en apparence très semblable à la création de modèles non pilotés, utilisant l'attribut [managed](#) de la classe `Meta` d'un modèle. Les deux alternatives ne sont pas totalement équivalentes et il vaut la peine d'évaluer laquelle correspond le mieux à votre besoin.

Une des différences est que vous pouvez (et vous devez même, au risque d'obtenir un modèle vide) définir les champs des modèles ayant `Meta.managed=False`. Il est possible, en définissant soigneusement [Meta.db_table](#), de créer un modèle non piloté masquant un modèle existant et d'y ajouter des méthodes Python. Cependant, cette solution est très répétitive et fragile dans la mesure où il faut manuellement synchroniser les deux en cas de modifications.

L'autre différence qui est plus importante pour les modèles mandataires est la façon dont les gestionnaires de modèles sont traités. Il est entendu que les modèles mandataires se comportent exactement de la même façon que leur modèle de base. Ils héritent des gestionnaires de modèles de leur parent, y compris du gestionnaire par défaut. Dans le cas habituel de l'héritage de modèle multi-table, les enfants n'héritent pas des gestionnaires de leur parent puisque les gestionnaires personnalisés ne sont pas toujours adéquats lorsque des champs supplémentaires entrent en jeu. La [documentation des gestionnaires](#) présente plus de détails au sujet de cette dernière situation.

Lorsque ces deux fonctionnalités ont été établies, il y a eu des tentatives d'en faire une seule solution. Il en est ressorti que les interactions avec l'héritage en général et avec les gestionnaires plus particulièrement rendaient l'API très compliquée et potentiellement difficile à comprendre et à utiliser. En conclusion, comme deux options étaient nécessaires dans tous les cas, c'est le scénario de la distinction actuelle qui a prévalu.

Ainsi, les règles générales sont :

1. Si vous reflétez un modèle ou une table de base de données existant et que vous ne souhaitez pas reproduire toutes les colonnes d'origine de la table, utilisez `Meta.managed=False`. Cette option est habituellement utile pour la modélisation de tables ou vues de base de données qui ne sont pas sous le contrôle de Django.
2. Si votre objectif est de ne modifier que le comportement Python d'un modèle tout en conservant les mêmes champs que l'original, utilisez `Meta.proxy=True`. Cette configuration assure que le modèle mandataire est une copie exacte de la structure de stockage du modèle de base lorsque des données sont enregistrées.

Héritage multiple

Tout comme l'héritage en Python, les modèles Django peuvent hériter de plusieurs modèles parents. N'oubliez pas que les règles Python habituelles de résolution de nom s'appliquent. La première classe de base dans laquelle apparaît un nom particulier (par ex. [Meta](#)) prévaut sur les autres apparitions ; par exemple, cela signifie que si plusieurs parents contiennent une classe [Meta](#), seule la première occurrence sera utilisée, et toutes les autres seront ignorées.

Généralement, il ne devrait pas être nécessaire d'hériter de plusieurs parents. La principale raison de le faire est dans le cas de classes « mix-in » : l'ajout d'un champ supplémentaire particulier ou d'une méthode dans chaque classe héritant de la classe « mix-in ». Essayez de garder votre hiérarchie d'héritage aussi simple et compréhensible que possible afin de ne pas devoir batailler au moment de retrouver la provenance d'une information ou d'un comportement spécifique.

Notez qu'en héritant de plusieurs modèles possédant un champ de clé primaire `id` commun, une erreur sera signalée. Pour un héritage multiple correct, vous pouvez utiliser un champ [AutoField](#) explicite dans les modèles de base :

```
class Article(models.Model):
    article_id = models.AutoField(primary_key=True)
    ...

class Book(models.Model):
    book_id = models.AutoField(primary_key=True)
    ...

class BookReview(Book, Article):
    pass
```

Ou utiliser un ancêtre commun contenant le champ [AutoField](#):

```
class Piece(models.Model):
    pass

class Article(Piece):
    ...

class Book(Piece):
```

```
...  
class BookReview(Book, Article):  
    pass
```

Le masquage de nom de champ n'est pas autorisé

Dans l'héritage habituel de classes Python, il est permis de surcharger n'importe quel attribut de la classe parent par la classe enfant. Avec Django, cela n'est pas autorisé pour les attributs qui sont des instances de [Field](#) (en tout cas pas pour le moment). Si une classe de base possède un champ `auteur`, il n'est pas possible de créer un autre champ de modèle `auteur` dans les classes qui héritent de cette classe de base.

La surcharge de champs de modèles parents pose des problèmes dans les domaines de l'initialisation de nouvelles instances (pour indiquer quel champ est initialisé dans `Model.__init__`) et de la sérialisation. Ce sont des situations qui n'affectent pas de la même manière l'héritage habituel de classes Python, cette différence entre l'héritage des classes Python et l'héritage des modèles Django n'est donc pas arbitraire.

Cette restriction ne s'applique qu'aux attributs qui sont des instances de [Field](#). Les attributs Python normaux peuvent être surchargés sans problème. Cela ne s'applique aussi qu'au nom de l'attribut tel que Python le voit : si vous indiquez manuellement le nom de la colonne de base de données, il est possible d'avoir le même nom de colonne apparaissant à la fois dans un enfant et son modèle parent dans de l'héritage multi-table (ce sont finalement des colonnes appartenant à deux tables de base de données différentes).

Django signale une erreur [FieldError](#) si vous surchargez un champ de modèle d'une classe parente.

Organizing models in a package

The [manage.py startapp](#) command creates an application structure that includes a `models.py` file. If you have many models, organizing them in separate files may be useful.

To do so, create a `models` package. Remove `models.py` and create a `myapp/models/` directory with an `__init__.py` file and the files to store your models. You must import the models in the `__init__.py` file.

For example, if you had `organic.py` and `synthetic.py` in the `models` directory:

```
myapp/models/__init__.py  
  
from .organic import Person  
from .synthetic import Robot
```

Explicitly importing each model rather than using `from .models import *` has the advantages of not cluttering the namespace, making code more readable, and keeping code analysis tools useful.

Voir aussi

[La référence des modèles](#)

Documente toutes les API liées aux modèles, y compris les champs de modèles, les objets liés et les requêtes (QuerySet).

[Modèles et bases de données](#)

[Création de requêtes](#)

Création de requêtes

Une fois les [modèles de données](#) créés, Django offre automatiquement une API d'abstraction de base de données qui permet de créer, obtenir, mettre à jour et supprimer des objets. Ce document explique comment utiliser cette API. Consultez la [référence des modèles de données](#) pour des détails complets sur toutes les options d'interrogation des modèles.

À travers ce guide (et dans la référence), nous nous référons aux modèles suivants, qui forment une application de blog :

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()

    def __str__(self):
        return self.name

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField()
    authors = models.ManyToManyField(Author)
    n_comments = models.IntegerField()
    n_pingbacks = models.IntegerField()
    rating = models.IntegerField()

    def __str__(self):
```

```
return self.headline
```

Création d'objets

Pour représenter des données d'une table de base de données en objets Python, Django utilise un système intuitif : une classe de modèle représente une table de base de données, et une instance de cette classe représente un enregistrement particulier dans la table de base de données.

Pour créer un objet, créez une instance de la classe de modèle en utilisant des paramètres nommés, puis appelez [save\(\)](#) pour l'enregistrer dans la base de données.

En supposant que les modèles se trouvent dans un fichier `mysite/blog/models.py`, voici un exemple :

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

Ceci exécute une commande SQL `INSERT` en arrière-plan. Django ne sollicite pas la base de données tant que vous n'appellez pas explicitement [save\(\)](#).

La méthode [save\(\)](#) ne renvoie aucune valeur.

Voir aussi

[save\(\)](#) accepte un certain nombre d'options avancées décrites ailleurs. Consultez la documentation de [save\(\)](#) pour les détails complets.

Pour créer et enregistrer un objet en une seule étape, utilisez la méthode [create\(\)](#).

Enregistrement des modifications d'objets

Pour enregistrer les modifications d'un objet existant dans la base de données, utilisez [save\(\)](#).

Considérant une instance `b5` d'un `Blog` ayant déjà été enregistrée dans la base de données, cet exemple modifie son nom et met à jour son enregistrement dans la base de données :

```
>>> b5.name = 'New name'
>>> b5.save()
```

Ceci génère une commande SQL `UPDATE` en arrière-plan. Django ne sollicite pas la base de données tant que vous n'appellez pas explicitement [save\(\)](#).

Enregistrement des champs `ForeignKey` et `ManyToManyField`

La mise à jour d'un champ [ForeignKey](#) fonctionne exactement de la même façon que l'enregistrement d'un champ normal, il suffit d'attribuer un objet du bon type au champ en question. Cet exemple met à jour l'attribut `blog` d'une instance `Entry` nommée `entry`, en supposant que les

instances appropriées de `Entry` et de `Blog` sont déjà enregistrées dans la base de données (afin de pouvoir être récupérées ci-dessous) :

```
>>> from blog.models import Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

La mise à jour d'un champ [ManyToManyField](#) fonctionne un peu différemment ; utilisez la méthode [add\(\)](#) du champ pour ajouter un enregistrement à la relation. Cet exemple ajoute l'instance `Author` `joe` à l'objet `entry`:

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

Pour ajouter plusieurs enregistrements à un champ [ManyToManyField](#) d'un seul coup, indiquez plusieurs paramètres dans l'appel à [add\(\)](#), comme ceci :

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

Django se plaint si vous essayez d'assigner ou d'ajouter un objet de type inapproprié.

Sélection d'objets

Pour extraire des objets de la base de données, construisez un [QuerySet](#) via un [Manager](#) de la classe de votre modèle.

Un [QuerySet](#) représente une collection d'objets de la base de données. Il peut comporter zéro, un ou plusieurs *filters*. Les filtres réduisent les résultats de requêtes en fonction des paramètres donnés. En termes SQL, un [QuerySet](#) équivaut à une commande `SELECT` et un filtre correspond à une clause restrictive telle que `WHERE` ou `LIMIT`.

On obtient un [QuerySet](#) en utilisant le [Manager](#) du modèle. Chaque modèle a au moins un [Manager](#) ; il s'appelle [objects](#) par défaut. On y accède directement via la classe du modèle, comme ceci :

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objects
Traceback:
...
AttributeError: "Manager isn't accessible via Blog instances."
```

Note

Les `Managers` ne sont accessibles que par les classes de modèle et non par les instances de modèle, afin d'accentuer la séparation entre les opérations sur les « tables » et les opérations sur les « enregistrements ».

Le [Manager](#) est la source principale des `QuerySets` d'un modèle. Par exemple, `Blog.objects.all()` renvoie un [QuerySet](#) contenant tous les objets `Blog` de la base de données.

Sélection de tous les objets

La façon la plus simple d'obtenir des objets d'une table est de tous les sélectionner. Pour ce faire, utilisez la méthode [all\(\)](#) d'un [Manager](#):

```
>>> all_entries = Entry.objects.all()
```

La méthode [all\(\)](#) renvoie un [QuerySet](#) de tous les objets dans la base de données.

Sélection d'objets spécifiques avec les filtres

Le [QuerySet](#) renvoyé par [all\(\)](#) contient tous les objets de la table de la base de données. Mais le plus souvent, seul un sous-ensemble de tous les objets devra être sélectionné.

Pour créer un tel sous-ensemble, vous affinez le [QuerySet](#) initial en y ajoutant des filtres de conditions. Les deux façons les plus utilisées pour affiner un [QuerySet](#) sont :

`filter(**kwargs)`

Renvoie un nouveau [QuerySet](#) contenant les objets qui répondent aux paramètres de recherche donnés.

`exclude(**kwargs)`

Renvoie un nouveau [QuerySet](#) contenant les objets qui ne répondent *pas* aux paramètres de recherche donnés.

Les paramètres de recherche (`**kwargs` dans les définitions de fonction ci-dessus) doivent être dans le format décrit dans [Recherches dans les champs](#) ci-dessous.

Par exemple, pour obtenir un [QuerySet](#) des articles de blog de l'année 2006, utilisez [filter\(\)](#) comme ceci :

```
Entry.objects.filter(pub_date__year=2006)
```

Avec la classe de gestionnaire par défaut, c'est équivalent à :

```
Entry.objects.all().filter(pub_date__year=2006)
```

Enchaînement des filtres

Le résultat de l'affinage d'un [QuerySet](#) est lui-même un [QuerySet](#), il est donc possible d'enchaîner les filtrages successifs. Par exemple :

```
>>> Entry.objects.filter(
...     headline__startswith='What '
... ).exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(
...     pub_date__gte=datetime(2005, 1, 30)
... )
```

À partir du [QuerySet](#) initial de toutes les lignes dans la base de données, le code ci-dessus ajoute un filtre, puis une exclusion, puis un autre filtre. Le résultat final est un [QuerySet](#) contenant tous les enregistrements ayant un titre (headline) commençant par « What », ayant été publiés entre le 30 janvier 2005 et aujourd'hui.

Les QuerySet filtrés sont uniques

À chaque fois que vous affinez un [QuerySet](#), vous obtenez un [QuerySet](#) tout neuf qui n'est lié d'aucune manière au [QuerySet](#) précédent. Chaque affinage crée un [QuerySet](#) séparé et distinct qui peut être stocké, utilisé et réutilisé.

Exemple :

```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

Ces trois [QuerySet](#) sont séparés. Le premier est un [QuerySet](#) de base contenant tous les enregistrements qui contiennent un titre commençant par « What ». Le second est un sous-ensemble du premier, avec un critère supplémentaire qui exclut les enregistrements dont le champ `pub_date` est aujourd'hui ou dans le futur. Le troisième est un sous-ensemble du premier, avec un critère supplémentaire qui sélectionne uniquement les enregistrements dont le champ `pub_date` est aujourd'hui ou dans le futur. Le [QuerySet](#) initial (`q1`) n'est pas affecté par le processus d'affinage.

Les objets QuerySet sont différés

Les [QuerySets](#) sont différés (« lazy ») ; la création d'un [QuerySet](#) ne génère aucune activité au niveau de la base de données. Vous pouvez empiler les filtres toute la journée, Django ne lance aucune requête tant que le [QuerySet](#) n'est pas *évalué*. Regardez cet exemple :

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

Malgré le fait que ceci ressemble à trois interrogations de la base de données, en réalité une seule interrogation est faite, à la dernière ligne (`print(q)`). En général, les résultats d'un [QuerySet](#) ne

sont récupérés de la base de données que lorsque vous les « demandez ». Lorsque cela arrive, le [QuerySet](#) est *évalué* en accédant à la base de données. Pour plus de détails sur le moment exact où l'évaluation a lieu, consultez [Quand les objets QuerySet sont évalués](#).

Sélection d'un objet unique avec `get()`

[filter\(\)](#) renvoie toujours un [QuerySet](#), même si la requête ne renvoie qu'un seul objet ; dans ce cas, le [QuerySet](#) ne contiendra simplement qu'un seul élément.

Si vous savez qu'un seul objet correspondra à votre requête, vous pouvez utiliser la méthode [get\(\)](#) d'un [Manager](#) qui renvoie directement l'objet :

```
>>> one_entry = Entry.objects.get(pk=1)
```

N'importe quelle expression de recherche peut être utilisée avec [get\(\)](#), comme pour [filter\(\)](#). Voir [Recherches dans les champs](#) ci-dessous.

Notez qu'il y a une différence entre l'utilisation de [get\(\)](#) et celle de [filter\(\)](#) avec le segment `[0]`. Si aucun résultat ne correspond à la requête, [get\(\)](#) génère une exception `DoesNotExist`. Cette exception est un attribut de la classe de modèle sur laquelle la requête est appliquée ; ainsi, dans le code ci-dessus, s'il n'existe pas d'objet `Entry` avec une clé primaire de 1, Django génère l'exception `Entry.DoesNotExist`.

De même, Django réagit si plus d'un élément correspond à la requête [get\(\)](#). Dans ce cas, il lève l'exception [MultipleObjectsReturned](#) qui est également un attribut de la classe de modèle elle-même.

Autres méthodes de `QuerySet`

La plupart du temps, vous utiliserez [all\(\)](#), [get\(\)](#), [filter\(\)](#) et [exclude\(\)](#) lorsque vous aurez besoin de sélectionner des objets dans la base de données. Cependant, c'est loin d'être tout ; consultez la [référence de l'API QuerySet](#) pour une liste complète des diverses méthodes de [QuerySet](#).

Limitation des `QuerySet`

Utilisez un sous-ensemble de la syntaxe Python de segmentation des listes pour restreindre un [QuerySet](#) à un certain nombre de résultats. C'est l'équivalent des clauses SQL `LIMIT` et `OFFSET`.

Par exemple, ceci renvoie les 5 premiers objets (`LIMIT 5`) :

```
>>> Entry.objects.all()[:5]
```

Ceci renvoie du sixième au dixième objet (`OFFSET 5 LIMIT 5`) :

```
>>> Entry.objects.all()[5:10]
```

Les index négatifs (ex. : `Entry.objects.all()[-1]`) ne sont pas pris en charge.

De manière générale, la segmentation d'un [QuerySet](#) renvoie un nouveau [QuerySet](#) ; la requête n'est pas évaluée, sauf si vous utilisez le paramètre `step` de la syntaxe de segmentation de Python. Par exemple, ceci exécuterait effectivement la requête pour renvoyer une liste d'un objet sur 2 parmi les 10 premiers :

```
>>> Entry.objects.all()[10:2]
```

Pour obtenir un *seul* objet plutôt qu'une liste (ex. : `SELECT foo FROM bar LIMIT 1`), utilisez un simple index au lieu d'un segment. Par exemple, ceci renvoie le premier objet `Entry` de la base de données, après avoir trié les objets alphabétiquement par titre (`headline`) :

```
>>> Entry.objects.order_by('headline')[0]
```

C'est grossièrement équivalent à :

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

Notez toutefois que si aucun objet ne correspond à la requête, une exception `IndexError` est générée dans le premier cas, tandis que dans le second cas, c'est l'exception `DoesNotExist` qui sera générée. Voir [get\(\)](#) pour plus de détails.

Recherches dans les champs

La recherche dans les champs est ce qui constitue le cœur des clauses SQL `WHERE`. La syntaxe s'exprime par des paramètres nommés dans les méthodes [filter\(\)](#), [exclude\(\)](#) et [get\(\)](#) de [QuerySet](#).

Les paramètres nommés de base de ces requêtes prennent la forme `champ__typerequete=valeur` (il s'agit d'un double soulignement). Par exemple :

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

peut être grossièrement traduit en code SQL comme :

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

Comment est-ce possible ?

Python has the ability to define functions that accept arbitrary name-value arguments whose names and values are evaluated at runtime. For more information, see [Keyword Arguments](#) in the official Python tutorial.

Le champ indiqué dans une recherche doit correspondre au nom d'un champ de modèle. Il existe cependant une exception ; dans le cas d'une clé étrangère [ForeignKey](#), vous pouvez indiquer le nom du champ additionné du suffixe `_id`. Dans ce cas, le paramètre valeur doit contenir la valeur brute de la clé primaire du modèle étranger. Par exemple :

```
>>> Entry.objects.filter(blog_id=4)
```

Si vous fournissez un paramètre nommé non valide, une fonction de recherche signalera une exception `TypeError`.

L'API de base de données gère une vingtaine de types de recherche ; une référence complète se trouve dans la [référence des recherches de champs](#). Pour vous donner une idée de ce qui est possible, voici quelques recherches parmi les plus fréquemment utilisées :

exact

Une correspondance « exacte ». Par exemple :

```
>>> Entry.objects.get(headline__exact="Cat bites dog")
```

Produit le code SQL ressemblant à :

```
SELECT ... WHERE headline = 'Cat bites dog';
```

Si vous n'indiquez pas de type de recherche, c'est-à-dire si votre paramètre nommé ne contient pas de double soulignement, Django considère que le type de recherche est `exact`.

Par exemple, les deux lignes suivantes sont équivalentes :

```
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14)        # __exact is implied
```

Il s'agit d'un raccourci commode, puisque les recherches `exact` sont les cas les plus courants.

iexact

Une recherche insensible à la casse. Ainsi, la requête :

```
>>> Blog.objects.get(name__iexact="beatles blog")
```

Aurait pour occurrence un Blog intitulé "Beatles Blog", "beatles blog" ou même "BeAtLEs bLoG".

contains

Test d'inclusion sensible à la casse. Par exemple :

```
Entry.objects.get(headline__contains='Lennon')
```

Pourrait être traduit grosso modo par ce code SQL :

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

Notez que le titre (headline) 'Today Lennon honored' correspondrait à cette recherche, mais pas 'today lennon honored'.

Il existe aussi une version insensible à la casse, [icontains](#).

[startswith](#), [endswith](#)

Recherche « commençant par » et « finissant par », respectivement. Il en existe aussi des versions insensibles à la casse : [istartswith](#) et [iendswith](#).

Encore une fois, ce n'est qu'un survol. Une référence complète est disponible dans la [référence des recherches de champ](#).

Recherches traversant les relations

Django offre une approche puissante et intuitive pour « suivre » les relations dans les recherches, se chargeant automatiquement des JOIN SQL en arrière-plan. Pour atteindre une relation, utilisez simplement les noms de champ qui servent de relation vers d'autres modèles, séparés par des doubles soulignements, jusqu'à atteindre le champ souhaité.

Cet exemple sélectionne tous les objets `Entry` d'un `Blog` ayant pour `name` la valeur `'Beatles Blog'`:

```
>>> Entry.objects.filter(blog__name='Beatles Blog')
```

Ce mécanisme de traversée de relations peut être aussi profond que vous le souhaitez.

Ceci fonctionne également dans le sens inverse. Pour faire référence à une relation « inverse », utilisez simplement le nom du modèle en minuscules.

Cet exemple sélectionne tous les objets `Blog` ayant au moins une `Entry` ayant un `headline` contenant `'Lennon'`:

```
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

Si vous filtrez à travers plusieurs relations et qu'un des modèles intermédiaires n'a pas de valeur répondant à la condition du filtre, Django le traite comme s'il y avait là un objet vide (toutes les valeurs sont NULL), mais valide. C'est-à-dire qu'aucune erreur ne sera signalée. Par exemple, dans ce filtre :

```
Blog.objects.filter(entry__authors__name='Lennon')
```

(partant du principe qu'il existe un modèle `Author` lié), s'il n'y a pas de `author` associé à une `entry`, ce sera traité comme s'il n'y avait pas non plus de `name`, plutôt que de signaler une erreur à cause de l'`author` manquant. Normalement, c'est bien le comportement souhaité. Le seul cas où ça pourrait porter à confusion, c'est quand vous utilisez [isnull](#). Ainsi :

```
Blog.objects.filter(entry__authors__name__isnull=True)
```

renvoie les objets `Blog` ayant un `author` avec le `name` vide ainsi que tous ceux ayant un `author` vide dans l'`entry`. Si vous ne voulez pas de cette deuxième catégorie d'objets, vous pourriez écrire :

```
Blog.objects.filter(entry__authors__isnull=False,  
entry__authors__name__isnull=True)
```

Traversée des relations multivaluées

Quand vous filtrez un objet basé sur un champ [ManyToManyField](#) ou un champ [ForeignKey](#) inversé, deux types de filtres peuvent être intéressants. Considérez la relation Blog/Entry (Blog vers Entry est une relation un-à-plusieurs). Nous pourrions être intéressés à trouver des blogues qui ont une entrée contenant “Lennon” dans le titre et ayant été publiée en 2008. Ou nous pourrions vouloir trouver des blogues qui contiennent une entrée avec “Lennon” dans le titre et une entrée qui a été publiée en 2008. Comme il y a plusieurs entrées associées à un seul Blog, ces deux requêtes sont possibles et ont du sens dans certaines situations.

Le même genre de situation survient avec un champ [ManyToManyField](#). Par exemple, si une Entry avait un [ManyToManyField](#) nommé tags, nous pourrions vouloir trouver des entrées liées aux tags nommés “music” et “bands”, ou nous pourrions vouloir une entrée qui contient un tag avec le nom “music” et un statut “public”.

Pour gérer ces deux situations, Django a une façon cohérente de traiter les appels de [filter\(\)](#). Tous les paramètres d’un appel à [filter\(\)](#) sont appliqués simultanément pour filtrer les éléments qui correspondent à tous ces critères. Les appels subséquents à [filter\(\)](#) restreignent ensuite davantage les éléments renvoyés, mais pour les relations multivaluées ces appels s’appliquent à tous les objets liés au modèle principal, pas nécessairement aux objets (dépendants du modèle principal) qui ont été sélectionnés par un appel précédent à [filter\(\)](#).

Ceci peut paraître un peu déroutant ; nous espérons qu’un exemple viendra clarifier les choses. Pour sélectionner tous les blogues qui contiennent des entrées qui ont à la fois “Lennon” dans le titre et qui ont été publiées en 2008 (la même entrée qui satisfait les deux conditions), nous pourrions écrire :

```
Blog.objects.filter(entry__headline__contains='Lennon', entry__pub_date__year=2008)
```

Pour sélectionner tous les blogues qui contiennent une entrée avec “Lennon” dans le titre **et d’autre part** une entrée qui a été publiée en 2008, nous pourrions écrire :

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(entry__pub_date__year=2008)
```

En supposant qu’un seul blogue contienne à la fois des articles contenant “Lennon” et des articles de 2008, mais qu’aucun des articles de 2008 ne contient “Lennon”, la première requête ne renverrait aucun blogue alors que la seconde requête renverrait ce blogue.

Dans le deuxième exemple, le premier filtre restreint le queryset à tous les blogues avec un lien vers des articles contenant “Lennon” dans le titre. Le second filtre restreint *davantage* les blogues résultants à ceux qui ont aussi un lien vers les articles publiés en 2008. Les articles sélectionnés par le second filtre peuvent être ou non les mêmes que ceux sélectionnés par le premier filtre. Nous filtrons les éléments Blog avec chaque appel de filtre, pas les éléments Entry.

Note

Le comportement de `filter()` pour les requêtes qui couvrent des relations multi-valeurs tel que décrit ci-dessus, n'est pas implémenté de la même manière pour `exclude()`. Au contraire, les conditions de restriction indiquées lors qu'un appel à `exclude()` ne doivent pas nécessairement s'appliquer au même élément.

Par exemple, la requête ci-dessous exclurait les blogues qui contiennent **à la fois** "*Lennon*" dans le titre et des entrées publiées en 2008

```
Blog.objects.exclude(
    entry__headline__contains='Lennon',
    entry__pub_date__year=2008,
)
```

Cependant, contrairement au comportement lors de l'utilisation de `filter()`, cela ne sélectionnera pas les blogs basés sur des entrées qui satisfont les deux conditions en même temps. Pour faire cela, c'est à dire pour sélectionner tous les blogs qui ne contiennent pas d'entrées publiées avec "*Lennon*" et qui ont été publiées en 2008, vous devez faire deux requêtes

```
Blog.objects.exclude(
    entry=Entry.objects.filter(
        headline__contains='Lennon',
        pub_date__year=2008,
    ),
)
```

Les filtres peuvent référencer les champs d'un modèle

Dans les exemples donnés jusqu'ici, nous avons construit des filtres comparant la valeur d'un champ de modèle avec une constante. Mais qu'en est-il si vous souhaitez comparer la valeur d'un champ de modèle avec un autre champ du même modèle ?

Django propose les expressions F pour autoriser de telles comparaisons. Les instances de `F()` agissent comme des références vers un champ de modèle à l'intérieur d'une requête. Ces références peuvent ensuite être utilisées dans des filtres de requêtes pour comparer les valeurs de deux champs différents d'une même instance de modèle.

Par exemple, pour obtenir la liste de tous les articles de blogue ayant reçus plus de commentaires que de « pings », nous construisons un objet `F()` pour référencer le nombre de pings afin d'utiliser cet objet `F()` dans la requête :

```
>>> from django.db.models import F
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks'))
```

Django prend en charge l'utilisation des opérateurs arithmétiques addition, soustraction, multiplication, division, modulo et puissance avec les objets `F()`, aussi bien avec des constantes qu'avec d'autres objets `F()`. Pour obtenir tous les articles de blogue avec *plus de deux fois* de commentaires que de pings, nous modifions ainsi la requête :

```
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks') * 2)
```


Pour trouver tous les articles où la notation de l'article est plus petite que la somme des pings et des commentaires, nous pourrions effectuer cette requête :

```
>>> Entry.objects.filter(rating__lt=F('n_comments') + F('n_pingbacks'))
```

Vous pouvez aussi employer la notation en double soulignement pour traverser les relations dans un objet `F()`. Un objet `F()` contenant un double soulignement produira toute jointure nécessaire pour accéder aux objets liés. Par exemple, pour obtenir tous les articles dont le nom de l'auteur est égal au nom du blogue, nous pourrions effectuer la requête :

```
>>> Entry.objects.filter(authors__name=F('blog__name'))
```

Pour les champs `date` et `date/heure`, il est possible d'ajouter ou de soustraire un objet [`timedelta`](#). L'exemple suivant renvoie tous les articles modifiés plus de 3 jours après avoir été publiés :

```
>>> from datetime import timedelta
>>> Entry.objects.filter(mod_date__gt=F('pub_date') + timedelta(days=3))
```

Les objets `F()` prennent aussi en charge les opérations bit à bit via `.bitand()` et `.bitor()`, par exemple :

```
>>> F('somefield').bitand(16)
```

Le raccourci de recherche pk

Pour des raisons pratiques, Django fournit un raccourci de recherche `pk`, signifiant « primary key » (clé primaire).

Dans le modèle d'exemple `Blog`, la clé primaire est le champ `id`. Ainsi, ces trois lignes sont équivalentes :

```
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
>>> Blog.objects.get(pk=14) # pk implies id__exact
```

L'emploi de `pk` n'est pas limité aux requêtes `__exact`; toute expression de requête peut être combinée avec `pk` pour effectuer une requête sur la clé primaire d'un modèle :

```
# Get blogs entries with id 1, 4 and 7
>>> Blog.objects.filter(pk__in=[1,4,7])
```

```
# Get all blog entries with id > 14
>>> Blog.objects.filter(pk__gt=14)
```

Les recherches sur `pk` fonctionnent également en traversant les relations. Par exemple, ces trois lignes sont équivalentes :

```
>>> Entry.objects.filter(blog__id__exact=3) # Explicit form
>>> Entry.objects.filter(blog__id=3)        # __exact is implied
>>> Entry.objects.filter(blog__pk=3)         # __pk implies __id__exact
```

Échappement des signes pour cent et des soulignements dans les instructions LIKE

Les recherches de champs produisant des instructions SQL LIKE (`iexact`, `contains`, `icontains`, `startswith`, `istartswith`, `endswith` et `iendswith`) échappent automatiquement les deux caractères à signification spéciale des instructions LIKE: le signe pour cent et le soulignement (dans une instruction LIKE, le signe pour cent est utilisé comme caractère de remplacement multiple alors qu'un soulignement est utilisé comme caractère de remplacement unique).

Cela signifie que les choses devraient fonctionner intuitivement, sans faille d'abstraction. Par exemple, pour obtenir la liste de tous les articles contenant un signe pour cent, il suffit d'indiquer le signe pour cent comme pour n'importe quel autre caractère :

```
>>> Entry.objects.filter(headline__contains='%')
```

Django se charge de l'échappement à votre place ; le code SQL résultant ressemblera à quelque chose comme :

```
SELECT ... WHERE headline LIKE '%\%%';
```

Même chose pour les soulignements. Aussi bien les signes pour cent que les soulignements sont gérés pour vous de manière transparente.

Mise en cache et objets QuerySet

Chaque [QuerySet](#) contient un cache pour minimiser l'accès à la base de données. La compréhension de son fonctionnement vous permet d'écrire le code le plus efficace possible.

Pour un nouvel objet [QuerySet](#), son cache est vide. À la première évaluation d'un [QuerySet](#) (et donc qu'une requête de base de données est effectuée), Django enregistre les résultats de la requête dans le cache de l'objet [QuerySet](#) et renvoie les résultats qui ont été explicitement demandés (par ex. l'élément suivant dans le cas d'une itération du [QuerySet](#)). Les évaluations suivantes de l'objet [QuerySet](#) réutilisent les résultats mis en cache.

Gardez à l'esprit ce comportement de mise en cache, car vous pourriez vous faire avoir si vous n'utilisez pas correctement l'objet [QuerySet](#). Par exemple, le code suivant crée deux objets [QuerySet](#), les évalue et les abandonne :

```
>>> print([e.headline for e in Entry.objects.all()])
>>> print([e.pub_date for e in Entry.objects.all()])
```

Dans ce cas, la même requête de base de données est exécutée deux fois, ce qui double en pratique la charge sur la base de données. Il est également imaginable que les deux listes ne contiennent pas exactement les mêmes enregistrements de base de données, car un objet `Entry` pourrait avoir été ajouté ou supprimé dans la fraction de seconde entre les deux requêtes.

Pour éviter ce problème, il suffit de stocker l'objet [QuerySet](#) et de le réutiliser :

```
>>> queryset = Entry.objects.all()
```

```
>>> print([p.headline for p in queryset]) # Evaluate the query set.
>>> print([p.pub_date for p in queryset]) # Re-use the cache from the evaluation.
```

Quand les objets QuerySet ne sont pas mis en cache

Les objets QuerySet ne mettent pas toujours leurs résultats en cache. Lorsque seule une *partie* d'un queryset est évaluée, le cache est consulté mais s'il n'est pas rempli, les éléments renvoyés par la requête suivante ne sont pas mis en cache. Spécifiquement, cela signifie que la [restriction d'un jeu de requêtes](#) en employant une segmentation ou un index de liste ne remplira pas le cache.

Par exemple, l'interrogation répétée d'un objet QuerySet à l'aide d'un index effectue une requête dans la base de données à chaque fois :

```
>>> queryset = Entry.objects.all()
>>> print queryset[5] # Queries the database
>>> print queryset[5] # Queries the database again
```

Cependant, si le jeu de requête complet a déjà été évalué, le cache sera tout de même mis à contribution :

```
>>> queryset = Entry.objects.all()
>>> [entry for entry in queryset] # Queries the database
>>> print queryset[5] # Uses cache
>>> print queryset[5] # Uses cache
```

Voici quelques exemples d'autres actions qui déclenchent l'évaluation de tout l'objet QuerySet et qui par conséquent remplissent le cache :

```
>>> [entry for entry in queryset]
>>> bool(queryset)
>>> entry in queryset
>>> list(queryset)
```

Note

L'affichage simple d'un QuerySet ne remplit pas le cache. Ceci parce que l'appel à `__repr__()` ne renvoie qu'un segment du jeu de requête complet.

Requêtes complexes avec des objets Q

Les requêtes à paramètres nommés (dans [filter\(\)](#), etc.) sont combinées par AND (ET). Si vous avez besoin d'exécuter des requêtes plus complexes (par exemple des requêtes contenant des instructions avec OR (OU)), vous pouvez utiliser des [objets Q](#).

Un [objet Q](#) (`django.db.models.Q`) est un objet utilisé pour englober plusieurs paramètres nommés. Ces paramètres nommés sont indiqués comme pour les « Recherches dans les champs » ci-dessus.

Par exemple, cet objet Q représente une seule requête LIKE:

```
from django.db.models import Q
Q(question__startswith='What')
```

Les objets Q peuvent être combinés à l'aide des opérateurs & et |. Lorsqu'un opérateur est utilisé avec deux objets Q, cela produit un nouvel objet Q.

Par exemple, cette ligne produit un seul objet Q représentant la combinaison par « OR » de deux requêtes "question__startswith":

```
Q(question__startswith='Who') | Q(question__startswith='What')
```

C'est équivalent à la clause SQL WHERE ci-dessous :

```
WHERE question LIKE 'Who%' OR question LIKE 'What%'
```

Vous pouvez composer des instructions de complexité arbitraire en combinant des objets Q avec les opérateurs & et | et en les groupant par des parenthèses. De même, les objets Q peuvent être inversés par l'opérateur de négation ~, permettant à des requêtes combinées d'utiliser à la fois des requêtes normales et des requêtes inversées (NOT) :

```
Q(question__startswith='Who') | ~Q(pub_date__year=2005)
```

Chaque fonction de recherche acceptant des paramètres nommés (par ex. [filter\(\)](#), [exclude\(\)](#), [get\(\)](#)) peut aussi recevoir un ou plusieurs objets Q comme paramètre positionnel (non nommé). Si vous indiquez plusieurs objets Q comme paramètres d'une fonction de recherche, les paramètres seront combinés avec « AND ». Par exemple :

```
Poll.objects.get(
    Q(question__startswith='Who'),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

... se traduit grossièrement en SQL par :

```
SELECT * from polls WHERE question LIKE 'Who%'
AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

Les fonctions de recherche peuvent mélanger l'utilisation d'objets Q et de paramètres nommés. Tous les paramètres fournis à une fonction de recherche (que ce soit des paramètres nommés ou des objets Q) sont combinés par l'opérateur « AND ». Cependant, si un objet Q est fourni, il doit précéder toute définition de paramètre nommé. Par exemple :

```
Poll.objects.get(
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
    question__startswith='Who',
)
```

... correspond à une requête valable, équivalente à l'exemple précédent ; mais :

```
# INVALID QUERY
```

```
Poll.objects.get(
    question__startswith='Who',
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

... n'est pas correct.

Voir aussi

Les [exemples de requêtes OR](#) dans les tests unitaires de Django montrent quelques utilisations possibles de Q.

Comparaison d'objets

Pour comparer deux instances de modèle, il suffit d'utiliser l'opérateur de comparaison standard de Python, le double signe égal : `==`. En arrière-plan, ce sont les valeurs clés primaires des deux modèles qui sont comparées.

En utilisant l'exemple `Entry` ci-dessus, les deux instructions suivantes sont équivalentes :

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

Si la clé primaire d'un modèle ne s'appelle pas `id`, aucun problème. Les comparaisons utilisent toujours la clé primaire, quel que soit son nom. Par exemple, si un champ clé primaire d'un modèle s'appelle `name`, ces deux lignes sont équivalentes :

```
>>> some_obj == other_obj
>>> some_obj.name == other_obj.name
```

Suppression d'objets

La méthode de suppression se nomme [`delete\(\)`](#). Cette méthode supprime immédiatement l'objet et renvoie le nombre d'objets supprimés ainsi qu'un dictionnaire avec le nombre de suppressions par type d'objet. Exemple :

```
>>> e.delete()
(1, {'weblog.Entry': 1})
```

Changed in Django 1.9:

La valeur renvoyée contenant le nombre d'objets supprimés a été ajoutée.

Il est aussi possible de supprimer des objets groupés. Chaque [`QuerySet`](#) comporte une méthode [`delete\(\)`](#) qui supprime tous les objets contenus dans l'objet [`QuerySet`](#).

Par exemple, cette commande supprime tous les objets `Entry` dont l'année de `pub_date` est 2005 :

```
>>> Entry.objects.filter(pub_date__year=2005).delete()
(5, {'webapp.Entry': 5})
```

Sachez toutefois que cette opération sera autant que possible exécutée purement au niveau SQL, ce qui signifie que les méthodes `delete()` des instances individuelles ne seront pas forcément appelées durant le processus. Si vous avez écrit une méthode `delete()` personnalisée dans une classe de modèle et que vous voulez être certain qu'elle soit appelée, vous devrez supprimer « manuellement » les instances de ce modèle (par ex. en itérant sur un objet [QuerySet](#) et en appelant explicitement `delete()` sur chaque instance) plutôt que d'employer la méthode de suppression groupée [delete\(\)](#) de l'objet [QuerySet](#).

Changed in Django 1.9:

La valeur renvoyée contenant le nombre d'objets supprimés a été ajoutée.

Lorsque Django supprime un objet, il émule par défaut le comportement de la contrainte SQL `ON DELETE CASCADE`. En d'autres termes, tout objet possédant des clés étrangères vers l'objet en cours de suppression seront également supprimés. Par exemple :

```
b = Blog.objects.get(pk=1)
# This will delete the Blog and all of its Entry objects.
b.delete()
```

Ce comportement en cascade peut être personnalisé au moyen du paramètre [on_delete](#) de la classe [ForeignKey](#).

Notez que [delete\(\)](#) est la seule méthode de [QuerySet](#) qui n'est pas exposée sur un objet [Manager](#). Il s'agit d'un mécanisme de sécurité pour vous empêcher d'exécuter accidentellement `Entry.objects.delete()` ce qui supprimerait *toutes* les lignes. Si vous souhaitez *vraiment* supprimer tous les objets, vous devez alors indiquer explicitement une requête contenant tous les objets :

```
Entry.objects.all().delete()
```

Copie des instances de modèles

Même s'il n'existe pas de méthode intégrée pour la copie d'instances de modèles, il est possible de créer facilement de nouvelles instances en copiant toutes les valeurs des champs d'une autre instance. Dans le cas le plus simple, il suffit de définir `pk` à `None`. En utilisant notre exemple de blogue :

```
blog = Blog(name='My blog', tagline='Bloggging is easy')
blog.save() # blog.pk == 1

blog.pk = None
blog.save() # blog.pk == 2
```

Les choses se compliquent lorsqu'il y a de l'héritage. Considérons une sous-classe de `Blog`:

```
class ThemeBlog(Blog):
    theme = models.CharField(max_length=200)
```

```
django_blog = ThemeBlog(name='Django', tagline='Django is easy', theme='python')
django_blog.save() # django_blog.pk == 3
```

En raison du fonctionnement de l'héritage, vous devez définir à None à la fois pk et id:

```
django_blog.pk = None
django_blog.id = None
django_blog.save() # django_blog.pk == 4
```

Ce procédé ne copie pas les objets liés. Si vous souhaitez copier les relations, il faudra écrire un peu plus de code. Dans notre exemple, Entry possède un champ plusieurs-à-plusieurs vers Author:

```
entry = Entry.objects.all()[0] # some previous entry
old_authors = entry.authors.all()
entry.pk = None
entry.save()
entry.authors = old_authors # saves new many2many relations
```

Mise à jour simultanée de plusieurs objets

Il peut arriver que vous souhaitiez définir la valeur d'un certain champ pour tous les objets d'un [QuerySet](#). C'est faisable à l'aide de la méthode [update\(\)](#). Par exemple :

```
# Update all the headlines with pub_date in 2007.
Entry.objects.filter(pub_date__year=2007).update(headline='Everything is the same')
```

Seuls les champs qui ne sont pas des relations et les champs [ForeignKey](#) peuvent être mis à jour avec cette méthode. Pour mettre à jour un champ qui ne représente pas une relation, indiquez la nouvelle valeur sous forme de constante. Pour mettre à jour un champ [ForeignKey](#), indiquez la nouvelle valeur sous forme d'une instance de modèle vers laquelle le champ devra pointer. Par exemple :

```
>>> b = Blog.objects.get(pk=1)

# Change every Entry so that it belongs to this Blog.
>>> Entry.objects.all().update(blog=b)
```

La méthode `update()` est appliquée immédiatement et renvoie le nombre de lignes correspondant à la requête (ce qui ne correspond pas toujours au nombre de lignes mises à jour si certaines lignes ont déjà la bonne valeur). La seule restriction sur l'objet [QuerySet](#) mis à jour est qu'il ne peut accéder qu'à une seule table de base de données, la table principale du modèle. Vous pouvez filtrer selon des champs liés, mais vous ne pouvez mettre à jour des colonnes que dans la table principale du modèle. Exemple :

```
>>> b = Blog.objects.get(pk=1)

# Update all the headlines belonging to this Blog.
>>> Entry.objects.select_related().filter(blog=b).update(headline='Everything is the same')
```

Soyez conscient que la méthode `update()` est directement convertie en instruction SQL. Il s'agit d'une opération groupée pour des mises à jour directes. Elle ne fait appel à aucune méthode [`save\(\)`](#) des modèles, n'émet aucun signal `pre_save` ou `post_save` (qui sont des conséquences de l'appel à [`save\(\)`](#)) et ne respecte pas l'option de champ [`auto_now`](#). Si vous souhaitez enregistrer chaque élément d'un objet [`QuerySet`](#) tout en garantissant que la méthode [`save\(\)`](#) de chaque instance est appelée, vous n'avez pas besoin d'une fonction particulière pour cela. Il suffit d'effectuer une boucle et d'appeler [`save\(\)`](#) pour chaque objet :

```
for item in my_queryset:
    item.save()
```

Les appels à `update` peuvent aussi utiliser les [`expressions F`](#) pour mettre à jour un champ en fonction de la valeur d'un autre champ du modèle. C'est particulièrement utile pour incrémenter des compteurs en fonction de leur valeur actuelle. Par exemple, pour incrémenter le nombre de pings de chaque article d'un blogue :

```
>>> Entry.objects.all().update(n_pingbacks=F('n_pingbacks') + 1)
```

Cependant, au contraire des objets `F()` dans les clauses `filter()` et `exclude()`, il n'est pas autorisé d'introduire des jointures lors de l'utilisation de `F()` dans des mises à jour ; seuls des champs du modèle en cours de mise à jour peuvent être référencés. Si vous essayez d'introduire une jointure avec un objet `F()`, une erreur `FieldError` sera générée :

```
# THIS WILL RAISE A FieldError
>>> Entry.objects.update(headline=F('blog__name'))
```

Objets liés

Lorsque vous définissez une relation dans un modèle (par ex. un champ [`ForeignKey`](#), [`OneToOneField`](#) ou [`ManyToManyField`](#)), les instances de ce modèle disposeront d'une API agréable pour accéder aux objets liés.

En utilisant les modèles définis au sommet de cette page, par exemple, un objet `Entry e` peut atteindre son objet `Blog` lié en accédant à l'attribut `blog`: `e.blog`.

(En arrière-plan, cette fonctionnalité est implémentée par des [`descripteurs`](#) Python. Cela n'a pas de conséquence particulière, mais nous le mentionnons pour les curieux).

Django crée également une API d'accession par l'autre côté de la relation, le lien depuis le modèle lié au modèle qui définit la relation. Par exemple, un objet `Blog b` peut accéder à une liste de tous les objets `Entry` liés par l'attribut `entry_set`: `b.entry_set.all()`.

Tous les exemples de cette section utilisent les modèles d'exemple `Blog`, `Author` et `Entry` définis au sommet de cette page.

Relations un-à-plusieurs

Sens « descendant »

Si un modèle possède un champ [ForeignKey](#), les instances de ce modèle ont accès à l'objet lié par un simple attribut du modèle.

Exemple :

```
>>> e = Entry.objects.get(id=2)
>>> e.blog # Returns the related Blog object.
```

Un attribut d'objet lié (clé étrangère) est accessible en lecture et en écriture. Comme l'on peut s'y attendre, les modifications de la clé étrangère ne sont pas enregistrées en base de données tant que [save\(\)](#) n'a pas été appelée. Exemple :

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

Si un champ [ForeignKey](#) possède `null=True` (c'est-à-dire qu'il autorise les valeurs NULL), vous pouvez lui attribuer `None` pour supprimer la relation. Exemple :

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = None
>>> e.save() # "UPDATE blog_entry SET blog_id = NULL ...;"
```

L'accès « descendant » aux relations un-à-plusieurs est mis en cache lors du premier accès à l'objet lié. Les accès suivants à la clé étrangère depuis la même instance d'objet se font à partir du cache.

Exemple :

```
>>> e = Entry.objects.get(id=2)
>>> print(e.blog) # Hits the database to retrieve the associated Blog.
>>> print(e.blog) # Doesn't hit the database; uses cached version.
```

Notez que la méthode [select_related\(\)](#) de [QuerySet](#) peuple de manière récursive et anticipée le cache de toutes les relations un-à-plusieurs. Exemple :

```
>>> e = Entry.objects.select_related().get(id=2)
>>> print(e.blog) # Doesn't hit the database; uses cached version.
>>> print(e.blog) # Doesn't hit the database; uses cached version.
```

Accès aux relations inverses

Si un modèle possède un champ [ForeignKey](#), les instances du modèle de clé étrangère auront accès à un [Manager](#) renvoyant toutes les instances du premier modèle. Par défaut, ce [Manager](#) se nomme `XXX_set`, où XXX est le nom du modèle source en minuscules. Ce [Manager](#) renvoie des objets [QuerySet](#), qui peuvent être filtrés et manipulés comme le décrit la section « Sélection d'objets » ci-dessus.

Exemple :

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all() # Returns all Entry objects related to Blog.

# b.entry_set is a Manager that returns QuerySets.
>>> b.entry_set.filter(headline__contains='Lennon')
>>> b.entry_set.count()
```

Il est possible de surcharger le nom XXX_set en définissant le paramètre [related_name](#) dans la définition du champ [ForeignKey](#). Par exemple, si le modèle Entry était modifié en définissant `blog = ForeignKey(Blog, on_delete=models.CASCADE, related_name='entries')`, l'exemple ci-dessus aurait été écrit comme ceci :

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all() # Returns all Entry objects related to Blog.

# b.entries is a Manager that returns QuerySets.
>>> b.entries.filter(headline__contains='Lennon')
>>> b.entries.count()
```

Utilisation d'un gestionnaire inverse personnalisé

Par défaut, le gestionnaire [RelatedManager](#) utilisé pour les relations inverses est une sous-classe du [gestionnaire par défaut](#) du modèle. Si vous aimeriez indiquer un autre gestionnaire pour une requête précise, vous pouvez utiliser la syntaxe suivante :

```
from django.db import models

class Entry(models.Model):
    #...
    objects = models.Manager() # Default Manager
    entries = EntryManager()   # Custom Manager

b = Blog.objects.get(id=1)
b.entry_set(manager='entries').all()
```

Si `EntryManager` effectuait du filtrage par défaut dans sa méthode `get_queryset()`, ce filtrage s'appliquerait à la méthode `all()`.

Naturellement, en indiquant un gestionnaire inverse personnalisé, cela permet aussi d'appeler ses méthodes personnalisées :

```
b.entry_set(manager='entries').is_published()
```

Méthodes supplémentaires pour manipuler des objets liés

En plus des méthodes [QuerySet](#) présentées ci-dessus dans « Sélection d'objets », l'objet [Manager](#) d'une [ForeignKey](#) possède des méthodes supplémentaires pour gérer l'ensemble des objets liés. Vous trouverez ci-dessous un résumé de ces méthodes et des détails plus complets peuvent être lus dans la [référence des objets liés](#).

```
add(obj1, obj2, ...)
```

Ajoute les objets modèles indiqués à l'ensemble des objets liés.

`create(**kwargs)`

Crée un nouvel objet, l'enregistre et le place dans l'ensemble des objets liés. Renvoie l'objet nouvellement créé.

`remove(obj1, obj2, ...)`

Enlève les objets modèles indiqués de l'ensemble des objets liés.

`clear()`

Enlève tous les objets de l'ensemble des objets liés.

`set(objs)`

Remplace l'ensemble des objets liés.

Pour définir les membres d'un ensemble d'objets liés d'un seul coup, il suffit d'attribuer des valeurs provenant d'un objet itérable. Celui-ci peut contenir des instances d'objets ou une simple liste de valeurs de clés primaires. Par exemple :

```
b = Blog.objects.get(id=1)
b.entry_set = [e1, e2]
```

Dans cet exemple, `e1` et `e2` peuvent être des instances normales de `Entry` ou des valeurs entières de clés primaires.

Si la méthode `clear()` est disponible, tout objet pré-existant sera enlevé de `entry_set` avant que tous les objets de la séquence itérable (dans ce cas, une liste) soient ajoutés à l'ensemble. Si la méthode `clear()` n'est pas disponible, tous les objets de la séquence itérable sont ajoutés sans que les éléments existants ne soient enlevés.

Toute opération « inverse » décrite dans cette section prend immédiatement effet au niveau de la base de données. Tout ajout, création ou suppression est immédiatement et automatiquement enregistré dans la base de données.

Relations plusieurs-à-plusieurs

Les deux côtés d'une relation plusieurs-à-plusieurs obtiennent automatiquement l'API d'accès à l'autre côté. L'API fonctionne de manière identique à la relation « inverse » d'une relation un-à-plusieurs décrite ci-dessus.

La seule différence tient dans le nommage de l'attribut : le modèle qui définit le champ [`ManyToManyField`](#) utilise le nom d'attribut du champ lui-même, alors que le modèle « inverse » utilise le nom de modèle en minuscules du modèle original en y ajoutant `'_set'` (tout comme pour les relations un-à-plusieurs inverses).

Un exemple va rendre tout ceci plus compréhensible :

```
e = Entry.objects.get(id=3)
e.authors.all() # Returns all Author objects for this Entry.
e.authors.count()
e.authors.filter(name__contains='John')

a = Author.objects.get(id=5)
a.entry_set.all() # Returns all Entry objects for this Author.
```

Tout comme pour [ForeignKey](#), il est possible d'indiquer une valeur [related_name](#) dans [ManyToManyField](#). Dans l'exemple ci-dessus, si le champ [ManyToManyField](#) de Entry était défini avec `related_name='entries'`, chaque instance de Author posséderait un attribut `entries` au lieu de `entry_set`.

Relations un-à-un

Les relations un-à-un sont très semblables aux relations plusieurs-à-un. Si vous définissez un champ [OneToOneField](#) dans votre modèle, les instances de ce modèle auront accès à l'objet lié via un simple attribut du modèle.

Par exemple :

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry, on_delete=models.CASCADE)
    details = models.TextField()

ed = EntryDetail.objects.get(id=2)
ed.entry # Returns the related Entry object.
```

La différence survient dans les requêtes « inverses ». Le modèle lié dans une relation un-à-un a aussi accès à un objet [Manager](#), mais celui-ci représente un objet unique et non pas une collection d'objets :

```
e = Entry.objects.get(id=2)
e.entrydetail # returns the related EntryDetail object
```

Si aucun objet n'a été attribué à cette relation, Django génère une exception `DoesNotExist`.

Il est possible d'attribuer des instances à la relation inverse sur le même principe que pour les relations descendantes :

```
e.entrydetail = ed
```

Comment la relation inverse est-elle possible ?

D'autres interfaces d'abstraction objet-relationnel exigent de définir les relations des deux côtés. Les développeurs Django pensent qu'il s'agit d'une violation du principe DRY (Don't Repeat Yourself, ne pas se répéter), c'est pourquoi Django ne demande de définir la relation que d'un seul côté.

Mais comment est-ce possible, dans la mesure où une classe de modèle ne sait pas quelles autres classes de modèles lui sont liées avant que ces autres classes soient elles-mêmes chargées ?

La réponse se trouve dans le [registre des applications](#). Lorsque Django démarre, il importe chaque application figurant dans [INSTALLED_APPS](#), puis le module `models` dans chaque application. Chaque fois qu'une nouvelle classe est créée, Django ajoute les relations inverses aux modèles liés. Si les modèles liés n'ont pas encore été importés, Django garde la trace de ces relations et les complète au moment où les modèles concernés sont finalement importés.

C'est pour cette raison qu'il est particulièrement important que tous les modèles utilisés soient définis dans des applications qui figurent dans [INSTALLED_APPS](#). Sinon, les relations inverses pourraient ne pas fonctionner correctement.

Les requêtes sur les objets liés

Les requêtes impliquant des objets liés suivent les mêmes règles que les requêtes contenant des valeurs de champs normales. Pour indiquer la valeur à laquelle doit correspondre une requête, vous pouvez utiliser soit une instance de l'objet lui-même, soit la valeur de clé primaire de l'objet.

Par exemple, si vous avez un objet `Blog b` avec `id=5`, les trois requêtes suivantes sont identiques :

```
Entry.objects.filter(blog=b) # Query using object instance
Entry.objects.filter(blog=b.id) # Query using id from instance
Entry.objects.filter(blog=5) # Query using id directly
```

Recours au code SQL brut

Si vous rencontrez le besoin de devoir écrire une requête SQL trop complexe pour l'abstraction de base de données de Django, il peut être nécessaire d'écrire le code SQL à la main. Django offre quelques options pour l'écriture de requêtes SQL brutes ; voir [Lancement de requêtes SQL brutes](#).

Finalement, il est important de noter que la couche de base de données de Django n'est qu'une interface vers votre base de données. Vous pouvez accéder à votre base de données par d'autres outils, langages de programmation ou environnements de base de données ; il n'y a rien de spécifique à Django en ce qui concerne la base de données.

[Modèles](#)
[Agrégation](#)

Agrégation

Le guide thématique sur l'[API d'abstraction de base de données de Django](#) décrit la façon dont les requêtes Django peuvent être utilisées pour créer, récupérer, mettre à jour et supprimer des objets individuels. Il est cependant parfois nécessaire de récupérer des valeurs dérivées du résumé ou de

l'*agrégation* d'une collection d'objets. Ce guide thématique décrit la manière dont des valeurs agrégées peuvent être générées et renvoyées en utilisant des requêtes Django.

Tout au long de ce guide, nous nous référerons aux modèles suivants. Ces modèles sont utilisés pour gérer l'inventaire d'une série de bibliothèques en ligne :

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

class Publisher(models.Model):
    name = models.CharField(max_length=300)
    num_awards = models.IntegerField()

class Book(models.Model):
    name = models.CharField(max_length=300)
    pages = models.IntegerField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    rating = models.FloatField()
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    pubdate = models.DateField()

class Store(models.Model):
    name = models.CharField(max_length=300)
    books = models.ManyToManyField(Book)
    registered_users = models.PositiveIntegerField()
```

Antisèche

Vous êtes pressé ? Voici comment effectuer des requêtes avec agrégation, sur la base des modèles ci-dessus :

```
# Total number of books.
>>> Book.objects.count()
2452

# Total number of books with publisher=BaloneyPress
>>> Book.objects.filter(publisher__name='BaloneyPress').count()
73

# Average price across all books.
>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg('price'))
{'price__avg': 34.35}

# Max price across all books.
>>> from django.db.models import Max
>>> Book.objects.all().aggregate(Max('price'))
{'price__max': Decimal('81.20')}

# Cost per page
>>> from django.db.models import F, FloatField, Sum
>>> Book.objects.all().aggregate(
```

```

...     price_per_page=Sum(F('price')/F('pages'), output_field=FloatField()))
{'price_per_page': 0.4470664529184653}

# All the following queries involve traversing the Book<->Publisher
# foreign key relationship backwards.

# Each publisher, each with a count of books as a "num_books" attribute.
>>> from django.db.models import Count
>>> pubs = Publisher.objects.annotate(num_books=Count('book'))
>>> pubs
[<Publisher BaloneyPress>, <Publisher SalamiPress>, ...]
>>> pubs[0].num_books
73

# The top 5 publishers, in order by number of books.
>>> pubs = Publisher.objects.annotate(num_books=Count('book')).order_by('-
num_books')[:5]
>>> pubs[0].num_books
1323

```

Génération de requêtes d'agrégation sur un objet QuerySet

Django propose deux manières de générer des agrégations. La première est de générer un résumé des valeurs d'un QuerySet entier. Par exemple, admettons que vous vouliez calculer le prix moyen de tous les livres disponibles à la vente. La syntaxe de requêtes de Django fournit une façon de décrire l'ensemble de tous les livres :

```
>>> Book.objects.all()
```

Ce dont nous avons besoin, c'est d'une manière de calculer un résumé des valeurs de tous les objets appartenant à ce QuerySet. Cela se fait en ajoutant une clause `aggregate()` à la suite de l'objet QuerySet:

```

>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg('price'))
{'price__avg': 34.35}

```

Dans cet exemple, `all()` est redondant, cela pourrait donc être simplifié en écrivant :

```

>>> Book.objects.aggregate(Avg('price'))
{'price__avg': 34.35}

```

Le paramètre de la clause `aggregate()` décrit la valeur d'agrégation que vous souhaitez calculer, dans ce cas la moyenne du champ `price` du modèle `Book`. Une liste des fonctions d'agrégation disponibles se trouve dans la [référence de QuerySet](#).

`aggregate()` est la clause terminale d'un objet QuerySet qui, lorsqu'il est évalué, renvoie un dictionnaire de paires nom-valeur. Le nom est un identifiant de la valeur agrégée ; la valeur est le résultat du calcul d'agrégation. Le nom est généré automatiquement à partir du nom du champ et de la fonction d'agrégation. Si vous voulez indiquer manuellement le nom de la valeur agrégée, vous pouvez le faire en précisant ce nom dans la clause d'agrégation :

```
>>> Book.objects.aggregate(average_price=Avg('price'))
{'average_price': 34.35}
```

Si vous voulez générer plus d'une valeur agrégée, il suffit d'ajouter un autre paramètre dans la clause `aggregate()`. Ainsi, si nous voulions aussi connaître les prix maximaux et minimaux de tous les livres, nous écririons la requête suivante :

```
>>> from django.db.models import Avg, Max, Min
>>> Book.objects.aggregate(Avg('price'), Max('price'), Min('price'))
{'price__avg': 34.35, 'price__max': Decimal('81.20'), 'price__min':
Decimal('12.99')}
```

Génération de valeurs agrégées pour chaque élément d'un QuerySet

La deuxième manière de générer des valeurs récapitulatives est de générer un récapitulatif individuel pour chaque objet d'un `QuerySet`. Par exemple, si vous récupérez une liste de livres, il peut être utile de connaître le nombre d'auteurs ayant contribué à chaque livre. Chaque objet `Book` possède une relation plusieurs-à-plusieurs vers le modèle `Author`; nous voulons récapituler cette relation pour chaque livre de la requête.

Les récapitulatifs par objet peuvent être générés par la clause `annotate()`. Lorsque celle-ci est mentionnée, chaque objet de la requête est annoté avec les valeurs indiquées.

La syntaxe de ces annotations est identique à celle utilisée pour les clauses `aggregate()`. Chaque paramètre d'`annotate()` décrit une valeur agrégée à calculer. Par exemple, pour annoter les livres avec le nombre de leurs auteurs :

```
# Build an annotated queryset
>>> from django.db.models import Count
>>> q = Book.objects.annotate(Count('authors'))
# Interrogate the first object in the queryset
>>> q[0]
<Book: The Definitive Guide to Django>
>>> q[0].authors__count
2
# Interrogate the second object in the queryset
>>> q[1]
<Book: Practical Django Projects>
>>> q[1].authors__count
1
```

Comme pour `aggregate()`, le nom de l'annotation est automatiquement dérivé du nom de la fonction d'agrégation et du nom du champ sur lequel se fait le calcul. Vous pouvez surcharger ce nom par défaut en fournissant un alias lors de la définition de l'annotation :

```
>>> q = Book.objects.annotate(num_authors=Count('authors'))
>>> q[0].num_authors
2
>>> q[1].num_authors
```


Au contraire de `aggregate()`, `annotate()` n'est *pas* une clause terminale. Le résultat de la clause `annotate()` est un objet `QuerySet`. Cet objet peut très bien être modifié par une autre opération de type `QuerySet`, comme par exemple `filter()`, `order_by()` ou même d'autres appels à `annotate()`.

Combinaison de plusieurs agrégations

La combinaison de plusieurs agrégations avec `annotate()` produit un mauvais résultat [<https://code.djangoproject.com/ticket/10060>](https://code.djangoproject.com/ticket/10060), car des jointures sont utilisées à la place de sous-requêtes :

```
>>> book = Book.objects.first()
>>> book.authors.count()
2
>>> book.store_set.count()
3
>>> q = Book.objects.annotate(Count('authors'), Count('store'))
>>> q[0].authors__count
6
>>> q[0].store__count
6
```

Pour la plupart des agrégations, il n'existe pas de solution à ce problème. Cependant, l'agrégation [`Count`](#) possède un paramètre `distinct` qui peut être utile :

```
>>> q = Book.objects.annotate(Count('authors', distinct=True), Count('store',
distinct=True))
>>> q[0].authors__count
2
>>> q[0].store__count
3
```

En cas de doute, inspectez la requête SQL !

Afin de comprendre ce qui se produit dans la requête, il est possible d'examiner la propriété `query` de l'objet `QuerySet`.

Jointures et agrégations

Jusqu'ici, nous avons eu affaire à des agrégations sur des champs appartenant au modèle sur lequel portait la requête. Cependant, la valeur à agréger peut parfois appartenir à un modèle qui est lié au modèle sur lequel porte la requête.

Lors de la définition du champ à agréger dans une fonction d'agrégation, Django permet d'utiliser la même [notation de double soulignement](#) qui est utilisée pour se référer aux champs liés dans les filtres. Django se charge ensuite des jointures de tables nécessaires pour récupérer et agréger la valeur liée.

Par exemple, pour trouver l'intervalle des prix pratiqués dans chaque magasin, vous pourriez utiliser l'annotation :

```
>>> from django.db.models import Max, Min
>>> Store.objects.annotate(min_price=Min('books__price'),
max_price=Max('books__price'))
```

Ceci indique à Django de récupérer le modèle `Store`, de faire la jointure avec le modèle `Book` (par la relation plusieurs-à-plusieurs) et d'agréger sur le champ du prix du modèle `Book` pour produire les valeurs du minimum et du maximum.

Les mêmes règles s'appliquent à la clause `aggregate()`. Si vous vouliez connaître le prix le plus bas et le prix le plus haut de tous les livres mis en vente dans n'importe quel magasin, vous pourriez utiliser la fonction d'agrégation :

```
>>> Store.objects.aggregate(min_price=Min('books__price'),
max_price=Max('books__price'))
```

Les chaînes de jointure peuvent être aussi profondes que nécessaire. Par exemple, pour extraire l'âge du plus jeune auteur de tous les livres mis en vente, vous pourriez effectuer la requête :

```
>>> Store.objects.aggregate(youngest_age=Min('books__authors__age'))
```

Suivi des relations inverses

D'une manière similaire aux [Recherches traversant les relations](#), les agrégations et les annotations sur les champs de modèles ou sur les modèles liés à celui sur lequel porte la requête peuvent contenir des relations « inverses » traversantes. Ici également, il est possible d'utiliser la syntaxe des noms en minuscules des modèles liés et des doubles soulignements.

Par exemple, il est possible d'extraire tous les éditeurs annotés avec leur décompte respectif de tous leurs livres publiés (notez la manière d'utiliser `'book'` pour définir la relation de clé étrangère inversée) :

```
>>> from django.db.models import Count, Min, Sum, Avg
>>> Publisher.objects.annotate(Count('book'))
```

(chaque `Publisher` du jeu de requête résultant possédera un attribut supplémentaire nommé `book__count`)

Une autre requête pourrait porter sur le plus ancien livre publié par chaque éditeur :

```
>>> Publisher.objects.aggregate(oldest_pubdate=Min('book__pubdate'))
```

Le dictionnaire résultant possédera une clé nommée `'oldest_pubdate'`. Si l'alias n'avait pas été précisé, cette clé se serait appelée `'book__pubdate__min'`, ce qui est plutôt long.

Cela ne s'applique pas seulement aux clés étrangères, mais fonctionne aussi avec les relations plusieurs-à-plusieurs. Par exemple, nous pouvons rechercher tous les auteurs annotés avec le nombre

total de pages de tous les livres pour lesquels ils ont été (co-)auteurs (notez la manière d'utiliser 'book' pour définir la relation plusieurs-à-plusieurs inversée de Author vers Book)

```
>>> Author.objects.annotate(total_pages=Sum('book__pages'))
```

Chaque Author du jeu de requête résultant possédera un attribut supplémentaire nommé total_pages. Si l'alias n'avait pas été précisé, cette clé se serait appelée 'book__pages__sum', ce qui est plutôt long.

Nous pourrions encore extraire la moyenne des notes de tous les livres écrits par les auteurs enregistrés :

```
>>> Author.objects.aggregate(average_rating=Avg('book__rating'))
```

(The resulting dictionary will have a key called 'average_rating'. If no such alias were specified, it would be the rather long 'book__rating__avg'.)

Agrégations et autres clauses de QuerySet

filter() et exclude()

Les agrégations peuvent aussi intervenir dans les filtres. Toute méthode filter() (ou exclude()) appliquée à des champs de modèle normaux fera l'effet de restreindre les objets concernés par l'agrégation.

Lorsqu'il est combiné à une clause annotate(), un filtre fait l'effet de restreindre les objets sur lesquels l'annotation est calculée. Par exemple, vous pouvez générer une liste annotée de tous les livres ayant leur titre commençant par « Django » à l'aide de la requête :

```
>>> from django.db.models import Count, Avg
>>> Book.objects.filter(name__startswith="Django").annotate(num_authors=Count('authors'))
```

Lorsqu'il est combiné à une clause aggregate(), un filtre fait l'effet de restreindre les objets sur lesquels l'agrégation est calculée. Par exemple, vous pouvez générer le prix moyen de tous les livres ayant leur titre commençant par « Django » à l'aide de la requête :

```
>>> Book.objects.filter(name__startswith="Django").aggregate(Avg('price'))
```

Filtrage sur les annotations

Les valeurs annotées peuvent également être filtrées. L'alias de l'annotation peut être utilisé dans les clauses filter() et exclude() de la même manière que pour n'importe quel autre champ de modèle.

Par exemple, pour générer une liste de livres ayant plus d'un auteur, vous pouvez écrire la requête :

```
>>> Book.objects.annotate(num_authors=Count('authors')).filter(num_authors__gt=1)
```

Cette requête génère un jeu de requête annoté, puis filtre le résultat sur la base de l'annotation.

Ordre des clauses `annotate()` et `filter()`

Lors de la rédaction de requête complexe impliquant à la fois des clauses `annotate()` et `filter()`, une attention particulière doit être donnée à l'ordre dans lequel les clauses sont appliquées à l'objet `QuerySet`.

Lorsqu'une clause `annotate()` est appliquée à une requête, l'annotation est calculée sur l'état de la requête au point exact où l'annotation est demandée. L'implication pratique de ceci est que `filter()` et `annotate()` ne sont pas des opérations commutatives.

Étant donné :

- Un « Publisher » A possède deux livres avec notations 4 et 5.
- Un « Publisher » B possède deux livres avec notations 1 et 4.
- Un « Publisher » A possède un livre avec notation 1.

Voici un exemple avec l'agrégation `Count`:

```
>>> a, b = Publisher.objects.annotate(num_books=Count('book',
distinct=True)).filter(book__rating__gt=3.0)
>>> a, a.num_books
(<Publisher: A>, 2)
>>> b, b.num_books
(<Publisher: B>, 2)

>>> a, b =
Publisher.objects.filter(book__rating__gt=3.0).annotate(num_books=Count('book'))
>>> a, a.num_books
(<Publisher: A>, 2)
>>> b, b.num_books
(<Publisher: B>, 1)
```

Les deux requêtes renvoient une liste d'objets « Publisher » ayant au moins un livre avec une note dépassant 3.0, ce qui fait que Publisher C est exclu.

Dans la première requête, l'annotation précède le filtre, ce qui fait que le filtre n'a aucun effet sur l'annotation. `distinct=True` est obligatoire pour éviter une [anomalie de requête](#).

La seconde requête compte le nombre de livres ayant une note dépassant 3.0 pour chaque Publisher. Le filtre précède l'annotation, ce qui fait que le filtre limite les objets pris en compte lors du calcul de l'annotation.

Voici un autre exemple avec l'agrégation `Avg`:

```
>>> a, b =
Publisher.objects.annotate(avg_rating=Avg('book__rating')).filter(book__rating__gt=
3.0)
```

```
>>> a, a.avg_rating
(<Publisher: A>, 4.5) # (5+4)/2
>>> b, b.avg_rating
(<Publisher: B>, 2.5) # (1+4)/2

>>> a, b =
Publisher.objects.filter(book__rating__gt=3.0).annotate(avg_rating=Avg('book__rating'))
>>> a, a.avg_rating
(<Publisher: A>, 4.5) # (5+4)/2
>>> b, b.avg_rating
(<Publisher: B>, 4.0) # 4/1 (book with rating 1 excluded)
```

La première requête calcule la note moyenne de tous les livres de chaque éditeur pour les éditeurs ayant au moins un livre avec une note dépassant 3.0. La seconde requête calcule la moyenne des notes des livres d'un éditeur pour tous les livres dont la note dépasse 3.0.

Il est difficile de deviner comment l'ORM va traduire des jeux de requêtes complexes en instructions SQL ; en cas de doute, inspectez le code SQL produit avec `str(queryset.query)` et écrivez de nombreux tests.

order_by()

Les annotations peuvent être utilisées comme base de tri. Lorsque vous définissez une clause `order_by()`, les agrégations que vous indiquez peuvent référencer n'importe quel alias défini dans le cadre d'une clause `annotate()` de la requête.

Par exemple, pour trier un jeu de requête de livres par le nombre d'auteurs ayant contribué au livre, vous pourriez écrire la requête suivante :

```
>>> Book.objects.annotate(num_authors=Count('authors')).order_by('num_authors')
```

values()

En principe, les annotations sont générées pour chaque objet ; un jeu de requête annoté renvoie un résultat par objet du jeu de requête original. Cependant, lorsqu'une clause `values()` est utilisée pour restreindre les colonnes renvoyées dans le jeu de requête, la méthode d'évaluation des annotations est légèrement différente. Au lieu de renvoyer un résultat annoté pour chaque résultat du jeu de requête original, les résultats d'origine sont groupés selon les combinaisons uniques des champs indiqués dans la clause `values()`. Puis l'annotation est fournie pour chaque groupe unique ; l'annotation est donc calculée sur tous les membres du groupe.

Par exemple, considérons une requête sur les auteurs cherchant à trouver la moyenne des notes des livres écrits par chaque auteur :

```
>>> Author.objects.annotate(average_rating=Avg('book__rating'))
```

Ceci renvoie un résultat pour chaque auteur de la base de données, annoté par la note moyenne de leurs livres.

Cependant, le résultat sera légèrement différent si vous utilisez une clause `values()`:

```
>>> Author.objects.values('name').annotate(average_rating=Avg('book__rating'))
```

Dans cet exemple, les auteurs sont groupés par nom, vous allez donc obtenir un résultat annoté pour chaque nom d'auteur *unique*. Cela signifie que si vous avez deux auteurs de même nom, leurs résultats seront fusionnés en un seul dans la requête résultante ; la moyenne sera calculée sur tous les livres écrits par les deux auteurs.

Ordre des clauses `annotate()` et `values()`

Comme pour la clause `filter()`, l'ordre d'apparition des clauses `annotate()` et `values()` dans une requête est important. Si la clause `values()` précède `annotate()`, l'annotation est calculée par rapport aux groupements définis par la clause `values()`.

Cependant, si la clause `annotate()` précède la clause `values()`, les annotations sont générées sur la totalité du jeu de requête. Dans ce cas, la clause `values()` ne restreint que les champs dans le résultat final.

Par exemple, si nous inversons l'ordre des clauses `values()` et `annotate()` dans notre exemple précédent :

```
>>> Author.objects.annotate(average_rating=Avg('book__rating')).values('name',  
'average_rating')
```

Cela produit maintenant un résultat unique pour chaque auteur ; cependant, seuls le nom de l'auteur et l'annotation `average_rating` sont renvoyés dans les données de résultat.

Remarquez également que `average_rating` a été explicitement inclus dans la liste des valeurs à renvoyer. C'est obligatoire à cause de l'ordre des clauses `values()` et `annotate()`.

Si la clause `values()` précède la clause `annotate()`, toute annotation sera automatiquement ajoutée au résultat de la requête. Toutefois, si la clause `values()` est appliquée après la clause `annotate()`, vous devez inclure explicitement la colonne agrégée.

Interaction avec le tri par défaut ou `order_by()`

Les champs mentionnés dans la partie `order_by()` d'une requête (ou ceux faisant partie du tri par défaut d'un modèle) sont utilisés lors de la sélection des données du résultat, même s'ils ne sont pas spécifiés explicitement dans l'appel à `values()`. Ces champs supplémentaires sont aussi pris en compte pour regrouper les résultats similaires et peuvent faire apparaître de manière distincte des lignes qui seraient regroupées sans la présence de ces champs. C'est particulièrement flagrant lorsqu'il s'agit de compter des objets.

Pour fournir un exemple, supposons que vous ayez un modèle comme celui-ci :

```
from django.db import models
```

```
class Item(models.Model):
    name = models.CharField(max_length=10)
    data = models.IntegerField()

    class Meta:
        ordering = ["name"]
```

L'élément important est ici le tri par défaut sur le champ `name`. Si vous souhaitez compter le nombre d'apparitions distinctes de la valeur `data`, vous pourriez écrire ceci :

```
# Warning: not quite correct!
Item.objects.values("data").annotate(Count("id"))
```

...ce qui regroupe les objets `Item` par leurs valeurs `data` communes, puis compte le nombre de valeurs `id` dans chaque groupe. Sauf que ça ne marchera pas comme prévu. Le tri par défaut selon `name` intervient aussi dans le regroupement, ce qui fait que la requête regroupe sur les paires distinctes (`data`, `name`), ce qui ne correspond pas à l'intention de départ. Voici en réalité ce qu'il faut écrire pour obtenir la requête espérée :

```
Item.objects.values("data").annotate(Count("id")).order_by()
```

...effaçant ainsi tout ordre de tri dans la requête. Il serait aussi possible de trier sur le champ `data` sans conséquence néfaste, dans la mesure où ce champ est déjà impliqué dans la requête.

Ce comportement est le même que ce qui est noté dans la documentation des requêtes concernant [`distinct\(\)`](#), la règle générale étant identique : il n'est normalement pas souhaité que des colonnes supplémentaires soient incluses dans le résultat, il faut donc effacer l'ordre de tri ou en tout cas s'assurer que les champs de tri sont déjà impliqués dans l'appel à `values()`.

Note

Vous pourriez raisonnablement vous demander pourquoi Django ne supprime pas ces colonnes parasites à notre place. La raison principale est de conserver la cohérence avec `distinct()` et d'autres endroits : Django ne supprime **jamais** les contraintes de tri que vous avez indiquées (et nous ne pouvons pas modifier le comportement de ces autres méthodes car cela trahirait notre politique de [Stabilité de l'API](#)).

Agrégation des annotations

Vous pouvez aussi générer une agrégation des résultats d'une annotation. Lorsque vous définissez une clause `aggregate()`, cette agrégation peut se référer à n'importe quel alias défini dans une clause `annotate()` de la même requête.

Par exemple, si vous souhaitiez calculer le nombre moyen d'auteurs par livre, vous annoteriez premièrement les livres avec le nombre d'auteurs, puis vous agrégeriez ce nombre en vous référant au champ annoté :

```
>>> from django.db.models import Count, Avg
```

```
>>>
Book.objects.annotate(num_authors=Count('authors')).aggregate(Avg('num_authors'))
{'num_authors__avg': 1.66}
```

[Création de requêtes](#)
[Gestionnaires](#)

Gestionnaires

`class Manager` [\[source\]](#)

Un gestionnaire (objet `Manager`) est l'interface par laquelle les opérations de requêtes de base de données sont mises à disposition des modèles Django. Il existe au moins un `Manager` pour chaque modèle d'une application Django.

Le fonctionnement des classes `Manager` est documenté dans [Création de requêtes](#); ce document aborde spécifiquement les options de modèles qui personnalisent le comportement des gestionnaires.

Noms des gestionnaires

Par défaut, Django ajoute un gestionnaire nommé `objects` à chaque classe de modèle Django. Cependant, si vous aimeriez utiliser `objects` comme nom de champ ou que vous aimeriez nommer le gestionnaire autrement que `objects`, vous pouvez le renommer au niveau du modèle. Pour renommer le gestionnaire d'une classe donnée, définissez un attribut de classe de type `models.Manager()` dans le modèle. Par exemple :

```
from django.db import models

class Person(models.Model):
    #...
    people = models.Manager()
```

En utilisant cet exemple de modèle, `Person.objects` générera une exception `AttributeError`, mais `Person.people.all()` fournira effectivement la liste de tous les objets `Person`.

Gestionnaires personnalisés

Vous pouvez utiliser un gestionnaire personnalisé dans un modèle particulier en étendant la classe `Manager` de base et en créant votre propre instance de `Manager` dans votre modèle.

Il y a deux raisons de vouloir personnaliser un gestionnaire : pour lui ajouter des méthodes supplémentaires ou pour modifier l'objet `QuerySet` initial que le gestionnaire renvoie.

Ajout de méthodes de gestionnaire supplémentaires

L'ajout de méthodes de gestionnaire supplémentaires est la façon privilégiée d'ajouter des fonctionnalités au « niveau table » à des modèles (pour les fonctionnalités au « niveau ligne », c'est-à-dire les fonctions qui agissent sur une seule instance d'un objet de modèle, utilisez les [méthodes de modèles](#) et non pas des méthodes de gestionnaire personnalisées).

Une méthode de gestionnaire personnalisée peut renvoyer tout ce qu'on veut, elle n'est pas tenue de renvoyer un objet `QuerySet`.

Par exemple, cette méthode de gestionnaire personnalisée offre une méthode `with_counts()` qui renvoie une liste de tous les objets `OpinionPoll`, chacun recevant un attribut supplémentaire `num_responses` qui est le résultat d'une requête d'agrégation :

```
from django.db import models

class PollManager(models.Manager):
    def with_counts(self):
        from django.db import connection
        cursor = connection.cursor()
        cursor.execute("""
            SELECT p.id, p.question, p.poll_date, COUNT(*)
            FROM polls_opinionpoll p, polls_response r
            WHERE p.id = r.poll_id
            GROUP BY p.id, p.question, p.poll_date
            ORDER BY p.poll_date DESC""")
        result_list = []
        for row in cursor.fetchall():
            p = self.model(id=row[0], question=row[1], poll_date=row[2])
            p.num_responses = row[3]
            result_list.append(p)
        return result_list

class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    poll_date = models.DateField()
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(OpinionPoll, on_delete=models.CASCADE)
    person_name = models.CharField(max_length=50)
    response = models.TextField()
```

Avec cet exemple, vous écririez `OpinionPoll.objects.with_counts()` pour obtenir la liste des objets `OpinionPoll` comportant l'attribut `num_responses`.

Une autre chose à relever au sujet de cet exemple est que les méthodes de gestionnaire ont accès à `self.model` pour obtenir la classe de modèle à laquelle elles sont liées.

Modification des `QuerySet` initiaux des gestionnaires

Le `QuerySet` de base d'un gestionnaire renvoie tous les objets du système. Par exemple, en utilisant ce modèle :

```
from django.db import models
```

```
class Book(models.Model):  
    title = models.CharField(max_length=100)  
    author = models.CharField(max_length=50)
```

...l'instruction `Book.objects.all()` renvoie tous les livres de la base de données.

Vous pouvez surcharger le `QuerySet` de base d'un gestionnaire en surchargeant la méthode `Manager.get_queryset()`. `get_queryset()` doit renvoyer un objet `QuerySet` doté des propriétés nécessaires.

Par exemple, le modèle suivant possède *deux* gestionnaires, un qui renvoie tous les objets et un autre qui ne renvoie que les livres de Roald Dahl :

```
# First, define the Manager subclass.  
class DahlBookManager(models.Manager):  
    def get_queryset(self):  
        return super(DahlBookManager, self).get_queryset().filter(author='Roald  
Dahl')
```

```
# Then hook it into the Book model explicitly.
```

```
class Book(models.Model):  
    title = models.CharField(max_length=100)  
    author = models.CharField(max_length=50)
```

```
    objects = models.Manager() # The default manager.
```

```
    dahl_objects = DahlBookManager() # The Dahl-specific manager.
```

Avec cet exemple de modèle, `Book.objects.all()` renvoie tous les livres de la base de données, mais `Book.dahl_objects.all()` ne retourne que ceux qui ont été écrits par Roald Dahl.

Naturellement, comme `get_queryset()` renvoie un objet `QuerySet`, vous pouvez appliquer `filter()`, `exclude()` et toutes les autres méthodes de `QuerySet`. Ainsi, ces instructions sont toutes valables :

```
Book.dahl_objects.all()  
Book.dahl_objects.filter(title='Matilda')  
Book.dahl_objects.count()
```

Cet exemple a aussi mis en évidence une autre technique intéressante : l'emploi de plusieurs gestionnaires dans un même modèle. Vous pouvez lier autant d'instances de gestionnaires que vous voulez à un modèle. C'est une manière simple de définir des « filtres » fréquemment utilisés dans des modèles.

Par exemple :

```
class AuthorManager(models.Manager):  
    def get_queryset(self):  
        return super(AuthorManager, self).get_queryset().filter(role='A')
```

```
class EditorManager(models.Manager):  
    def get_queryset(self):
```

```

        return super(EditorManager, self).get_queryset().filter(role='E')

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices= (('A', _('Author')), ('E',
_(_('Editor'))))
    people = models.Manager()
    authors = AuthorManager()
    editors = EditorManager()

```

Cet exemple permet d'effectuer les requêtes `Person.authors.all()`, `Person.editors.all()` et `Person.people.all()` en sachant à quoi vous attendre au niveau des résultats.

Gestionnaires par défaut

Si vous utilisez des objets `Manager` personnalisés, sachez que le premier `Manager` que Django rencontre (dans l'ordre où ils ont été définis dans le modèle) reçoit un statut spécial. Django interprète le premier gestionnaire défini dans une classe comme le gestionnaire par défaut, et plusieurs parties de Django (y compris [dumpdata](#)) utilisent exclusivement ce gestionnaire pour le modèle en question. En conséquence, il est conseillé de choisir avec prudence le gestionnaire par défaut afin d'éviter une situation où la surcharge de `get_queryset()` aboutit à l'incapacité de récupérer des objets avec lesquels vous avez besoin de travailler.

Utilisation de gestionnaires pour accéder aux objets liés

Par défaut, Django utilise une instance de classe de gestionnaire « normale » pour accéder aux objets liés (par ex. `choice.poll`), et non pas le gestionnaire par défaut de l'objet lié. Ceci parce que Django doit absolument être capable de récupérer l'objet lié, même dans les cas où il serait filtré (et donc inaccessible) par son gestionnaire par défaut.

Si la classe de gestionnaire normale ([django.db.models.Manager](#)) ne convient pas dans un cas particulier, vous pouvez forcer Django à utiliser la même classe que le gestionnaire par défaut du modèle et définissant l'attribut `use_for_related_fields` dans la classe du gestionnaire. Vous en trouverez la documentation complète [ci-dessous](#).

Appel personnalisé de méthodes QuerySet depuis le gestionnaire

Alors que la plupart des méthodes d'un `QuerySet` standard sont directement accessibles à partir d'un `Manager`, ce n'est le cas pour les méthodes supplémentaires définies sur un `QuerySet` personnalisé que si vous les implémentez également sur le `Manager`:

```

class PersonQuerySet(models.QuerySet):
    def authors(self):
        return self.filter(role='A')

    def editors(self):
        return self.filter(role='E')

```

```

class PersonManager(models.Manager):
    def get_queryset(self):
        return PersonQuerySet(self.model, using=self._db)

    def authors(self):
        return self.get_queryset().authors()

    def editors(self):
        return self.get_queryset().editors()

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices= (('A', _('Author')), ('E',
_(_('Editor')))))
    people = PersonManager()

```

Cet exemple permet d'appeler `authors()` et `editors()` directement depuis le gestionnaire `Person.people`.

Création d'objet Manager avec des méthodes de QuerySet

En lieu et place de l'approche ci-dessus qui nécessite de dupliquer les méthodes à la fois sur les objets `QuerySet` et `Manager`, [`QuerySet.as_manager\(\)`](#) peut être utilisé pour créer une instance de `Manager` avec une copie des méthodes d'un objet `QuerySet` personnalisé :

```

class Person(models.Model):
    ...
    people = PersonQuerySet.as_manager()

```

L'instance de `Manager` créée par [`QuerySet.as_manager\(\)`](#) sera pratiquement identique au `PersonManager` de l'exemple précédent.

Toutes les méthodes `QuerySet` n'ont pas forcément de sens au niveau `Manager`; par exemple, nous empêchons volontairement la méthode [`QuerySet.delete\(\)`](#) d'être copiée vers la classe `Manager`.

Les méthodes sont copiées selon les règles suivantes :

- Les méthodes publiques sont copiées par défaut.
- Les méthodes privées (commençant par un soulignement) ne sont pas copiées par défaut.
- Les méthodes avec un attribut `queryset_only` à `False` sont toujours copiées.
- Les méthodes avec un attribut `queryset_only` à `True` ne sont jamais copiées.

Par exemple :

```

class CustomQuerySet(models.QuerySet):
    # Available on both Manager and QuerySet.
    def public_method(self):

```

```

        return

# Available only on QuerySet.
def _private_method(self):
    return

# Available only on QuerySet.
def _opted_out_public_method(self):
    return
    opted_out_public_method.queryset_only = True

# Available on both Manager and QuerySet.
def _opted_in_private_method(self):
    return
    _opted_in_private_method.queryset_only = False

```

from_queryset()

classmethod from_queryset(queryset_class)

Pour une utilisation plus avancée, il peut être souhaitable d’avoir à la fois un Manager personnalisé et un QuerySet personnalisé. Vous pouvez le faire en appelant `Manager.from_queryset()` qui renvoie une *sous-classe* du Manager de base avec une copie des méthodes QuerySet personnalisées :

```

class BaseManager(models.Manager):
    def manager_only_method(self):
        return

class CustomQuerySet(models.QuerySet):
    def manager_and_queryset_method(self):
        return

class MyModel(models.Model):
    objects = BaseManager.from_queryset(CustomQuerySet)()

```

Vous pouvez également stocker la classe générée dans une variable :

```

CustomManager = BaseManager.from_queryset(CustomQuerySet)

class MyModel(models.Model):
    objects = CustomManager()

```

Gestionnaires personnalisés et héritage de modèle

L’héritage de classes et les gestionnaires de modèle ne font pas très bon ménage. Les gestionnaires sont souvent spécifiques aux classes dans lesquelles ils sont définis et leur héritage dans des sous-classes n’est pas forcément toujours une bonne idée. Et comme le premier gestionnaire défini devient le *gestionnaire par défaut*, il est important de pouvoir contrôler cet ordre de définition. Voici donc comment Django se comporte face aux gestionnaires personnalisés et à [l’héritage de modèles](#):

1. Les gestionnaires définis dans les classes de base non abstraites ne sont *pas* hérités par les classes enfants. Si vous souhaitez réutiliser un gestionnaire d’une base non abstraite, il faut le

redéclarer explicitement dans la classe enfant. Ce type de gestionnaire est généralement très spécifique à la classe dans laquelle il est défini, c'est pourquoi leur héritage pourrait produire des résultats inattendus (particulièrement lorsqu'il s'agit du gestionnaire par défaut). Ils ne sont donc pas transmis à leurs classes enfants.

2. Les gestionnaires des classes de base abstraites sont toujours hérités par les classes enfants, par le moyen Python habituel de l'ordre de résolution de nom (les noms des classes enfants surchargent ceux de leurs parents). Les classes de base abstraites sont conçues pour centraliser les informations et les comportements communs à leurs classes enfants. La définition de gestionnaires communs est un aspect entrant tout à fait dans ces informations communes.
3. Le gestionnaire par défaut d'une classe est soit le premier gestionnaire déclaré dans la classe, s'il y en a un, ou le gestionnaire par défaut de la première classe de base abstraite dans la hiérarchie d'héritage, le cas échéant. Si aucun gestionnaire par défaut n'est explicitement déclaré, c'est le gestionnaire par défaut habituel de Django qui est utilisé.

Ces règles assurent la souplesse nécessaire permettant d'installer un ensemble de gestionnaires personnalisés sur un groupe de modèles via une classe de base abstraite, tout en personnalisant aussi le gestionnaire par défaut. Prenons l'exemple de cette classe de base :

```
class AbstractBase(models.Model):
    # ...
    objects = CustomManager()

    class Meta:
        abstract = True
```

Si vous l'utilisez directement dans une sous-classe, `objects` sera le gestionnaire par défaut si aucun gestionnaire n'est déclaré dans la classe de base :

```
class ChildA(AbstractBase):
    # ...
    # This class has CustomManager as the default manager.
    pass
```

Si vous voulez hériter de `AbstractBase`, mais avec un autre gestionnaire par défaut, vous pouvez définir le gestionnaire par défaut dans la classe enfant :

```
class ChildB(AbstractBase):
    # ...
    # An explicit default manager.
    default_manager = OtherManager()
```

Ici, `default_manager` est le gestionnaire par défaut. Le gestionnaire `objects` est toujours disponible, puisqu'il est hérité. Il n'est juste plus le gestionnaire par défaut.

Pour terminer avec cet exemple, supposons que vous vouliez ajouter des gestionnaires supplémentaires dans la classe enfant tout en conservant le gestionnaire par défaut de `AbstractBase`. Vous ne pouvez pas ajouter directement le nouveau gestionnaire dans la classe enfant, car cela surchargerait le

gestionnaire par défaut et vous devriez aussi inclure explicitement tous les gestionnaires de la classe de base abstraite. La solution est de placer les gestionnaires supplémentaires dans une autre classe de base et introduire celle-ci dans la hiérarchie d'héritage *après* ceux par défaut :

```
class ExtraManager(models.Model):
    extra_manager = OtherManager()

    class Meta:
        abstract = True

class ChildC(AbstractBase, ExtraManager):
    # ...
    # Default manager is CustomManager, but OtherManager is
    # also available via the "extra_manager" attribute.
    pass
```

Notez que même si vous pouvez *définir* un gestionnaire personnalisé dans le modèle abstrait, vous ne pouvez *appeler* aucune de ses méthodes au travers du modèle abstrait. C'est-à-dire :

```
ClassA.objects.do_something()
```

est légal, mais :

```
AbstractBase.objects.do_something()
```

générera une exception. C'est parce que les gestionnaires sont censés intégrer la logique de gestion d'ensembles d'objets. Comme il n'est pas possible d'avoir un ensemble d'objets abstraits, leur gestion n'a pas de sens. Si vous gérez des fonctionnalités s'appliquant au modèle abstrait, vous devriez les placer dans une méthode `staticmethod` ou `classmethod` du modèle abstrait.

Détails d'implémentation

Quelles que soient les fonctionnalités ajoutées à un gestionnaire personnalisé, il doit toujours être possible de faire une copie légère (« shallow ») d'une de ses instances ; c'est-à-dire que le code suivant doit fonctionner :

```
>>> import copy
>>> manager = MyManager()
>>> my_copy = copy.copy(manager)
```

Django effectue des copies légères des objets de gestionnaire durant certaines requêtes ; si votre gestionnaire ne peut pas être copié, ces requêtes échoueront.

Ce ne sera pas un problème pour la plupart des gestionnaires personnalisés. Si vous ne faites qu'ajouter des méthodes simples à votre classe `Manager`, il est improbable que vous empêchiez la copie de votre gestionnaire sans le vouloir. Cependant, si vous surchargez `__getattr__` ou d'autres méthodes privées de l'objet `Manager` qui contrôlent l'état de l'objet, il faut vous assurer que vous n'altériez pas la capacité de votre gestionnaire d'être copié.

Contrôle du type des gestionnaires automatiques

Ce document a déjà mentionné quelques endroits où Django crée automatiquement une classe de gestionnaire : [gestionnaires par défaut](#) et le gestionnaire « de base » utilisé pour [accéder aux objets liés](#). Il y a d'autres endroits dans l'implémentation de Django où des gestionnaires de base temporaires sont nécessaires. Ces gestionnaires automatiquement créés sont normalement des instances de la classe [django.db.models.Manager](#).

Tout au long de cette section, nous utiliserons le terme « gestionnaire automatique » pour désigner les gestionnaires que Django crée pour vous, soit comme un gestionnaire par défaut d'un modèle sans gestionnaire ou pour un usage temporaire lors de l'accès à des objets liés.

Cette classe par défaut n'est parfois pas le bon choix. Le gestionnaire par défaut peut ne pas disposer de toutes les méthodes nécessaires à la manipulation des données. Une classe de gestionnaire personnalisée selon vos besoins permet de créer des objets `QuerySet` personnalisés pour obtenir les informations dont vous avez besoin.

Django propose aux développeurs de gestionnaires personnalisés une manière d'indiquer que leur classe de gestionnaire doit être utilisée pour les gestionnaires automatiques chaque fois qu'ils deviennent le gestionnaire par défaut d'un modèle. Ceci se fait en définissant l'attribut `use_for_related_fields` dans la classe du gestionnaire :

```
class MyManager(models.Manager):
    use_for_related_fields = True
    # ...
```

Si cet attribut est défini dans le gestionnaire *par défaut* d'un modèle (seul le gestionnaire par défaut est considéré dans ces situations), Django utilise cette classe chaque fois qu'il a besoin de créer automatiquement un gestionnaire pour la classe. Sinon, il utilise [django.db.models.Manager](#).

Note historique

Étant donné le but pour lequel il est utilisé, le nom de cet attribut (`use_for_related_fields`) peut sembler un peu bizarre. À l'origine, cet attribut ne contrôlait que le type de gestionnaire utilisé pour l'accès aux champs liés, d'où son nom. Ce nom n'a ensuite pas été modifié, même quand il paraissait de plus en plus clair que le concept était plus largement utile. Ceci principalement pour [préserver le fonctionnement du code existant](#) dans les versions futures de Django.

Écriture correcte de gestionnaires pour les instances automatiques

La fonctionnalité `use_for_related_fields` est principalement axée sur les gestionnaires qui ont besoin de renvoyer une sous-classe de `QuerySet` personnalisée. Lorsque vous utilisez cette fonctionnalité dans un gestionnaire, il ne fait pas oublier certaines choses.

Ne jamais filtrer les résultats dans ce type de sous-classe de gestionnaire

L'une des raisons pour lesquelles un gestionnaire automatique est utilisé est le besoin d'accéder aux objets liés à partir d'un autre modèle. Dans ces situations, Django doit être capable de voir tous les objets du modèle qu'il récupère, afin que *tout* ce qui est référencé puisse être récupéré.

Si vous surchargez la méthode `get_queryset()` et que vous excluez certaines lignes, Django renverra des résultats incorrects. Ne faites pas cela. Un gestionnaire excluant des résultats dans `get_queryset()` n'est pas approprié comme gestionnaire automatique.

Définir `use_for_related_fields` lors de la création de la classe

L'attribut `use_for_related_fields` doit être défini dans la *classe* du gestionnaire, et non pas dans une *instance* de la classe. L'exemple précédent montre la bonne manière de le faire, tandis que le code suivant ne fonctionnera pas :

```
# BAD: Incorrect code
class MyManager(models.Manager):
    # ...
    pass

# Sets the attribute on an instance of MyManager. Django will
# ignore this setting.
mgr = MyManager()
mgr.use_for_related_fields = True

class MyModel(models.Model):
    # ...
    objects = mgr

# End of incorrect code.
```

De plus, vous ne devriez pas modifier l'attribut de l'objet classe après qu'il a été utilisé dans un modèle, car la valeur de l'attribut est prise en compte au moment de la création de la classe de modèle et n'est plus relue plus tard. Définissez l'attribut de la classe de gestionnaire lors de sa définition initiale comme dans le premier exemple de cette section et tout devrait fonctionner à merveille.

[Agrégation](#)

[Lancement de requêtes SQL brutes](#)

Lancement de requêtes SQL brutes

Lorsque l'[API d'interrogation des modèles](#) atteint ses limites, il est possible de recourir à l'écriture de code SQL brut. Django propose deux manières d'exécuter des requêtes SQL brutes : vous pouvez utiliser `Manager.raw()` pour [exécuter des requêtes brutes et renvoyer des instances de modèles](#) ou

vous pouvez outrepasser complètement la couche des modèles et [exécuter directement du code SQL personnalisé](#).

Avertissement

Vous devez être très prudent lors de l'écriture d'instructions SQL brutes. Lors de chaque utilisation, vous devez échapper correctement tout paramètre pouvant être contrôlé par les utilisateurs en employant `params` afin de vous protéger contre les attaques par injection SQL. Lisez attentivement les paragraphes sur la [protection contre les injections SQL](#).

Lancement de requêtes brutes

La méthode de gestionnaire `raw()` peut être utilisée pour exécuter des requêtes SQL brutes qui renvoient des instances de modèles :

```
Manager.raw(raw_query, params=None, translations=None)
```

Cette méthode accepte une requête SQL brute, l'exécute et renvoie une instance `django.db.models.query.RawQuerySet`. Il est possible alors d'effectuer une boucle sur cette instance `RawQuerySet` tout comme pour un objet [QuerySet](#) normal afin d'accéder aux instances d'objets.

Un exemple vaut mieux que mille mots. Supposons que vous ayez créé le modèle suivant :

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

Vous pouvez alors exécuter du code SQL personnalisé comme ceci :

```
>>> for p in Person.objects.raw('SELECT * FROM myapp_person'):
...     print(p)
John Smith
Jane Jones
```

Naturellement, cet exemple n'est pas des plus passionnants, car il correspond exactement à l'expression `Person.objects.all()`. Toutefois, `raw()` comporte quelques autres options qui en font un outil très puissant.

Noms de table des modèles

D'où vient le nom de la table ``Person`` dans cet exemple?

Par défaut, Django compose un nom de table de base de données en combinant l'« étiquette d'application » du modèle (le nom utilisé dans `manage.py startapp`) avec le nom de classe du modèle, séparés par un soulignement. Dans l'exemple, nous sommes partis du principe que le modèle `Person` se trouvait dans une application nommée `myapp`, et donc que le nom de la table était `myapp_person`.

Pour plus de détails, consultez la documentation de l'option [db_table](#) qui vous permet également de définir manuellement le nom de table de la base de données.

Avertissement

Aucune vérification n'est effectuée pour les instructions SQL transmises à `.raw()`. Django s'attend à ce que la requête renvoie un ensemble de lignes de la base de données, mais ne fait rien pour le vérifier a priori. Si la requête ne retourne pas de ligne, une erreur (potentiellement cryptique) sera générée.

Avertissement

Si vous effectuez des requêtes vers MySQL, notez que le forçage de type silencieux de MySQL peut produire des résultats inattendus lors du mélange de types. Si une requête porte sur une colonne de type chaîne mais contient une valeur nombre entier, MySQL transforme le type de toutes les valeurs de la table en nombre entier avant d'effectuer la comparaison. Par exemple, si la table contient les valeurs 'abc', 'def' et que la requête contient `WHERE ma_colonne=0`, les deux lignes seront sélectionnées. Pour empêcher cela, effectuez les transformations de type avant d'utiliser une valeur dans une requête.

Avertissement

Alors qu'il est possible d'itérer sur une instance `RawQuerySet` comme pour un objet [QuerySet](#) normal, `RawQuerySet` n'implémente pas toutes les méthodes qu'il est possible d'utiliser avec un `QuerySet`. Par exemple, `__bool__()` et `__len__()` ne sont pas définis pour `RawQuerySet`, ce qui fait que toutes les instances `RawQuerySet` sont considérées comme `True`. La raison pour laquelle ces méthodes ne sont pas implémentées par `RawQuerySet` est que leur implémentation sans cache interne serait ruineuse en terme de performances et que l'ajout de cache briserait la rétrocompatibilité.

Correspondance entre champs de requête et champs de modèle

`raw()` fait automatiquement correspondre les champs de la requête avec les champs du modèle.

L'ordre des champs dans la requête n'est pas important. En d'autres termes, les deux requêtes suivantes donneront le même résultat :

```
>>> Person.objects.raw('SELECT id, first_name, last_name, birth_date FROM
myapp_person')
...
>>> Person.objects.raw('SELECT last_name, birth_date, first_name, id FROM
myapp_person')
...
```

La correspondance se fait sur le nom. Cela signifie que vous pouvez utiliser les clauses SQL `AS` pour faire correspondre les champs de la requête aux champs du modèle. Ainsi, si vous disposez d'une autre table contenant les données de `Person`, vous pouvez facilement faire correspondre ces données avec des instances de `Person`:

```
>>> Person.objects.raw('''SELECT first AS first_name,
...                          last AS last_name,
...                          bd AS birth_date,
...                          pk AS id,
...                          FROM some_other_table''')
```

Tant que les noms correspondent, les instances de modèle seront créées correctement.

Il est aussi possible de faire correspondre les champs de requête aux champs de modèle en utilisant le paramètre `translations` de `raw()`. Il s'agit d'un dictionnaire faisant correspondre les noms des champs de la requête aux noms des champs du modèle. Par exemple, la requête ci-dessus aurait aussi pu être écrite de cette manière :

```
>>> name_map = {'first': 'first_name', 'last': 'last_name', 'bd': 'birth_date',
...             'pk': 'id'}
>>> Person.objects.raw('SELECT * FROM some_other_table', translations=name_map)
```

Filtrage par index

`raw()` autorise l'utilisation d'index ; dans le cas où vous souhaitez obtenir uniquement le premier résultat, vous pouvez écrire :

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_person')[0]
```

Cependant, l'indexation et la segmentation ne sont pas effectuées au niveau de la base de données. Si la base de données contient une grande quantité d'objets `Person`, il est plus efficace de limiter la requête au niveau SQL :

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_person LIMIT 1')[0]
```

Report des champs de modèle

Il est aussi possible d'ignorer certains champs :

```
>>> people = Person.objects.raw('SELECT id, first_name FROM myapp_person')
```

Les objets `Person` renvoyés par cette requête constitueront des instances de modèle différées (voir [defer\(\)](#)). Cela signifie que les champs omis dans la requête seront chargés à la demande. Par exemple :

```
>>> for p in Person.objects.raw('SELECT id, first_name FROM myapp_person'):
...     print(p.first_name, # This will be retrieved by the original query
...           p.last_name) # This will be retrieved on demand
...
John Smith
Jane Jones
```

En apparence, il semble que la requête ait récupéré à la fois le prénom et le nom. Cependant, cet exemple effectue en réalité 3 requêtes. Seuls les prénoms (`first_name`) ont été obtenus par la requête `raw()`, les noms (`last_name`) ont été obtenus chacun à la demande au moment où ils ont été affichés.

Un seul champ ne peut pas être omis, c'est le champ clé primaire. Django utilise la clé primaire pour identifier les instances de modèle, elle doit donc être obligatoirement incluse dans la requête brute. Une exception `InvalidQuery` est générée si vous oubliez d'inclure la clé primaire.

Ajout d'annotations

Vous pouvez aussi exécuter des requêtes contenant des champs qui ne sont pas définis dans le modèle. Par exemple, il serait possible d'utiliser la [fonction age\(\) de PostgreSQL](#) pour obtenir une liste de personnes avec leur âge calculé par la base de données :

```
>>> people = Person.objects.raw('SELECT *, age(birth_date) AS age FROM
myapp_person')
>>> for p in people:
...     print("%s is %s." % (p.first_name, p.age))
John is 37.
Jane is 42.
...
```

Transmission de paramètres dans raw()

S'il est nécessaire d'effectuer des requêtes paramétrées, il est possible d'utiliser le paramètre `params` de `raw()`:

```
>>> lname = 'Doe'
>>> Person.objects.raw('SELECT * FROM myapp_person WHERE last_name = %s', [lname])
```

`params` est une liste ou un dictionnaire de paramètres. Dans la chaîne de requête, il faut alors inclure des substituts `%s` pour une liste ou des substituts `%(clé)s` pour un dictionnaire (où clé est remplacé par une clé de dictionnaire, bien sûr), quel que soit le moteur de base de données. Ces substituts seront remplacés par le contenu du paramètre params.`

Note

Les paramètres de type dictionnaire ne sont pas pris en charge par le moteur SQLite ; avec ce moteur, vous devez transmettre les paramètres sous forme de liste.

Avertissement

N'utilisez pas de formatage de chaîne dans les requêtes brutes !

Il est tentant d'écrire la requête ci-dessus comme ceci :

```
>>> query = 'SELECT * FROM myapp_person WHERE last_name = %s' % lname
>>> Person.objects.raw(query)
```

Ne le faites pas.

En utilisant le paramètre `params`, vous êtes complètement protégé contre les [attaques d'injection SQL](#), une faille courante où un attaquant injecte du code SQL arbitraire dans votre base de données. Si

vous utilisez l'interpolation de chaîne, vous serez tôt ou tard victime d'injection SQL. Tant que vous n'oubliez pas de toujours utiliser le paramètre `params`, vous serez protégé.

Exécution directe de code SQL

Dans certains cas, même `Manager.raw()` ne suffit pas : il se peut que des requêtes doivent être effectuées sans correspondre proprement à des modèles ou que vous vouliez exécuter directement des requêtes `UPDATE`, `INSERT` ou `DELETE`.

Dans ces situations, vous pouvez toujours accéder directement à la base de données, outrepassant complètement la couche des modèles.

L'objet `django.db.connection` représente la connexion à la base de données par défaut. Pour utiliser la connexion à la base de données, appelez `connection.cursor()` pour obtenir un objet curseur. Puis, appelez `cursor.execute(sql, [params])` pour exécuter le code SQL et `cursor.fetchone()` ou `cursor.fetchall()` pour obtenir les lignes de résultat.

Par exemple :

```
from django.db import connection

def my_custom_sql(self):
    cursor = connection.cursor()

    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])

    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
    row = cursor.fetchone()

    return row
```

Notez que si vous voulez inclure des signes « pour cent » littéraux dans la requête, vous devez les doubler dans le cas où vous transmettez des paramètres :

```
cursor.execute("SELECT foo FROM bar WHERE baz = '30%'")
cursor.execute("SELECT foo FROM bar WHERE baz = '30%%' AND id = %s", [self.id])
```

Si vous utilisez [plus d'une base de données](#), vous pouvez utiliser `django.db.connections` pour obtenir la connexion (et le curseur) pour une base de données spécifique.

`django.db.connections` est un objet de type dictionnaire permettant de récupérer une connexion spécifique en employant son alias :

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
# Your code here...
```

Par défaut, l'API de base de données de Python renvoie les résultats sans les noms de champs, ce qui signifie que vous vous retrouvez avec une liste de valeurs plutôt qu'un dictionnaire. Pour un faible coût

en performances et en mémoire, vous pouvez obtenir les résultats sous forme de dictionnaire en écrivant quelque chose comme :

```
def dictfetchall(cursor):
    "Return all rows from a cursor as a dict"
    columns = [col[0] for col in cursor.description]
    return [
        dict(zip(columns, row))
        for row in cursor.fetchall()
    ]
```

Une autre option est d'utiliser une structure [collections.namedtuple\(\)](#) de la bibliothèque Python standard. Un `namedtuple` est un objet de type tuple dont les champs sont accessibles sous forme d'attribut ; l'accès par indice est aussi possible et l'objet est itérable. Les résultats sont immuables et accessibles par nom de champ ou par indice, ce qui pourrait être pratique :

```
from collections import namedtuple

def namedtuplefetchall(cursor):
    "Return all rows from a cursor as a namedtuple"
    desc = cursor.description
    nt_result = namedtuple('Result', [col[0] for col in desc])
    return [nt_result(*row) for row in cursor.fetchall()]
```

Voici un exemple de la différence entre les trois :

```
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> cursor.fetchall()
((54360982, None), (54360880, None))

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982}, {'parent_id': None, 'id': 54360880}]

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> results = namedtuplefetchall(cursor)
>>> results
[Result(id=54360982, parent_id=None), Result(id=54360880, parent_id=None)]
>>> results[0].id
54360982
>>> results[0][0]
54360982
```

Connexions et curseurs

`connection` et `cursor` implémentent essentiellement l'API de base de données standard de Python décrite dans la [PEP 249](#), à l'exception de ce qui concerne la [gestion des transactions](#).

Si cette API DB de Python ne vous est pas familière, notez que l'instruction SQL dans `cursor.execute()` utilise des substituants, "%s", plutôt que d'ajouter directement les paramètres dans la chaîne SQL. Si vous utilisez cette technique, la bibliothèque sous-jacente de base de données s'occupe automatiquement d'échapper vos paramètres au besoin.

Notez également que Django compte sur des substituts "%S", *pas* de substituts "?" qui sont utilisés par la bibliothèque SQLite de Python, pour des raisons de cohérence et de bon sens.

L'utilisation d'un curseur en tant que gestionnaire de contexte :

```
with connection.cursor() as c:  
    c.execute(...)
```

est équivalent à :

```
c = connection.cursor()  
try:  
    c.execute(...)  
finally:  
    c.close()
```

[Gestionnaires](#)

[Transactions de base de données](#)

Transactions de base de données

Django donne quelques moyens de contrôler la gestion des transactions de base de données.

Gestion des transactions de base de données

Comportement de transaction par défaut de Django

Le comportement par défaut de Django est de fonctionner en mode de validation automatique (« autocommit »). Chaque requête est immédiatement validée dans la base de données, sauf si une transaction est en cours. [Voir ci-dessous pour plus de détails](#).

Django utilise automatiquement des transactions ou des points de sauvegarde pour garantir l'intégrité des opérations de l'ORM qui nécessitent plusieurs requêtes, particulièrement les requêtes [delete\(\)](#) et [update\(\)](#).

La classe [TestCase](#) de Django englobe aussi chaque test dans une transaction pour des raisons de performance.

Couplage des transactions aux requêtes HTTP

Une façon fréquente de gérer les transactions sur le Web est d'envelopper chaque requête dans une transaction. Définissez [ATOMIC_REQUESTS](#) à True dans la configuration de chaque base de données pour laquelle vous souhaitez activer ce comportement.

Voici comment cela fonctionne. Avant d'appeler une fonction de vue, Django démarre une transaction. Si la réponse est renvoyée sans problème particulier, Django valide la transaction. Si la vue génère une exception, Django annule la transaction.

Il est possible d'effectuer des sous-transactions à l'aide de points de sauvegarde dans le code de la vue, typiquement avec le gestionnaire de contexte [atomic\(\)](#). Cependant, quand la vue se termine, soit tous les changements sont validés, soit tous sont annulés.

Avertissement

Bien que la simplicité de ce modèle de transaction est attrayant, il devient inefficace lorsque le trafic augmente. L'ouverture d'une transaction pour chaque vue présente un certain coût. L'impact sur la performance dépend de la manière dont vos applications font usage des requêtes et de la manière dont la base de données gère les verrous.

Transactions par requête et réponses en flux

Lorsqu'une vue renvoie une réponse en flux ([StreamingHttpResponse](#)), la lecture du contenu de la réponse exécute fréquemment du code pour générer le contenu. Comme la vue s'est déjà terminée, ce code s'exécute en dehors de la transaction.

De manière générale, il n'est pas conseillé d'écrire dans la base de données durant la génération d'une réponse en flux, car il n'y a plus de méthode raisonnable pour gérer les erreurs après le début de l'envoi de la réponse.

En pratique, cette fonctionnalité ne fait qu'envelopper chaque fonction de vue dans le décorateur [atomic\(\)](#) décrit ci-dessous.

Notez que seule l'exécution de la vue est incluse dans la transaction. Les intergiciels s'exécutent en dehors de la transaction, de même que le rendu des réponses par gabarit.

Lorsque [ATOMIC_REQUESTS](#) est actif, il est toujours possible d'empêcher les vues de s'exécuter dans une transaction.

`non_atomic_requests(using=None)`[\[source\]](#)

Ce décorateur annule l'effet de [ATOMIC_REQUESTS](#) pour une vue donnée :

```
from django.db import transaction

@transaction.non_atomic_requests
def my_view(request):
    do_stuff()

@transaction.non_atomic_requests(using='other')
def my_other_view(request):
    do_stuff_on_the_other_database()
```

Cela ne fonctionne que lorsque le décorateur est appliqué à la vue elle-même.

Contrôle explicite des transactions

Django fournit une API unifiée pour contrôler les transactions de base de données.

`atomic(using=None, savepoint=True)`[\[source\]](#)

L'atomicité est la propriété de base des transactions de base de données. `atomic` permet de créer un bloc de code à l'intérieur duquel l'atomicité est garantie au niveau de la base de données. Si le bloc de code se termine avec succès, les modifications sont validées dans la base de données. Si une exception apparaît, les modifications sont annulées en bloc.

Les blocs `atomic` peuvent être imbriqués. Dans ce cas, lorsqu'un bloc intérieur se termine avec succès, ses effets peuvent encore être annulés si une exception est générée plus loin dans le bloc englobant.

`atomic` peut être utilisé comme décorateur:

```
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    # This code executes inside a transaction.
    do_stuff()
```

et comme gestionnaire de contexte:

```
from django.db import transaction

def viewfunc(request):
    # This code executes in autocommit mode (Django's default).
    do_stuff()

    with transaction.atomic():
        # This code executes inside a transaction.
        do_more_stuff()
```

L'insertion d'`atomic` dans un bloc `try/except` est une manière naturelle de gérer les erreurs d'intégrité :

```
from django.db import IntegrityError, transaction

@transaction.atomic
def viewfunc(request):
    create_parent()

    try:
        with transaction.atomic():
            generate_relationships()
    except IntegrityError:
        handle_exception()

    add_children()
```

Dans cet exemple, même si `generate_relationships()` provoque une erreur de base de données en cassant une contrainte d'intégrité, vous pouvez exécuter des requêtes dans `add_children()` et les modifications de `create_parent()` sont toujours présentes. Notez que toute opération exécutée dans `generate_relationships()` aura déjà été annulée proprement lorsque `handle_exception()` est appelée, ce qui fait que le gestionnaire d'exception peut très bien agir au niveau de la base de données si nécessaire.

Évitez d'intercepter des exceptions à l'intérieur d'`atomic`!

À la sortie d'un bloc `atomic`, Django examine si la sortie se fait normalement ou par une exception afin de déterminer s'il doit valider ou annuler la transaction. Si vous interceptez et gérez des exceptions à l'intérieur du bloc `atomic`, vous cachez à Django le fait qu'un problème est survenu. Cela peut aboutir à des comportements inattendus.

Ce problème concerne plus spécifiquement les exceptions [DatabaseError](#) et ses sous-classes telles que [IntegrityError](#). Après une telle erreur, la transaction est cassée et Django procédera à son annulation dès la sortie du bloc `atomic`. Si vous essayez d'exécuter des requêtes en base de données avant que l'annulation intervienne, Django générera une exception [TransactionManagementError](#). Vous pouvez également rencontrer ce comportement lorsqu'un gestionnaire de signal lié à l'ORM génère une exception.

La manière correcte d'intercepter les erreurs de base de données est de le faire autour du bloc `atomic` comme dans l'exemple ci-dessus. Si nécessaire, ajoutez un bloc `atomic` supplémentaire à cet effet. Cette stratégie présente un autre avantage : elle délimite explicitement les opérations qui seront annulées si une exception se produit.

Si vous interceptez les exceptions générées par des requêtes SQL brutes, le comportement de Django n'est pas défini et dépend de la base de données.

Afin de garantir l'atomicité, `atomic` désactive certaines API. Si vous tentez de valider une transaction, de l'annuler ou de modifier l'état de validation automatique de la connexion de base de données à l'intérieur d'un bloc `atomic`, vous obtiendrez une exception.

`atomic` accepte un paramètre `using` devant correspondre au nom d'une base de données. Si ce paramètre n'est pas présent, Django utilise la base de données "default".

En arrière-plan, le code de gestion des transactions de Django :

- ouvre une transaction lorsqu'il entre dans le premier bloc `atomic`;
- crée un point de sauvegarde lorsqu'il entre dans un bloc `atomic` imbriqué ;
- libère le point de sauvegarde ou annule la transaction jusqu'au point de sauvegarde en quittant le bloc imbriqué ;
- valide ou annule la transaction en quittant le bloc de départ.

Vous pouvez désactiver la création de points de sauvegarde pour les blocs imbriqués en définissant le paramètre `savepoint` à `False`. Si une exception survient, Django procède à l'annulation de la transaction au moment de quitter le premier bloc ayant un point de sauvegarde, le cas échéant, ou le bloc initial sinon. L'atomicité est toujours garantie par la transaction du bloc initial. Cette option ne devrait être utilisée que si la création des points de sauvegarde affecte les performances de manière évidente. Le désavantage est que cela casse la gestion d'erreurs telle que décrite précédemment.

Il est possible d'utiliser `atomic` lorsque la validation automatique est désactivée. Seuls des points de sauvegarde seront utilisés, même pour le bloc initial.

Changed in Django 1.8.5:

Précédemment, le bloc atomique initial ne pouvait pas être déclaré avec `savepoint=False` lorsque la validation automatique était désactivée.

Considérations sur la performance

Les transactions ouvertes constituent une pénalité de performance pour votre serveur de base de données. Pour minimiser cet impact, gardez vos transactions aussi brèves que possible. C'est particulièrement important si vous utilisez [`atomic\(\)`](#) dans des processus de longue durée, en dehors du cycle requête/réponse de Django.

Autocommit

Pourquoi Django utilise-t-il l'autocommit

Dans le standard SQL, chaque requête SQL démarre une transaction, sauf s'il y en a déjà une en cours. De telles transactions doivent ensuite être explicitement soit validées (`commit`), soit annulées (`rollback`).

Ce n'est pas toujours très pratique pour les développeurs d'applications. Pour contourner ce problème, la plupart des bases de données mettent à disposition un mode « autocommit ». Lorsque ce mode est actif et qu'il n'y a pas de transaction ouverte, chaque requête SQL est englobée dans sa propre transaction. En d'autres termes, chacune de ces requêtes non seulement démarre une transaction, mais cette transaction est aussi automatiquement validée ou annulée en fonction du résultat de la requête.

[PEP 249](#), la spécification d'API de base de données Python v2.0, exige que le mode autocommit soit initialement désactivé. Django surcharge ce comportement par défaut et active le mode autocommit.

Pour empêcher cela, vous pouvez [désactiver la gestion des transactions](#), mais ce n'est pas recommandé.

Désactivation de la gestion des transaction

Vous pouvez désactiver totalement la gestion des transactions de Django pour une base de données précise en définissant [`AUTOCOMMIT`](#) à `False` dans sa configuration. Si vous faites cela, Django

n'activera pas le mode autocommit et n'effectue aucune opération de commit. Vous obtenez alors le comportement habituel de la bibliothèque de base de données sous-jacente.

Cela demande que vous effectuiez un commit explicite de chaque transaction, même de celles initiées par Django ou par des bibliothèques tierces. Ainsi, cela convient mieux à des situations où vous souhaitez mettre en place votre propre intergiciel de gestion des transactions ou que vous faites des choses plutôt étranges.

Lancement d'actions après le commit

New in Django 1.9.

Il peut arriver que vous ayez besoin d'effectuer une action liée à la transaction de base de données en cours, mais seulement si la transaction se termine par un commit réussi. On peut citer comme exemple une tâche [Celery](#), une notification par courriel ou une invalidation de cache.

Django fournit la fonction [on_commit\(\)](#) pour inscrire des fonctions de rappel qui seront exécutées après le commit réussi de la transaction :

```
on_commit(func, using=None)\[source\]
```

Passez une fonction (qui n'accepte pas de paramètre) à [on_commit\(\)](#):

```
from django.db import transaction

def do_something():
    pass # send a mail, invalidate a cache, fire off a Celery task, etc.

transaction.on_commit(do_something)
```

Il est aussi possible d'envelopper la fonction dans une lambda :

```
transaction.on_commit(lambda: some_celery_task.delay('arg1'))
```

La fonction transmise sera appelée immédiatement après une écriture potentielle de base de données pour laquelle `on_commit()` a été appelée et se terminant par un commit réussi.

Si vous appelez `on_commit()` alors qu'aucune transaction n'est en cours, la fonction transmise est immédiatement exécutée.

Si cette écriture de base de données potentielle se trouve annulée (rollback), typiquement lorsqu'une exception non traitée est générée dans le bloc [atomic\(\)](#), la fonction est ignorée et ne sera jamais appelée.

Points de sauvegarde (« savepoints »)

Les points de sauvegarde (blocs [atomic\(\)](#) imbriqués) sont gérés correctement. C'est-à-dire qu'une fonction inscrite par [on_commit\(\)](#) après un point de sauvegarde (dans un bloc [atomic\(\)](#) imbriqué) sera appelée après le commit de la transaction la plus externe, mais pas dans le cas où une

annulation (rollback) de ce point de sauvegarde ou de tout autre point de sauvegarde précédent se produit pendant la transaction :

```
with transaction.atomic(): # Outer atomic, start a new transaction
    transaction.on_commit(foo)

    with transaction.atomic(): # Inner atomic block, create a savepoint
        transaction.on_commit(bar)

# foo() and then bar() will be called when leaving the outermost block
```

D'un autre côté, lorsqu'un point de sauvegarde est annulé (en raison de l'apparition d'une exception), la fonction définie à l'intérieur de point de sauvegarde ne sera pas appelée :

```
with transaction.atomic(): # Outer atomic, start a new transaction
    transaction.on_commit(foo)

    try:
        with transaction.atomic(): # Inner atomic block, create a savepoint
            transaction.on_commit(bar)
            raise SomeError() # Raising an exception - abort the savepoint
    except SomeError:
        pass

# foo() will be called, but not bar()
```

Ordre d'exécution

Les fonctions `on_commit` d'une transaction donnée sont exécutées dans l'ordre de leur inscription.

Gestion des exceptions

Si l'une des fonctions `on_commit` dans une transaction donnée génère une exception non traitée, aucune fonction inscrite à la suite dans cette même transaction ne sera exécutée. Ceci correspond évidemment au même comportement qui se serait produit si vous aviez exécuté ces fonctions séquentiellement vous-même sans [`on_commit\(\)`](#).

Ordre d'exécution

Les fonctions de rappel sont exécutées *après* un commit réussi, ce qui fait qu'une exception dans une fonction de rappel ne provoquera pas d'annulation de transaction. Elles ne sont exécutées qu'en cas de succès de la transaction, mais *sans* faire partie de la transaction. Pour les cas d'utilisation visés (notifications par courriel, tâches Celery, etc.) cela devrait convenir. Dans le cas contraire (par exemple si l'action à effectuer est si critique que son échec devrait aussi faire échouer la transaction principale), il ne faut alors pas exploiter le point d'entrée [`on_commit\(\)`](#). Une alternative possible est un [`commit` en deux phases](#) tel que la [*prise en charge du protocole de commit en deux phases de psycopg*](#) ou les [*extensions facultatives de commit en deux phases dans la spécification DB-API de Python*](#).

Les fonctions de rappel ne sont pas appelées avant que la connexion ne repasse en mode autocommit après le commit (sinon toute requête dans une fonction de rappel ouvrirait une transaction implicite empêchant la connexion de repasser en mode de commit automatique).

Si l'on se trouve déjà en mode autocommit et en-dehors d'un bloc [atomic\(\)](#), la fonction est immédiatement exécutée sans attendre de commit.

Les fonctions `on_commit` ne fonctionnent qu'en [mode autocommit](#) et avec l'API de transaction [atomic\(\)](#) (ou [ATOMIC REQUESTS](#)). L'appel à [on_commit\(\)](#) lorsque le mode autocommit est désactivé et en-dehors d'un bloc atomique produit une erreur.

Utilisation dans les tests

La classe [TestCase](#) de Django enveloppe chaque test dans une transaction et annule cette transaction après chaque test, afin de garantir l'isolation des tests. Cela signifie qu'aucune transaction n'est réellement suivie d'un commit et donc qu'aucune des fonctions de rappel [on_commit\(\)](#) ne sera jamais exécutée. Si vous avez besoin de tester les résultats d'une fonction de rappel [on_commit\(\)](#), utilisez plutôt une classe [TransactionTestCase](#).

Pourquoi pas de point d'entrée pour les transactions annulées ?

Un point d'entrée pour les transactions annulées (rollback) est plus difficile à implémenter de manière solide qu'un point d'entrée de commit, car plusieurs choses peuvent provoquer une annulation implicite.

Par exemple, si la connexion à la base de données s'interrompt en raison d'un processus tué sans aucune chance de terminaison propre, le point d'entrée d'annulation ne sera jamais exécuté.

La solution est simple : au lieu de faire quelque chose à l'intérieur du bloc atomique (transaction) puis de le défaire en cas d'échec de transaction, utilisez [on_commit\(\)](#) pour déporter la tâche au moment où la transaction a réussi. C'est beaucoup plus facile de défaire quelque chose que l'on n'a jamais fait !

API de bas niveau

Avertissement

Si possible, préférez toujours [atomic\(\)](#). Il prend en compte les particularités de chaque base de données et évite les opérations non valides.

L'API de bas niveau n'est utile que si vous implémentez votre propre gestion des transactions.

Autocommit

Django fournit une API basique dans le module [django.db.transaction](#) pour gérer l'état de validation automatique de chaque connexion de base de données.

`get_autocommit(using=None)`[\[source\]](#)

`set_autocommit(autocommit, using=None)`[\[source\]](#)

Ces fonctions acceptent un paramètre `using` qui doit correspondre au nom d'une base de données. Si ce paramètre n'est pas présent, Django utilise la base de données "default".

La validation automatique (« autocommit ») est initialement activée. Si vous la désactivez, il est de votre responsabilité de la restaurer ensuite.

Dès que vous désactivez la validation automatique, vous obtenez le comportement par défaut de votre adaptateur de base de données et Django ne vous aide plus. Même si ce comportement fait l'objet de la [PEP 249](#), les implémentations des adaptateurs ne sont pas toujours cohérentes entre elles. Parcourez attentivement la documentation de l'adaptateur que vous utilisez.

Vous devez vous assurer qu'aucune transaction n'est pendante, généralement en appelant [commit\(\)](#) ou [rollback\(\)](#) avant de réactiver la validation automatique.

Django refuse de désactiver la validation automatique lorsqu'un bloc [atomic\(\)](#) est actif, car l'atomicité ne serait alors plus respectée.

Transactions

Une transaction est un ensemble atomique de requêtes de base de données. Même si votre programme se plante, la base de données garantit que soit tous les changements seront appliqués, soit aucun.

Django n'offre pas d'API pour démarrer une transaction. La manière attendue de démarrer une transaction est de désactiver la validation automatique avec [set_autocommit\(\)](#).

Une fois dans la transaction, vous pouvez choisir d'appliquer les modifications effectuées jusqu'à ce point avec [commit\(\)](#), ou de toutes les annuler avec [rollback\(\)](#). Ces fonctions sont définies dans [django.db.transaction](#).

`commit(using=None)`[\[source\]](#)

`rollback(using=None)`[\[source\]](#)

Ces fonctions acceptent un paramètre `using` qui doit correspondre au nom d'une base de données. Si ce paramètre n'est pas présent, Django utilise la base de données "default".

Django refuse de valider ou d'annuler une transaction lorsqu'un bloc [atomic\(\)](#) est actif, car l'atomicité ne serait alors plus respectée.

Points de sauvegarde (« savepoints »)

Un point de sauvegarde est un marqueur dans une transaction qui vous permet d'annuler une transaction en partie, plutôt que dans sa totalité. Les points de sauvegarde sont disponibles pour les moteurs SQLite (≥ 3.6.8), PostgreSQL, Oracle et MySQL (avec le moteur de stockage InnoDB).

D'autres moteurs fournissent les fonctions des points de sauvegarde, mais ces fonctions sont vides, elles ne font rien du tout.

Les points de sauvegarde ne sont pas particulièrement utiles quand la validation automatique est active, ce qui est le comportement par défaut de Django. Cependant, dès que vous ouvrez une transaction avec [`atomic\(\)`](#), vous accumulez une série d'opérations de base de données en attente de validation ou d'annulation. Lorsque vous annulez avec un « rollback », toute la transaction est annulée. Les points de sauvegarde permettent d'annuler des opérations de manière plus sélective, plutôt que d'annuler en bloc comme le fait `transaction.rollback()`.

Lorsque le décorateur [`atomic\(\)`](#) est imbriqué, il crée un point de sauvegarde pour permettre une validation ou une annulation partielle. Vous êtes fortement encouragé à utiliser [`atomic\(\)`](#) plutôt que les fonctions présentées ci-dessous, mais elles font tout de même partie de l'API publique, et il n'est pas prévu de les rendre obsolètes.

Chacune de ces fonctions accepte un paramètre `using` devant correspondre au nom de la base de données pour laquelle le comportement s'applique. Si aucun paramètre `using` n'est transmis, c'est la base de données "default" qui est utilisée.

Les points de sauvegarde sont contrôlés par trois fonctions dans [`django.db.transaction`](#):

`savepoint(using=None)`[\[source\]](#)

Crée un nouveau point de sauvegarde. Un point est marqué dans la transaction, à un état qui est reconnu comme « bon ». Renvoie l'identifiant du point de sauvegarde (`sid`).

`savepoint_commit(sid, using=None)`[\[source\]](#)

Libère le point de sauvegarde `sid`. Les modifications effectuées depuis la création du point de sauvegarde sont intégrées dans la transaction.

`savepoint_rollback(sid, using=None)`[\[source\]](#)

Annule la transaction en revenant au point de sauvegarde `sid`.

Ces fonctions ne font rien si les points de sauvegarde ne sont pas pris en charge ou si la base de données est en mode de validation automatique.

Une fonction utilitaire est également disponible :

`clean_savepoints(using=None)`[\[source\]](#)

Réinitialise le compteur utilisé pour générer les identifiants uniques des points de sauvegarde.

L'exemple suivant illustre l'utilisation des points de sauvegarde :

```
from django.db import transaction
```

```

# open a transaction
@transaction.atomic
def viewfunc(request):

    a.save()
    # transaction now contains a.save()

    sid = transaction.savepoint()

    b.save()
    # transaction now contains a.save() and b.save()

    if want_to_keep_b:
        transaction.savepoint_commit(sid)
        # open transaction still contains a.save() and b.save()
    else:
        transaction.savepoint_rollback(sid)
        # open transaction now contains only a.save()

```

Les points de sauvegarde peuvent être utilisés pour se rétablir après une erreur de base de données en effectuant une annulation partielle des opérations. Si vous faites cela à l'intérieur d'un bloc `atomic()`, tout le bloc sera quand même annulé, car Django ne sait pas que vous avez géré la situation à un plus bas niveau ! Pour empêcher cela, vous pouvez contrôler le comportement d'annulation avec les fonctions suivantes.

`get_rollback(using=None)`[\[source\]](#)

`set_rollback(rollback, using=None)`[\[source\]](#)

En définissant le drapeau `rollback` à `True`, vous forcez une annulation lorsque vous sortez du bloc `atomic` le plus proche. Cela peut être utile pour provoquer une annulation sans générer d'exception.

En le définissant à `False`, vous empêchez une telle annulation. Avant de faire cela, assurez-vous d'avoir bien annulé la transaction jusqu'à un point de sauvegarde en bon état à l'intérieur du bloc `atomic` actuel. Sinon, vous cassez l'atomicité et des corruptions de données peuvent apparaître.

Notes spécifiques à certaines bases de données

Points de sauvegarde dans

Même si les points de sauvegarde sont pris à charge à partir de SQLite ≥ 3.6.8, un défaut de conception dans le module [sqlite3](#) les rend presque inutilisables.

Lorsque la validation automatique est active, les points de sauvegarde n'ont pas de raison d'être. Dans le cas contraire, [sqlite3](#) valide implicitement la transaction avant les instructions de points de sauvegarde (en fait, il valide avant toute instruction autre que `SELECT`, `INSERT`, `UPDATE`, `DELETE` et `REPLACE`). Ce bogue a deux conséquences :

- L'API de bas niveau des points de sauvegarde n'est utilisable qu'à l'intérieur d'une transaction, c'est-à-dire dans un bloc `atomic()`.
- Il est impossible d'utiliser `atomic()` lorsque la validation automatique est désactivée.

Transactions dans MySQL

Si vous utilisez MySQL, la prise en charge des transactions par vos tables varie ; tout dépend de la version de MySQL et des types de tables que vous utilisez (par « type de table », nous entendons quelque chose comme « InnoDB » ou « MyISAM »). Les particularités des transactions de MySQL vont au-delà du thème de cette documentation, mais le site de MySQL possède des [informations sur les transactions dans MySQL](#).

Si votre configuration MySQL ne gère *pas* les transactions, Django fonctionne toujours en mode de validation automatique : les instructions sont exécutées et validées dès qu'elles sont émises. Si votre configuration MySQL *gère* les transactions, Django traite les transactions comme expliqué dans ce document.

Traitement des exceptions dans les transactions PostgreSQL

Note

Cette section n'a de sens que si vous implémentez votre propre gestion des transactions. Ce problème ne peut pas survenir dans le mode par défaut de Django et `atomic()` s'en charge automatiquement.

À l'intérieur d'une transaction, lorsque l'appel à un curseur PostgreSQL génère une exception (typiquement `IntegrityError`), toutes les commandes SQL suivantes dans la même transaction échouent avec l'erreur « current transaction is aborted, queries ignored until end of transaction block ». Bien qu'il soit improbable qu'une utilisation simple de `save()` génère une exception avec PostgreSQL, il y a des schémas d'utilisation plus pointus qui sont susceptibles de le faire, comme l'enregistrement d'objets avec des champs uniques, l'enregistrement avec les options `force_insert/force_update` ou l'appel à des instructions SQL personnalisées.

Il existe plusieurs manières de se sortir de ce genre d'erreurs.

Annulation de la transaction

La première option est d'annuler la totalité de la transaction. Par exemple :

```
a.save() # Succeeds, but may be undone by transaction rollback
try:
    b.save() # Could throw exception
except IntegrityError:
    transaction.rollback()
c.save() # Succeeds, but a.save() may have been undone
```

L'appel de `transaction.rollback()` annule la totalité de la transaction. Toute opération de base de données non validée sera perdue. Dans cet exemple, les modifications effectuées par `a.save()` seront perdues, même si cette opération n'a pas elle-même généré d'erreur.

Annulation du point de sauvegarde

Vous pouvez utiliser des [points de sauvegarde](#) pour contrôler l'étendue d'une annulation. Avant d'effectuer une opération de base de données potentiellement délicate, vous pouvez définir ou mettre à jour le point de sauvegarde ; de cette façon, si l'opération échoue, vous pouvez annuler précisément l'opération concernée, plutôt que la totalité de la transaction. Par exemple :

```
a.save() # Succeeds, and never undone by savepoint rollback
sid = transaction.savepoint()
try:
    b.save() # Could throw exception
    transaction.savepoint_commit(sid)
except IntegrityError:
    transaction.savepoint_rollback(sid)
c.save() # Succeeds, and a.save() is never undone
```

Dans cet exemple, `a.save()` ne sera pas annulé dans le cas où `b.save()` génère une exception.

[Lancement de requêtes SQL brutes](#)

[Bases de données multiples](#)

Bases de données multiples

Ce guide thématique décrit la prise en charge de plusieurs bases de données par Django. La plupart de la documentation de Django suppose que vous travaillez avec une seule base de données. Si vous avez besoin de travailler avec plusieurs bases de données, il sera nécessaire de procéder à quelques configurations supplémentaires.

Définition des bases de données

La première chose à faire lorsqu'on veut utiliser plus d'une base de données avec Django est de le renseigner au sujet des serveurs de base de données à utiliser. Cela se fait au niveau du réglage [DATABASES](#). Ce réglage fait correspondre des alias de bases de données, qui sont une manière de se référer à des bases de données spécifiques dans le code de Django, à un dictionnaire de réglages pour la connexion concernée. Les réglages du dictionnaire interne sont détaillés dans la documentation de [DATABASES](#).

L'alias de base de données peut être librement choisi. Cependant, l'alias `default` a une signification spéciale. Django utilise la base de données ayant l'alias `default` lorsqu'aucune autre base de données n'a été sélectionnée.

L'exemple suivant est un extrait de `settings.py` définissant deux bases de données, une base de données PostgreSQL par défaut et une base de données MySQL nommée `users`:

```
DATABASES = {
    'default': {
        'NAME': 'app_data',
        'ENGINE': 'django.db.backends.postgresql',
        'USER': 'postgres_user',
        'PASSWORD': 's3krit'
    },
    'users': {
        'NAME': 'user_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'priv4te'
    }
}
```

Si le concept d'une base de données par défaut n'a pas de sens dans le contexte de votre projet, vous devez vous efforcer de systématiquement indiquer la base de données à utiliser. Django exige qu'une base de données nommée `default` soit définie, mais le dictionnaire des paramètres peut rester vierge s'il n'est pas utilisé. Vous devez configurer [DATABASE_ROUTERS](#) pour tous les modèles de vos applications, y compris ceux des applications « contrib » ou d'applications tierces que vous utilisez, afin qu'aucune requête ne soit dirigée vers la base de données par défaut. L'exemple suivant est un extrait de `settings.py` définissant deux bases de données spécifiques, avec l'entrée `default` volontairement laissée vide :

```
DATABASES = {
    'default': {},
    'users': {
        'NAME': 'user_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'superS3cret'
    },
    'customers': {
        'NAME': 'customer_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_cust',
        'PASSWORD': 'veryPriv@ate'
    }
}
```

Si vous essayez d'accéder à une base de données qui n'y pas été définie dans le réglage [DATABASES](#), Django génère une exception `django.db.utils.ConnectionDoesNotExist`.

Synchronisation des bases de données

La commande de gestion [migrate](#) opère sur une seule base de données à la fois. Par défaut, elle agit sur la base de données `default`, mais en renseignant l'option [--database](#), vous pouvez lui demander de synchroniser une autre base de données. Ainsi, pour synchroniser tous les modèles de toutes les bases de données de notre exemple, il faudrait exécuter :

```
$ ./manage.py migrate
$ ./manage.py migrate --database=users
```

Si vous ne voulez pas que chaque application soit synchronisée vers une base de données particulière, vous pouvez définir un [routeur de base de données](#) implémentant une politique de restriction de disponibilité de certains modèles.

Utilisation d'autres commandes de gestion

Les autres commandes `django-admin` interagissant avec la base de données fonctionnent de la même façon que [migrate](#), c'est-à-dire qu'ils n'opèrent toujours que sur une seule base de données à la fois, en se basant sur le paramètre `--database` pour savoir quelle base de données utiliser.

Routage automatique de base de données

La façon la plus simple d'utiliser plusieurs bases de données est de définir un plan de routage de base de données. Le plan de routage par défaut garantit que les objets restent « attachés » à leur base de données originale (c'est-à-dire qu'un objet extrait de la base de données `foo` sera également enregistré dans la même base). Le plan de routage par défaut garantit que sans indication particulière de base de données, toutes les requêtes sont dirigées vers la base de données `default`.

Vous n'avez rien à faire de particulier pour activer le plan de routage par défaut, il est activé d'origine pour tout projet Django. Cependant, si vous avez l'intention d'implémenter un comportement plus élaboré de distribution entre bases de données, vous avez la possibilité de définir et installer vos propres routeurs de base de données.

Routeurs de base de données

Un routeur de base de données (`Router`) est une classe fournissant jusqu'à quatre méthodes :

```
db_for_read(model, **hints)
```

Suggère la base de données à utiliser pour les opérations de lecture pour les objets de type `model`.

Si une opération de base de données est capable de fournir des informations supplémentaires pouvant aider à choisir la base de données, celles-ci seront contenues dans le dictionnaire `hints`. Vous trouverez [ci-dessous](#) plus de détails au sujet des contenus possibles de `hints`.

Renvoie `None` s'il n'y a pas de suggestion.

`db_for_write(model, **hints)`

Suggère la base de données à utiliser pour les opérations d'écriture pour les objets de type `Model`.

Si une opération de base de données est capable de fournir des informations supplémentaires pouvant aider à choisir la base de données, celles-ci seront contenues dans le dictionnaire `hints`. Vous trouverez [ci-dessous](#) plus de détails au sujet des contenus possibles de `hints`.

Renvoie `None` s'il n'y a pas de suggestion.

`allow_relation(obj1, obj2, **hints)`

Renvoie `True` si une relation entre `obj1` et `obj2` est autorisée, `False` si la relation est interdite ou `None` si le routeur n'a pas d'avis. Il s'agit purement d'une opération de validation utilisée par les opérations de clé étrangère et de relations plusieurs-à-plusieurs pour déterminer si une relation entre deux objets doit être permise ou non.

`allow_migrate(db, app_label, model_name=None, **hints)`

Détermine si l'opération de migration est autorisée à s'exécuter pour la base de données ayant l'alias `db`. Renvoie `True` si l'opération doit être appliquée, `False` si elle ne doit pas l'être ou `None` si le routeur n'a pas d'avis.

Le paramètre positionnel `app_label` est l'étiquette de l'application en cours de migration.

`model_name` est défini par la plupart des opérations de migration à la valeur de `model._meta.model_name` (la version en minuscules du `__name__` du modèle) du modèle en cours de migration. Sa valeur est `None` pour les opérations [RunPython](#) et [RunSQL](#) sauf si elles le fournissent par des indications (`hints`).

`hints` est utilisé par certaines opérations pour communiquer des informations supplémentaires au routeur.

Lorsque `model_name` est défini, `hints` contient normalement la classe du modèle dans la clé `'model'`. Notez qu'il peut s'agir d'un [modèle historique](#), et de ce fait ne pas comporter d'attributs, méthodes ou gestionnaires personnalisés. Vous ne pouvez compter que sur `_meta`.

Cette méthode peut aussi être utilisée pour déterminer la disponibilité d'un modèle dans une base de données précise.

Notez que les migrations vont simplement omettre silencieusement toute opération pour un modèle pour lequel cette méthode renvoie `False`. Cela peut aboutir à des clés étrangères brisées, des tables en trop ou manquantes si vous changez cela après avoir déjà appliqué certaines migrations.

Changed in Django 1.8:

La signature de `allow_migrate` a changé de manière significative depuis les versions précédentes. Consultez les [notes d'obsolescence](#) pour plus de détails.

Un routeur ne doit pas absolument implémenter *toutes* ces méthodes, il peut en omettre une ou plusieurs. Si l'une des méthode est absente, Django saute ce routeur lorsqu'il effectue le contrôle correspondant.

Indications (**hints**)

Les indications reçues par le routeur de base de données dans le dictionnaire `hints` peuvent servir à décider quelle base de données doit recevoir une requête donnée.

Actuellement, la seule indication fournie est `instance`, une instance d'objet liée à l'opération de lecture ou d'écriture en cours. Il peut s'agir de l'instance en train d'être enregistrée ou d'une instance à ajouter dans une relation plusieurs-à-plusieurs. Dans certains cas, aucune indication d'instance n'est présente. Le routeur vérifie l'existence d'une indication d'instance et détermine lui-même si cette instance doit être utilisée pour modifier le comportement de routage.

Utilisation des routeurs

Les routeurs de base de données sont installés par le réglage [DATABASE_ROUTERS](#). Ce réglage définit une liste de noms de classes, chacune définissant un routeur devant être utilisé par le routeur maître (`django.db.router`).

Le routeur maître est utilisé par les opérations de base de données de Django pour désigner les bases de données à utiliser. Chaque fois qu'une requête a besoin de savoir quelle base de données utiliser, elle appelle le routeur maître, en indiquant un modèle ainsi qu'un indice (si disponible). Django essaie ensuite chaque routeur tour à tour jusqu'à ce qu'une suggestion de base de données soit trouvée. Si aucune suggestion n'est trouvée, il essaie d'utiliser la propriété `_state.db` de l'instance de l'indice. Si aucun indice n'a été indiqué ou si l'instance n'a actuellement pas d'état de base de données, le routeur maître désigne la base de données `default`.

Un exemple

À titre d'exemple uniquement !

Cet exemple est prévu pour démontrer la manière dont l'infrastructure de routage peut être utilisée pour modifier l'utilisation des bases de données. Il ignore volontairement certaines questions complexes afin de démontrer comment les routeurs peuvent être utilisés.

Cet exemple ne fonctionnera pas si l'un des modèles de `myapp` contient des relations vers des modèles stockés ailleurs que dans la base de données `other`. [Les relations croisées entre base de données](#) induisent des problèmes d'intégrité référentielle que Django ne peut actuellement pas gérer.

La configuration primaire/réplique présentée ici (désignée comme maître/esclave par certaines bases de données) est également biaisée, car elle ne propose aucune solution pour gérer les délais de réplication (c'est-à-dire les incohérences de requêtes dues au temps nécessaire à propager les écritures vers les répliques). Elle ne considère pas non plus l'interaction des transactions avec la stratégie d'utilisation des bases de données.

Mais qu'est-ce que cela signifie en pratique ? Considérons un autre exemple de configuration. Celle-ci contiendra plusieurs bases de données : une pour l'application `auth` et toutes les autres applications utiliseront une configuration primaire/réplique avec deux répliques en lecture seule. Voici les réglages définissant ces bases de données :

```
DATABASES = {
    'default': {},
    'auth_db': {
        'NAME': 'auth_db',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'swordfish',
    },
    'primary': {
        'NAME': 'primary',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'spam',
    },
    'replica1': {
        'NAME': 'replica1',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'eggs',
    },
    'replica2': {
        'NAME': 'replica2',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'bacon',
    },
}
```

Nous devons maintenant gérer le routage. Nous désirons premièrement un routeur sachant envoyer des requêtes pour l'application `auth` vers `auth_db`:

```
class AuthRouter(object):
    """
    A router to control all database operations on models in the
    auth application.
    """
    def db_for_read(self, model, **hints):
        """
        Attempts to read auth models go to auth_db.
        """
        if model._meta.app_label == 'auth':
            return 'auth_db'
        return None
```

```

def db_for_write(self, model, **hints):
    """
    Attempts to write auth models go to auth_db.
    """
    if model._meta.app_label == 'auth':
        return 'auth_db'
    return None

def allow_relation(self, obj1, obj2, **hints):
    """
    Allow relations if a model in the auth app is involved.
    """
    if obj1._meta.app_label == 'auth' or \
        obj2._meta.app_label == 'auth':
        return True
    return None

def allow_migrate(self, db, app_label, model_name=None, **hints):
    """
    Make sure the auth app only appears in the 'auth_db'
    database.
    """
    if app_label == 'auth':
        return db == 'auth_db'
    return None

```

Et nous voulons également un routeur pour que toutes les autres applications envoient leurs requêtes vers la configuration primaire/réplique, tout en choisissant au hasard une réplication pour la lecture :

```

import random

class PrimaryReplicaRouter(object):
    def db_for_read(self, model, **hints):
        """
        Reads go to a randomly-chosen replica.
        """
        return random.choice(['replica1', 'replica2'])

    def db_for_write(self, model, **hints):
        """
        Writes always go to primary.
        """
        return 'primary'

    def allow_relation(self, obj1, obj2, **hints):
        """
        Relations between objects are allowed if both objects are
        in the primary/replica pool.
        """
        db_list = ('primary', 'replica1', 'replica2')
        if obj1._state.db in db_list and obj2._state.db in db_list:
            return True
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        """
        All non-auth models end up in this pool.
        """
        return True

```

Pour terminer, nous ajoutons dans le fichier de réglages le code suivant (remplaçant `path.to.` par le chemin Python réel vers les modules où les routeurs sont définis) :

```
DATABASE_ROUTERS = ['path.to.AuthRouter', 'path.to.PrimaryReplicaRouter']
```

L'ordre dans lequel les routeurs sont traités a son importance. Les routeurs sont interrogés dans leur ordre d'apparition dans le réglage [DATABASE_ROUTERS](#). Dans cet exemple, `AuthRouter` est traité avant `PrimaryReplicaRouter`, et par conséquent, les décisions concernant les modèles de `auth` sont prises avant toute autre décision. Si le réglage [DATABASE_ROUTERS](#) contenait les deux routeurs dans l'ordre inverse, `PrimaryReplicaRouter.allow_migrate()` serait traité en premier. La nature « ramasse-tout » de l'implémentation de `PrimaryReplicaRouter` signifierait que tous les modèles seraient accessibles dans toutes les bases de données.

Avec l'installation de cette configuration, exécutons quelque extraits de code Django :

```
>>> # This retrieval will be performed on the 'auth_db' database
>>> fred = User.objects.get(username='fred')
>>> fred.first_name = 'Frederick'

>>> # This save will also be directed to 'auth_db'
>>> fred.save()

>>> # These retrieval will be randomly allocated to a replica database
>>> dna = Person.objects.get(name='Douglas Adams')

>>> # A new object has no database allocation when created
>>> mh = Book(title='Mostly Harmless')

>>> # This assignment will consult the router, and set mh onto
>>> # the same database as the author object
>>> mh.author = dna

>>> # This save will force the 'mh' instance onto the primary database...
>>> mh.save()

>>> # ... but if we re-retrieve the object, it will come back on a replica
>>> mh = Book.objects.get(title='Mostly Harmless')
```

Sélection manuelle d'une base de données

Django offre également une API permettant de conserver un contrôle total sur l'utilisation des bases de données dans votre code. L'attribution manuelle d'une base de données est prioritaire par rapport à l'attribution provenant d'un routeur.

Sélection manuelle d'une base de données pour un QuerySet

Vous pouvez choisir la base de données d'un `QuerySet` à tout moment dans la « chaîne » du `QuerySet`. Il suffit d'appeler `using()` sur le `QuerySet` pour obtenir un autre `QuerySet` utilisant la base de données indiquée.

`using()` accepte un seul paramètre : l'alias de la base de données devant recevoir la requête à exécuter. Par exemple :

```
>>> # This will run on the 'default' database.
>>> Author.objects.all()

>>> # So will this.
>>> Author.objects.using('default').all()

>>> # This will run on the 'other' database.
>>> Author.objects.using('other').all()
```

Sélection d'une base de données pour `save()`

Utilisez le paramètre nommé `using` de `Model.save()` pour indiquer dans quelle base de données les données doivent être enregistrées.

Par exemple, pour enregistrer un objet dans la base de données `legacy_users`, il faut écrire ceci :

```
>>> my_object.save(using='legacy_users')
```

Si vous ne renseignez pas `using`, la méthode `save()` enregistre dans la base de données par défaut attribuée par les routeurs.

Déplacement d'un objet entre bases de données

Si vous avez enregistré une instance dans une base de données, il peut être tentant d'utiliser `save(using=...)` comme astuce pour copier l'instance vers une nouvelle base de données. Cependant, si vous ne prenez pas certaines précautions nécessaires, cela peut amener à des conséquences inattendues.

Considérez l'exemple suivant :

```
>>> p = Person(name='Fred')
>>> p.save(using='first') # (statement 1)
>>> p.save(using='second') # (statement 2)
```

Dans l'instruction 1, un nouvel objet `Person` est enregistré dans la base de données `first`. À ce moment, `p` n'a pas encore de clé primaire, ce qui fait que Django génère une instruction SQL `INSERT`. La clé primaire est donc créée et Django attribue cette clé primaire à `p`.

Lorsque l'enregistrement est effectué à l'instruction 2, `p` a déjà reçu une valeur de clé primaire et Django va utiliser cette clé primaire dans la nouvelle base de données. Si cette valeur de clé primaire n'est pas déjà utilisée dans la base de données `second`, vous n'aurez pas de problème, l'objet sera effectivement copié dans la nouvelle base de données.

Cependant, si la clé primaire de `p` est déjà utilisée dans la base de données `second`, l'objet existant dans la base de données `second` sera écrasé lors de l'enregistrement de `p`.

Vous pouvez empêcher cela de deux manières. Premièrement, vous pouvez effacer la clé primaire de l'instance. Si un objet n'a pas de clé primaire, Django le traite comme un nouvel objet, évitant ainsi toute perte de données dans la base de données `second`:

```
>>> p = Person(name='Fred')
>>> p.save(using='first')
>>> p.pk = None # Clear the primary key.
>>> p.save(using='second') # Write a completely new object.
```

La seconde possibilité est d'utiliser l'option `force_insert` de `save()` pour être certain que Django produise une instruction SQL `INSERT`:

```
>>> p = Person(name='Fred')
>>> p.save(using='first')
>>> p.save(using='second', force_insert=True)
```

Cela garantit que la personne nommée `Fred` possédera la même clé primaire dans les deux bases de données. Si cette clé primaire est déjà utilisée lorsque vous essayez d'enregistrer dans la base de données `second`, une erreur sera générée.

Sélection d'une base de données pour la suppression

Par défaut, un appel à la suppression d'un objet existant sera exécuté dans la même base de données qui a été utilisée pour extraire l'objet au préalable :

```
>>> u = User.objects.using('legacy_users').get(username='fred')
>>> u.delete() # will delete from the `legacy_users` database
```

Pour indiquer la base de données dans laquelle un modèle sera supprimé, transmettez un paramètre nommé `using` à la méthode `Model.delete()`. Ce paramètre joue le même rôle que le même paramètre avec `save()`.

Par exemple, si vous migrez un utilisateur à partir de la base de données `legacy_users` vers la base de données `new_users`, voici les commandes que vous pourriez utiliser :

```
>>> user_obj.save(using='new_users')
>>> user_obj.delete(using='legacy_users')
```

Utilisation de gestionnaires avec plusieurs bases de données

Utilisez la méthode `db_manager()` des gestionnaires pour donner à ceux-ci accès à une base de données autre que celle par défaut.

Par exemple, admettons que vous ayez une méthode d'un gestionnaire personnalisé agissant sur la base de données, `User.objects.create_user()`. Comme `create_user()` est une méthode de gestionnaire, et non une méthode de `QuerySet`, il n'est pas possible d'écrire

`User.objects.using('new_users').create_user()` (la méthode `create_user()`

n'est disponible que sur `User.objects`, le gestionnaire, et non pas sur les objets `QuerySet` dérivés du gestionnaire). La solution est d'utiliser `db_manager()`, comme ceci :

```
User.objects.db_manager('new_users').create_user(...)
```

`db_manager()` renvoie une copie du gestionnaire lié à la base de données que vous indiquez.

Utilisation de `get_queryset()` avec plusieurs bases de données

Si vous surchargez `get_queryset()` dans votre gestionnaire, prenez soin de soit appeler la méthode du parent (en utilisant `super()`) ou soit de gérer de manière appropriée l'attribut `_db` du gestionnaire (une chaîne contenant le nom de la base de données à utiliser).

Par exemple, si vous souhaitiez renvoyer une classe `QuerySet` personnalisée à partir de la méthode `get_queryset`, vous pourriez faire ceci :

```
class MyManager(models.Manager):
    def get_queryset(self):
        qs = CustomQuerySet(self.model)
        if self._db is not None:
            qs = qs.using(self._db)
        return qs
```

Exposition de plusieurs bases de données dans l'interface d'administration

L'interface d'administration de Django ne contient pas de prise en charge explicite de plusieurs bases de données. Si vous souhaitez mettre à disposition une interface d'administration pour un modèle sur une autre base de données que celle indiquée par votre chaîne de routage, il vous faudra écrire des classes [ModelAdmin](#) personnalisées qui vont piloter l'administration vers l'utilisation d'une base de données spécifique pour le contenu.

Les objets `ModelAdmin` ont cinq méthodes qui nécessitent une adaptation pour la prise en charge de plusieurs bases de données :

```
class MultiDBModelAdmin(admin.ModelAdmin):
    # A handy constant for the name of the alternate database.
    using = 'other'

    def save_model(self, request, obj, form, change):
        # Tell Django to save objects to the 'other' database.
        obj.save(using=self.using)

    def delete_model(self, request, obj):
        # Tell Django to delete objects from the 'other' database
        obj.delete(using=self.using)

    def get_queryset(self, request):
        # Tell Django to look for objects on the 'other' database.
        return super(MultiDBModelAdmin,
self).get_queryset(request).using(self.using)
```

```

def formfield_for_foreignkey(self, db_field, request=None, **kwargs):
    # Tell Django to populate ForeignKey widgets using a query
    # on the 'other' database.
    return super(MultiDBModelAdmin, self).formfield_for_foreignkey(db_field,
request=request, using=self.using, **kwargs)

def formfield_for_manytomany(self, db_field, request=None, **kwargs):
    # Tell Django to populate ManyToMany widgets using a query
    # on the 'other' database.
    return super(MultiDBModelAdmin, self).formfield_for_manytomany(db_field,
request=request, using=self.using, **kwargs)

```

L'exemple présenté ci-dessus implémente une stratégie basée sur plusieurs bases de données où tous les objets d'un type donné sont stockés dans une base de données spécifique (par ex. tous les objets `User` se trouvent dans la base de données `other`). Si votre utilisation de plusieurs bases de données est plus complexe, votre classe `ModelAdmin` devra refléter cette stratégie.

Les sous-formulaires peuvent être gérés d'une manière similaire. Ils nécessitent trois méthodes personnalisées :

```

class MultiDBTabularInline(admin.TabularInline):
    using = 'other'

    def get_queryset(self, request):
        # Tell Django to look for inline objects on the 'other' database.
        return super(MultiDBTabularInline,
self).get_queryset(request).using(self.using)

    def formfield_for_foreignkey(self, db_field, request=None, **kwargs):
        # Tell Django to populate ForeignKey widgets using a query
        # on the 'other' database.
        return super(MultiDBTabularInline, self).formfield_for_foreignkey(db_field,
request=request, using=self.using, **kwargs)

    def formfield_for_manytomany(self, db_field, request=None, **kwargs):
        # Tell Django to populate ManyToMany widgets using a query
        # on the 'other' database.
        return super(MultiDBTabularInline, self).formfield_for_manytomany(db_field,
request=request, using=self.using, **kwargs)

```

Après avoir écrit les définitions des classes d'administration des modèles, celles-ci peuvent être inscrites dans n'importe quelle instance `Admin`:

```

from django.contrib import admin

# Specialize the multi-db admin objects for use with specific models.
class BookInline(MultiDBTabularInline):
    model = Book

class PublisherAdmin(MultiDBModelAdmin):
    inlines = [BookInline]

admin.site.register(Author, MultiDBModelAdmin)
admin.site.register(Publisher, PublisherAdmin)

```

```
othersite = admin.AdminSite('othersite')
othersite.register(Publisher, MultiDBModelAdmin)
```

Cet exemple configure deux sites d'administration. Dans le premier site, les objets `Author` et `Publisher` sont exposés ; les objets `Publisher` possèdent un sous-formulaire tabulaire affichant les livres publiés par cet éditeur. Le second site n'expose que les éditeurs, sans sous-formulaire.

Utilisation directe de curseurs avec plusieurs bases de données

Si vous utilisez plus d'une base de données, vous pouvez utiliser `django.db.connections` pour obtenir la connexion (et le curseur) d'une base de données spécifique. `django.db.connections` est un objet de type dictionnaire permettant d'obtenir une connexion particulière au moyen de son alias :

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
```

Limites des bases de données multiples

Relations croisées entre bases de données

Django n'offre actuellement aucune prise en charge des clés étrangères ou des relations plusieurs-à-plusieurs entre différentes bases de données. Si vous avez utilisé un routeur pour distribuer les modèles entre différentes bases de données, toute relation de clé étrangère ou plusieurs-à-plusieurs définie par ces modèles doit être interne à une seule base de données.

L'intégrité référentielle en est la cause. Afin de maintenir une relation entre deux objets, Django a besoin de savoir que la clé primaire de l'objet lié est valide. Si la clé primaire est stockée dans une autre base de données, il n'est pas possible d'évaluer de manière simple la validité de la clé primaire.

Si vous utilisez Postgres, Oracle ou MySQL avec InnoDB, cette garantie d'intégrité est assurée au niveau de la base de données, les contraintes de clés au niveau de la base de données empêchant la création de relations qui ne peuvent pas être validées.

Cependant, si vous utilisez SQLite ou MySQL avec des tables MyISAM, il n'y a pas d'intégrité référentielle ; par conséquent, il est possible de créer des clés étrangères « simulées » entre bases de données. Toutefois, cette configuration n'est pas prise en charge officiellement par Django.

Comportement des applications contribuéées

Plusieurs applications contribuéées contiennent des modèles, et certaines applications ont des dépendances. Comme les relations croisées entre bases de données sont impossibles, il en résulte certaines restrictions sur la façon dont ces modèles peuvent être partagés entre différentes bases de données :

- Les modèles `contentType`, `session` et `site` peuvent être indépendamment stockés dans n'importe quelle base de données, pour autant qu'un routeur adéquat soit utilisé.
- Les modèles `auth` (`User`, `Group` et `Permission`) sont liés les uns aux autres et à `contentType`, ils doivent donc être stockés dans la même base de données que `contentType`.
- `admin` dépend de `auth`, donc ses modèles doivent se trouver dans la même base de données que `auth`.
- `flatpages` et `redirects` dépendent de `sites`, donc leurs modèles doivent être dans la même base de données que `sites`.

De plus, certains objets sont automatiquement créés juste après que [migrate](#) crée la table pour les stocker en base de données :

- un `Site` par défaut,
- un `ContentType` pour chaque modèle (y compris ceux qui ne sont pas stockés en base de données),
- trois objets `Permission` pour chaque modèle (y compris ceux qui ne sont pas stockés en base de données).

Pour des configurations courantes avec plusieurs bases de données, il n'est pas utile de disposer de ces objets dans plus d'une base de données. Ces configurations courantes incluent les configurations maître-esclave et la connexion à des bases de données externes. Il est donc recommandé d'écrire un [routeur de base de données](#) autorisant la synchronisation de ces trois modèles vers une seule base de données. Utilisez la même approche pour les autres applications contribuéées ou de tierce partie dont les tables n'ont pas besoin de se trouver dans plusieurs bases de données.

Avertissement

Si vous synchronisez les types de contenus dans plus d'une base de données, soyez conscient que leur clés primaires peuvent différer d'une base de données à l'autre. Cela peut aboutir à des corruptions ou des pertes de données.

[Transactions de base de données](#)
[Espaces de tables](#)

Espaces de tables

Un paradigme courant dans l'optimisation des performances des systèmes de base de données est l'utilisation des [espaces de tables](#) pour organiser l'agencement sur disque.

Avertissement

Django ne crée pas d'espace de tables pour vous. Consultez la documentation de votre moteur de base de données pour plus de détails sur la création et la gestion d'espaces de tables.

Déclaration d'espaces de tables pour les tables

Un espace de tables peut être indiqué pour une table générée par un modèle en renseignant l'option [db_tablespace](#) dans la class `Meta` du modèle. Cette option affecte aussi les tables créées automatiquement pour les champs [ManyToManyField](#) du modèle.

Vous pouvez utiliser le réglage [DEFAULT_TABLESPACE](#) pour indiquer une valeur par défaut pour [db_tablespace](#). C'est utile pour définir un espace de tables pour les applications Django de base et pour d'autres applications dont vous ne maîtrisez pas le code.

Déclaration des espaces de tables pour les index

Vous pouvez transmettre l'option [db_tablespace](#) à un constructeur d'un champ `Field` pour indiquer un espace de tables différent pour l'index de la colonne du champ. Si aucun index n'est créé pour la colonne en question, l'option est ignorée.

Vous pouvez utiliser le réglage [DEFAULT_INDEX_TABLESPACE](#) pour indiquer une valeur par défaut pour [db_tablespace](#).

Si [db_tablespace](#) n'est pas renseigné et que [DEFAULT_INDEX_TABLESPACE](#) n'a pas été défini, l'index est créé dans le même espace de table que les tables.

Un exemple

```
class TablespaceExample(models.Model):
    name = models.CharField(max_length=30, db_index=True, db_tablespace="indexes")
    data = models.CharField(max_length=255, db_index=True)
    edges = models.ManyToManyField(to="self", db_tablespace="indexes")

    class Meta:
        db_tablespace = "tables"
```

Dans cet exemple, les tables générées par le modèle `TablespaceExample` (c'est-à-dire la table du modèle et la table plusieurs-à-plusieurs) seront stockées dans l'espace de tables `tables`. L'index du champ `name` et les index de la table plusieurs-à-plusieurs seront stockés dans l'espace de tables

indexes. Le champ `data` génère aussi un index, mais aucun espace de tables n'est indiqué, il sera donc stocké par défaut dans l'espace de tables `tables`.

Prise en charge des bases de données

PostgreSQL et Oracle prennent en charge les espaces de tables, au contraire de SQLite et MySQL.

Lorsque vous utilisez un moteur qui ne prend pas en charge les espaces de tables, Django ignore toutes les options liées aux espaces de tables.

[Bases de données multiples](#)

[Optimisation de l'accès à la base de données](#)

Optimisation de l'accès à la base de données

La couche de base de données de Django fournit plusieurs manières d'aider les développeurs à tirer le meilleur de leurs bases de données. Ce document rassemble des liens vers la documentation adéquate et ajoute quelques astuces, réparties dans plusieurs chapitres respectant la chronologie des étapes à suivre lorsqu'on souhaite optimiser l'utilisation de la base de données.

Priorité au profilage

Comme principe de programmation générale, cela va sans dire. Recherchez [quelles sont les requêtes effectuées et ce qu'elles coûtent](#). Il est aussi possible d'utiliser un projet externe comme [django-debug-toolbar](#) ou un outil qui surveille directement votre base de données.

Rappelez-vous que l'on peut optimiser pour la rapidité, pour la mémoire ou pour les deux, en fonction des besoins. Parfois, l'optimisation d'un côté va péjorer la situation de l'autre, mais dans d'autres cas, les deux vont en profiter ensemble. De plus, le travail effectué par le processus de la base de données n'a pas toujours le même coût (pour vous) que le même travail effectué dans le processus Python. C'est à vous de décider des priorités, de pondérer avantages et inconvénients et de profiler tout cela selon les besoins car tout dépend de votre application et du serveur.

Avec tout ce qui suit, n'oubliez pas de profiler après chaque modification pour vous assurer que celle-ci est bénéfique et que le bénéfice est suffisant pour compenser la perte de lisibilité du code. Pour **toutes** les suggestions ci-dessous, il faut tenir compte du risque que le principe général ne s'applique pas dans votre cas de figure, ou que l'effet soit même inversé.

Techniques standards d'optimisation de base de données

Ceci comprend :

- Des [index](#). C'est la priorité n°1, *après* avoir déterminé par analyse de performance quels sont les index nécessaires. Utilisez [Field.db_index](#) ou [Meta.index_together](#) pour en ajouter à partir de Django. Envisagez l'utilisation d'index aux champs que vous interrogez fréquemment avec [filter\(\)](#), [exclude\(\)](#), [order_by\(\)](#), etc. car les index peuvent aider à accélérer les requêtes. Notez que la recherche des meilleurs index est un thème complexe et dépendant du moteur de base de données, également dépendant de l'application à construire. La charge induite par la maintenance d'un index peut anéantir les gains en terme de vitesse de requête.
- Utilisation appropriée des types de champs.

Nous supposons que vous avez effectué les opérations évidentes ci-dessus. La suite de ce document se concentre sur la façon d'utiliser Django de telle manière à ne pas effectuer de travail inutile. Ce document n'aborde cependant pas d'autres techniques d'optimisation qui s'appliquent à des opérations lourdes, comme la [mise en cache à portée générale](#).

Compréhension des objets QuerySet

La compréhension des [QuerySet](#) est capitale pour obtenir de bonnes performances avec du code simple. En particulier :

Compréhension de l'évaluation du QuerySet

Pour éviter des problèmes de performance, il est important de comprendre :

- que les [QuerySet sont différés](#).
- quand [ils sont évalués](#).
- comment [les données sont conservée en mémoire](#).

Compréhension des attributs mis en cache

Tout comme la mise en cache d'un QuerySet global, il y a mise en cache du résultat des attributs dans les objets de l'ORM. En général, les attributs qui ne sont pas exécutables seront mis en cache. Par exemple, à partir des [modèles d'exemple du Weblog](#):

```
>>> entry = Entry.objects.get(id=1)
>>> entry.blog      # Blog object is retrieved at this point
>>> entry.blog      # cached version, no DB access
```

Mais généralement, les attributs exécutables provoquent des accès à la base de données à chaque appel :

```
>>> entry = Entry.objects.get(id=1)
>>> entry.authors.all()  # query performed
>>> entry.authors.all()  # query performed again
```

Soyez prudent dans la lecture du code des gabarits, le système des gabarits ne permet pas d'utiliser des parenthèses, mais les objets exécutables sont appelés automatiquement, masquant la distinction ci-dessus.

Soyez prudent avec vos propres propriétés personnalisées, il vous revient d'implémenter leur mise en cache au besoin, par exemple en utilisant le décorateur [cached_property](#).

Utilisation de la balise de gabarit `with`

Pour faire usage du comportement de cache de `QuerySet`, il peut être nécessaire d'utiliser la balise de gabarit [with](#).

Utilisation de `iterator()`

Lorsque vous manipulez un grand nombre d'objets, le comportement de cache de `QuerySet` peut provoquer une grosse utilisation de la mémoire. Dans ce cas, [iterator\(\)](#) peut aider.

Travail de base de données en base de données plutôt qu'en Python

Par exemple :

- Au niveau le plus élémentaire, utilisez [filter et exclude](#) pour filtrer au niveau de la base de données.
- Utilisez les [expressions F](#) pour filtrer en rapport avec d'autres champs à l'intérieur du même modèle.
- Utilisez les [annotations pour effectuer le travail d'agrégation dans la base de données](#).

Si ces outils ne suffisent pas à générer le code SQL nécessaire :

Utilisation de `RawSQL`

Une méthode moins portable mais plus puissante est d'utiliser l'expression [RawSQL](#) qui permet d'ajouter explicitement du code SQL dans une requête. Si cela ne suffit toujours pas :

Utilisation de SQL brut

Écrivez votre [propre code SQL pour sélectionner des données ou remplir les modèles](#). Utilisez `django.db.connection.queries` pour examiner ce que Django écrit pour vous et considérez cela comme un point de départ.

Sélection d'objets individuels en utilisant une colonne unique et indexée

Il y a deux raisons d'utiliser une colonne avec [unique](#) ou [db_index](#) lorsqu'on utilise [get\(\)](#) pour récupérer des objets individuels. Premièrement, la requête sera plus rapide en raison de l'index de base de données. De plus, la requête pourrait s'effectuer beaucoup plus lentement si plusieurs objets correspondent à la recherche ; avec une contrainte d'unicité sur la colonne, vous avez la garantie que cela ne se produira jamais.

Ainsi, en se basant sur les [modèles d'exemple de Weblog](#):

```
>>> entry = Entry.objects.get(id=10)
```

sera plus rapide que :

```
>>> entry = Entry.objects.get(headline="News Item Title")
```

parce qu'`id` est indexé dans la base de données et que son unicité est garantie.

La ligne suivante est potentiellement très lente :

```
>>> entry = Entry.objects.get(headline__startswith="News")
```

Tout d'abord, `headline` n'est pas indexé, ce qui fait que la base de données sous-jacente prendra plus de temps pour trouver la ligne.

Deuxièmement, cette recherche ne garantit pas qu'un seul objet sera renvoyé. Si la requête correspond à plus d'un objet, elle va tous les sélectionner et les transférer depuis la base de données. Cette pénalité peut être conséquente si des centaines ou des milliers d'enregistrements sont renvoyés. Ce d'autant plus si la base de données est située sur un autre serveur, là où la latence et le surcoût de la liaison réseau jouent aussi un rôle.

Tout récupérer d'un coup quand on en connaît le besoin

Il est en général moins efficace de solliciter la base de données plusieurs fois pour différentes parties d'un ensemble de données que de tout récupérer par une seule requête quand on sait à l'avance que l'on aura besoin du tout. C'est particulièrement important lorsqu'une requête est exécutée dans une boucle et risque au final d'effectuer plusieurs requêtes en base de données alors qu'une seule n'est vraiment nécessaire. Ainsi :

Utilisation de `QuerySet.select_related()` et `prefetch_related()`

Il faut bien comprendre [select_related\(\)](#) et [prefetch_related\(\)](#) et les utiliser :

- dans le code des vues ;

- et dans les [gestionnaires \(par défaut ou non\)](#) lorsque cela fait sens. Ayez conscience des endroits où les gestionnaires sont (ou ne sont pas) utilisés ; ce n'est pas toujours évident de le savoir, ne faites donc pas de simples suppositions.

Ne pas récupérer des éléments inutilement

Utilisation de `QuerySet.values()` et `values_list()`

Lorsque vous ne voulez qu'obtenir un dictionnaire ou une liste de valeurs sans avoir besoin d'objets modèles de l'ORM, faites bon usage de [values\(\)](#). Cette méthode peut être utile pour remplacer des objets modèles dans le code des gabarits ; tant que les dictionnaires résultants possèdent les mêmes attributs que ceux qui sont utilisés dans le gabarit, tout va bien.

Utilisation de `QuerySet.defer()` et `only()`

Utilisez [defer\(\)](#) et [only\(\)](#) quand vous savez à l'avance que vous n'aurez pas besoin de certaines colonnes de base de données (ou non utiles dans la plupart des cas) afin d'éviter de les charger. Notez que si vous les utilisez quand même, l'ORM devra les charger à l'aide d'une nouvelle requête, ce qui peut potentiellement aggraver la situation quand cette technique n'est pas utilisée à bon escient.

Sachez également qu'il y a un léger surcoût lorsque Django doit construire un modèle avec des champs différés. N'utilisez pas trop agressivement les champs différés sans profilage car la base de données doit lire sur le disque la plupart des données non textuelles et non VARCHAR d'un enregistrement des résultats, même lorsqu'elle finit par n'exploiter que quelques colonnes. Les méthodes `defer()` et `only()` sont particulièrement adéquates lorsqu'on peut éviter de charger de grandes quantités de données textuelles ou quand la valeur de certains champs est coûteuse en terme de conversion en Python. Comme toujours, profilez d'abord, optimisez ensuite.

Utilisation de `QuerySet.count()`

...si vous n'avez besoin que du nombre de lignes, et évitez `len(queryset)`.

Utilisation de `QuerySet.exists()`

...si vous voulez uniquement savoir qu'il y a au moins un résultat, et évitez `if queryset`.

Mais :

Ne pas abuser de `count()` et `exists()`

Si vous allez avoir besoin de données réelles du QuerySet, autant l'évaluer.

Par exemple, si on dispose d'un modèle Email ayant un attribut `body` et une relation plusieurs-à-plusieurs vers User, le code de gabarit suivant est optimal :

```
{% if display_inbox %}
```

```
{% with emails=user.emails.all %}
{% if emails %}
    <p>You have {{ emails|length }} email(s)</p>
    {% for email in emails %}
        <p>{{ email.body }}</p>
    {% endfor %}
{% else %}
    <p>No messages today.</p>
{% endif %}
{% endwith %}
{% endif %}
```

Il est optimal car :

1. Comme les objets `QuerySet` sont différés, aucune requête de base de données n'est exécutée si 'display_inbox' vaut `False`.
2. L'emploi de `with` signifie que nous stockons `user.emails.all` dans une variable pour usage ultérieur, permettant ainsi de réutiliser son cache.
3. La ligne `{% if emails %}` provoque l'appel à `QuerySet.__bool__()`, ce qui provoque aussi l'exécution de la requête `user.emails.all()` dans la base de données et la construction du premier résultat au moins comme objet ORM. S'il n'y a pas de résultats, elle renvoie `False`, sinon `True`.
4. L'emploi de `{{ emails|length }}` appelle `QuerySet.__len__()`, ce qui remplit le reste du cache sans effectuer de nouvelle requête.
5. La boucle `for` utilise les données déjà en cache.

Au total, le code effectue une ou zéro requête de base de données. La seule optimisation explicite effectuée est l'emploi de la balise `with`. Si on avait utilisé `QuerySet.exists()` ou `QuerySet.count()` à n'importe quel endroit, cela aurait généré des requêtes supplémentaires.

Utilisation de `QuerySet.update()` et `delete()`

Plutôt que de récupérer un ensemble d'objets, de définir certaines valeurs et de les enregistrer individuellement, utilisez une instruction SQL UPDATE groupée au moyen de [QuerySet.update\(\)](#). De la même façon, effectuez des [suppressions groupées](#) chaque fois que c'est possible.

Notez cependant que des méthodes de mise à jour groupées ne peuvent pas appeler les méthodes `save()` et `delete()` des instances individuelles, ce qui signifie que tout comportement spécialisé que vous auriez ajouté avec ces méthodes ne sera pas exécuté, y compris tout ce qui découle de l'utilisation des [signaux](#) attachés aux objets de base de données.

Utilisation directe de valeurs de clé étrangère

Si vous n'avez besoin que de la valeur d'une clé étrangère, utilisez la valeur qui est déjà stockée dans l'objet que vous manipulez, plutôt que de récupérer tout l'objet lié pour simplement disposer de sa clé primaire. Par exemple, faites :

```
entry.blog_id
```

au lieu de :

```
entry.blog.id
```

Ne pas trier les résultats sans raison

Le tri n'est pas gratuit ; chaque champ de tri est une opération que la base de données doit effectuer. Si un modèle possède un ordre de tri par défaut ([Meta.ordering](#)) et que vous n'en avez pas besoin, vous pouvez l'enlever sur un QuerySet en appelant [order_by\(\)](#) sans paramètre.

L'ajout d'un index à votre base de données peut aider à améliorer les performances de tri.

Insertion groupée

Lors de la création d'objets, utilisez si possible la méthode [bulk_create\(\)](#) pour réduire le nombre de requêtes SQL. Par exemple :

```
Entry.objects.bulk_create([
    Entry(headline="Python 3.0 Released"),
    Entry(headline="Python 3.1 Planned")
])
```

...est préférable à :

```
Entry.objects.create(headline="Python 3.0 Released")
Entry.objects.create(headline="Python 3.1 Planned")
```

Notez qu'il existe un certain nombre de [restrictions à cette méthode](#), il s'agit donc de s'assurer qu'elle correspond à votre cas d'utilisation.

Cela s'applique aussi aux champs [ManyToManyFields](#), ce qui fait que :

```
my_band.members.add(me, my_friend)
```

...est préférable à :

```
my_band.members.add(me)
my_band.members.add(my_friend)
```

...où Bands et Artists sont liés par une relation plusieurs-à-plusieurs.

[Espaces de tables](#)

[Exemples d'utilisation de l'API de relations entre modèles](#)

Relations plusieurs-à-plusieurs

Pour définir une relation plusieurs-à-plusieurs, utilisez un champ [ManyToManyField](#).

Dans cet exemple, un `Article` peut être publié dans plusieurs objets `Publication` et une `Publication` possède plusieurs objets `Article`:

```
from django.db import models

class Publication(models.Model):
    title = models.CharField(max_length=30)

    def __str__(self):
        return self.title

    class Meta:
        ordering = ('title',)

class Article(models.Model):
    headline = models.CharField(max_length=100)
    publications = models.ManyToManyField(Publication)

    def __str__(self):
        return self.headline

    class Meta:
        ordering = ('headline',)
```

Vous trouverez ci-dessous des exemples d'opérations qui peuvent être effectuées en utilisant les capacités de l'API Python. Notez que si vous utilisez [un modèle intermédiaire](#) pour une relation plusieurs-à-plusieurs, certaines des méthodes du gestionnaire connexe sont désactivées, de sorte que certains de ces exemples ne fonctionneront pas avec ces modèles.

Créez quelques `Publications`:

```
>>> p1 = Publication(title='The Python Journal')
>>> p1.save()
>>> p2 = Publication(title='Science News')
>>> p2.save()
>>> p3 = Publication(title='Science Weekly')
>>> p3.save()
```

Créez un `Article`:

```
>>> a1 = Article(headline='Django lets you build Web apps easily')
```

Vous ne pouvez pas le lier à une `Publication` avant qu'il ait été enregistré :

```
>>> a1.publications.add(p1)
Traceback (most recent call last):
...
```

ValueError: 'Article' instance needs to have a primary key value before a many-to-many relationship can be used.

Enregistrez-le ! :

```
>>> a1.save()
```

Associez l'Article à une Publication:

```
>>> a1.publications.add(p1)
```

Créez un autre Article et faites-le paraître dans les deux Publications:

```
>>> a2 = Article(headline='NASA uses Python')
>>> a2.save()
>>> a2.publications.add(p1, p2)
>>> a2.publications.add(p3)
```

Ajouter une seconde fois ne pose pas problème :

```
>>> a2.publications.add(p3)
```

L'ajout d'un objet du mauvais type génère une exception [TypeError](#):

```
>>> a2.publications.add(a1)
Traceback (most recent call last):
...
TypeError: 'Publication' instance expected
```

Créez et ajoutez une Publication à un Article en une seule étape en utilisant [create\(\)](#):

```
>>> new_publication = a2.publications.create(title='Highlights for Children')
```

Les objets Article ont accès à leurs objets Publication liés :

```
>>> a1.publications.all()
[<Publication: The Python Journal>]
>>> a2.publications.all()
[<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>, <Publication: The Python Journal>]
```

Les objets Publication ont accès à leurs objets Article liés :

```
>>> p2.article_set.all()
[<Article: NASA uses Python>]
>>> p1.article_set.all()
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
>>> Publication.objects.get(id=4).article_set.all()
[<Article: NASA uses Python>]
```

Il est possible d'interroger les relations plusieurs-à-plusieurs en utilisant des [recherches traversant les relations](#):

```
>>> Article.objects.filter(publications__id=1)
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
```

```
>>> Article.objects.filter(publications__pk=1)
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
>>> Article.objects.filter(publications=1)
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
>>> Article.objects.filter(publications=p1)
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]

>>> Article.objects.filter(publications__title__startswith="Science")
[<Article: NASA uses Python>, <Article: NASA uses Python>]

>>> Article.objects.filter(publications__title__startswith="Science").distinct()
[<Article: NASA uses Python>]
```

La fonction [`count\(\)`](#) respecte aussi [`distinct\(\)`](#):

```
>>> Article.objects.filter(publications__title__startswith="Science").count()
2

>>>
Article.objects.filter(publications__title__startswith="Science").distinct().count(
)
1

>>> Article.objects.filter(publications__in=[1,2]).distinct()
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
>>> Article.objects.filter(publications__in=[p1,p2]).distinct()
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
```

Les requêtes plusieurs-à-plusieurs inverses sont possibles (c'est-à-dire des requêtes commençant par la table qui ne possède pas le champ [`ManyToManyField`](#)) :

```
>>> Publication.objects.filter(id=1)
[<Publication: The Python Journal>]
>>> Publication.objects.filter(pk=1)
[<Publication: The Python Journal>]

>>> Publication.objects.filter(article__headline__startswith="NASA")
[<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>, <Publication: The Python Journal>]

>>> Publication.objects.filter(article__id=1)
[<Publication: The Python Journal>]
>>> Publication.objects.filter(article__pk=1)
[<Publication: The Python Journal>]
>>> Publication.objects.filter(article=1)
[<Publication: The Python Journal>]
>>> Publication.objects.filter(article=a1)
[<Publication: The Python Journal>]

>>> Publication.objects.filter(article__in=[1,2]).distinct()
[<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>, <Publication: The Python Journal>]
>>> Publication.objects.filter(article__in=[a1,a2]).distinct()
[<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>, <Publication: The Python Journal>]
```

L'exclusion d'un élément lié fonctionne aussi tel qu'on pourrait le prévoir (même si le code SQL sous-jacent est un peu complexe) :

```
>>> Article.objects.exclude(publications=p2)
[<Article: Django lets you build Web apps easily>]
```

Si nous supprimons une `Publication`, ses `Articles` ne pourront plus y accéder :

```
>>> p1.delete()
>>> Publication.objects.all()
[<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>]
>>> a1 = Article.objects.get(pk=1)
>>> a1.publications.all()
[]
```

Si nous supprimons un `Article`, ses `Publications` ne pourront plus y accéder :

```
>>> a2.delete()
>>> Article.objects.all()
[<Article: Django lets you build Web apps easily>]
>>> p2.article_set.all()
[]
```

Ajout à partir de l'« autre côté » de la relation plusieurs-à-plusieurs :

```
>>> a4 = Article(headline='NASA finds intelligent life on Earth')
>>> a4.save()
>>> p2.article_set.add(a4)
>>> p2.article_set.all()
[<Article: NASA finds intelligent life on Earth>]
>>> a4.publications.all()
[<Publication: Science News>]
```

Ajout à partir de l'autre côté en utilisant des mots-clés :

```
>>> new_article = p2.article_set.create(headline='Oxygen-free diet works wonders')
>>> p2.article_set.all()
[<Article: NASA finds intelligent life on Earth>, <Article: Oxygen-free diet works wonders>]
>>> a5 = p2.article_set.all()[1]
>>> a5.publications.all()
[<Publication: Science News>]
```

Suppression d'une `Publication` à partir d'un `Article`:

```
>>> a4.publications.remove(p2)
>>> p2.article_set.all()
[<Article: Oxygen-free diet works wonders>]
>>> a4.publications.all()
[]
```

Et depuis l'autre côté :

```
>>> p2.article_set.remove(a5)
>>> p2.article_set.all()
[]
```

```
>>> a5.publications.all()
[]
```

Les jeux de relations peuvent recevoir des valeurs. L'attribution de valeurs efface tout membre préexistant du jeu :

```
>>> a4.publications.all()
[<Publication: Science News>]
>>> a4.publications = [p3]
>>> a4.publications.all()
[<Publication: Science Weekly>]
```

Les jeux de relations peuvent être effacés :

```
>>> p2.article_set.clear()
>>> p2.article_set.all()
[]
```

Et l'effacement peut se faire depuis l'autre côté :

```
>>> p2.article_set.add(a4, a5)
>>> p2.article_set.all()
[<Article: NASA finds intelligent life on Earth>, <Article: Oxygen-free diet works wonders>]
>>> a4.publications.all()
[<Publication: Science News>, <Publication: Science Weekly>]
>>> a4.publications.clear()
>>> a4.publications.all()
[]
>>> p2.article_set.all()
[<Article: Oxygen-free diet works wonders>]
```

Recréez l'Article et la Publication que nous avons supprimés :

```
>>> p1 = Publication(title='The Python Journal')
>>> p1.save()
>>> a2 = Article(headline='NASA uses Python')
>>> a2.save()
>>> a2.publications.add(p1, p2, p3)
```

Supprimez en vrac certaines Publications; les références aux publications supprimées ne sont plus présentes :

```
>>> Publication.objects.filter(title__startswith='Science').delete()
>>> Publication.objects.all()
[<Publication: Highlights for Children>, <Publication: The Python Journal>]
>>> Article.objects.all()
[<Article: Django lets you build Web apps easily>, <Article: NASA finds intelligent life on Earth>, <Article: NASA uses Python>, <Article: Oxygen-free diet works wonders>]
>>> a2.publications.all()
[<Publication: The Python Journal>]
```

Supprimez en vrac certains articles ; les références aux objets supprimés ne sont plus présentes :

```
>>> q = Article.objects.filter(headline__startswith='Django')
```

```
>>> print(q)
[<Article: Django lets you build Web apps easily>]
>>> q.delete()
```

Après la suppression avec [`delete\(\)`](#), le cache de [`QuerySet`](#) doit être effacé et les objets référencés ne sont plus présents :

```
>>> print(q)
[]
>>> p1.article_set.all()
[<Article: NASA uses Python>]
```

Une variante de l'appel à [`clear\(\)`](#) est d'attribuer un ensemble vide :

```
>>> p1.article_set = []
>>> p1.article_set.all()
[]

>>> a2.publications = [p1, new_publication]
>>> a2.publications.all()
[<Publication: Highlights for Children>, <Publication: The Python Journal>]
>>> a2.publications = []
>>> a2.publications.all()
[]
```

[Exemples d'utilisation de l'API de relations entre modèles](#)
[Relations plusieurs-à-un](#)

Relations plusieurs-à-un

Pour définir une relation plusieurs-à-un, utilisez [`ForeignKey`](#):

```
from django.db import models

class Reporter(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    email = models.EmailField()

    def __str__(self):
        # __unicode__ on Python 2
        return "%s %s" % (self.first_name, self.last_name)

class Article(models.Model):
    headline = models.CharField(max_length=100)
    pub_date = models.DateField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)

    def __str__(self):
        # __unicode__ on Python 2
        return self.headline

class Meta:
```

```
ordering = ('headline',)
```

Dans ce qui suit, vous trouverez des exemples d'opérations pouvant être effectuées en utilisant les possibilités de l'API Python.

Créez quelques objets Reporter :

```
>>> r = Reporter(first_name='John', last_name='Smith', email='john@example.com')
>>> r.save()

>>> r2 = Reporter(first_name='Paul', last_name='Jones', email='paul@example.com')
>>> r2.save()
```

Créez un objet Article :

```
>>> from datetime import date
>>> a = Article(id=None, headline="This is a test", pub_date=date(2005, 7, 27),
reporter=r)
>>> a.save()

>>> a.reporter.id
1

>>> a.reporter
<Reporter: John Smith>
```

Notez que vous devez enregistrer un objet avant de pouvoir l'attribuer à une relation de clé étrangère. Par exemple, la création d'un `Article` avec une instance `Reporter` non enregistrée produit une erreur `ValueError`:

```
>>> r3 = Reporter(first_name='John', last_name='Smith', email='john@example.com')
>>> Article.objects.create(headline="This is a test", pub_date=date(2005, 7, 27),
reporter=r3)
Traceback (most recent call last):
...
ValueError: save() prohibited to prevent data loss due to unsaved related object
'reporter'.
```

Changed in Django 1.8.4:

Précédemment, l'enregistrement d'un objet avec des objets liés non enregistrés ne produisait pas d'erreur et pouvait aboutir à des pertes de données silencieuses. Dans les versions 1.8-1.8.3, des instances de modèles non enregistrées ne pouvaient plus être attribuées à des champs de liaison, mais cette restriction a été levée pour permettre une utilisation facilitée de modèles uniquement en mémoire.

Les objets `Article` ont accès à leurs objets `Reporter` liés :

```
>>> r = a.reporter
```

Avec Python 2, il s'agit de chaînes de type `str` et non pas de chaînes Unicode car c'est ce qui a été utilisé lors de la création du reporter (et nous n'avons pas actualisé les données depuis la base de données, celle-ci renvoyant toujours des chaînes Unicode) :


```
>>> r.first_name, r.last_name
('John', 'Smith')
```

Créez un objet Article au travers de l'objet Reporter :

```
>>> new_article = r.article_set.create(headline="John's second story",
pub_date=date(2005, 7, 29))
>>> new_article
<Article: John's second story>
>>> new_article.reporter
<Reporter: John Smith>
>>> new_article.reporter.id
1
```

Créez un nouvel article et ajoutez-le au jeu d'articles du reporter :

```
>>> new_article2 = Article(headline="Paul's story", pub_date=date(2006, 1, 17))
>>> r.article_set.add(new_article2)
>>> new_article2.reporter
<Reporter: John Smith>
>>> new_article2.reporter.id
1
>>> r.article_set.all()
[<Article: John's second story>, <Article: Paul's story>, <Article: This is a
test>]
```

Ajoutez le même article à un autre jeu d'articles et vérifiez qu'il a bien été déplacé :

```
>>> r2.article_set.add(new_article2)
>>> new_article2.reporter.id
2
>>> new_article2.reporter
<Reporter: Paul Jones>
```

L'ajout d'un objet du mauvais type lève une exception TypeError :

```
>>> r.article_set.add(r2)
Traceback (most recent call last):
...
TypeError: 'Article' instance expected

>>> r.article_set.all()
[<Article: John's second story>, <Article: This is a test>]
>>> r2.article_set.all()
[<Article: Paul's story>]

>>> r.article_set.count()
2

>>> r2.article_set.count()
1
```

Notez que dans l'exemple précédent, l'article a été déplacé de John à Paul.

Les gestionnaires de relations acceptent aussi les recherches par champ. L'API suit automatiquement les relations à la profondeur nécessaire. Utilisez les doubles soulignements pour séparer les relations. Cela fonctionne à autant de niveaux que nécessaire, sans limite. Par exemple :

```
>>> r.article_set.filter(headline__startswith='This')
[<Article: This is a test>]
```

```
# Find all Articles for any Reporter whose first name is "John".
>>> Article.objects.filter(reporter__first_name='John')
[<Article: John's second story>, <Article: This is a test>]
```

Une correspondance exacte est sous-entendue ici :

```
>>> Article.objects.filter(reporter__first_name='John')
[<Article: John's second story>, <Article: This is a test>]
```

Double requête sur le champ lié. Cela se traduit par une condition AND dans la clause WHERE :

```
>>> Article.objects.filter(reporter__first_name='John',
reporter__last_name='Smith')
[<Article: John's second story>, <Article: This is a test>]
```

Pour la recherche sur la liaison, il est possible d'indiquer une clé primaire ou de transmettre explicitement l'objet lié :

```
>>> Article.objects.filter(reporter__pk=1)
[<Article: John's second story>, <Article: This is a test>]
>>> Article.objects.filter(reporter=1)
[<Article: John's second story>, <Article: This is a test>]
>>> Article.objects.filter(reporter=r)
[<Article: John's second story>, <Article: This is a test>]

>>> Article.objects.filter(reporter__in=[1,2]).distinct()
[<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]
>>> Article.objects.filter(reporter__in=[r,r2]).distinct()
[<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]
```

Vous pouvez aussi utiliser un résultat de requête (queryset) au lieu d'une liste explicite d'instances :

```
>>>
Article.objects.filter(reporter__in=Reporter.objects.filter(first_name='John')).distinct()
[<Article: John's second story>, <Article: This is a test>]
```

Requêtes dans la direction inverse :

```
>>> Reporter.objects.filter(article__pk=1)
[<Reporter: John Smith>]
>>> Reporter.objects.filter(article=1)
[<Reporter: John Smith>]
>>> Reporter.objects.filter(article=a)
[<Reporter: John Smith>]

>>> Reporter.objects.filter(article__headline__startswith='This')
[<Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>]
>>> Reporter.objects.filter(article__headline__startswith='This').distinct()
[<Reporter: John Smith>]
```

Le comptage dans la direction inverse fonctionne en combinant avec distinct() :

```
>>> Reporter.objects.filter(article__headline__startswith='This').count()
3
>>> Reporter.objects.filter(article__headline__startswith='This').distinct().count()
1
```

Les requêtes en boucle sont possibles :

```
>>> Reporter.objects.filter(article__reporter__first_name__startswith='John')
[<Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>]
>>> Reporter.objects.filter(article__reporter__first_name__startswith='John').distinct()
[<Reporter: John Smith>]
>>> Reporter.objects.filter(article__reporter=r).distinct()
[<Reporter: John Smith>]
```

Si vous supprimez un reporter, ses articles seront supprimés (en supposant que la clé étrangère a été créée avec l'attribut [django.db.models.ForeignKey.on_delete](#) défini à CASCADE, ce qui est la valeur par défaut) :

```
>>> Article.objects.all()
[<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]
>>> Reporter.objects.order_by('first_name')
[<Reporter: John Smith>, <Reporter: Paul Jones>]
>>> r2.delete()
>>> Article.objects.all()
[<Article: John's second story>, <Article: This is a test>]
>>> Reporter.objects.order_by('first_name')
[<Reporter: John Smith>]
```

Vous pouvez effectuer une suppression en utilisant une jointure (JOIN) dans la requête :

```
>>> Reporter.objects.filter(article__headline__startswith='This').delete()
>>> Reporter.objects.all()
[]
>>> Article.objects.all()
[]
```

[Relations plusieurs-à-plusieurs](#)

[Relations un-à-un](#)

Relations un-à-un

Pour définir une relation un-à-un, utilisez un champ [OneToOneField](#).

Dans cet exemple, un objet `Place` peut être un `Restaurant` dans certains cas :

```

from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

    def __str__(self):
        # __unicode__ on Python 2
        return "%s the place" % self.name

class Restaurant(models.Model):
    place = models.OneToOneField(
        Place,
        on_delete=models.CASCADE,
        primary_key=True,
    )
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)

    def __str__(self):
        # __unicode__ on Python 2
        return "%s the restaurant" % self.place.name

class Waiter(models.Model):
    restaurant = models.ForeignKey(Restaurant, on_delete=models.CASCADE)
    name = models.CharField(max_length=50)

    def __str__(self):
        # __unicode__ on Python 2
        return "%s the waiter at %s" % (self.name, self.restaurant)

```

Dans ce qui suit, vous trouverez des exemples d'opérations pouvant être effectuées en utilisant les possibilités de l'API Python.

Créez quelques objets **Places**:

```

>>> p1 = Place(name='Demon Dogs', address='944 W. Fullerton')
>>> p1.save()
>>> p2 = Place(name='Ace Hardware', address='1013 N. Ashland')
>>> p2.save()

```

Créez un **Restaurant**. Passez l'identifiant de l'objet « parent » comme clé primaire de cet objet :

```

>>> r = Restaurant(place=p1, serves_hot_dogs=True, serves_pizza=False)
>>> r.save()

```

Un **Restaurant** a accès à son objet **Place**:

```

>>> r.place
<Place: Demon Dogs the place>

```

Un objet **Place** peut accéder à son objet **Restaurant**, s'il existe :

```

>>> p1.restaurant
<Restaurant: Demon Dogs the restaurant>

```

p2 ne possède pas d'objet **Restaurant** associé :

```

>>> from django.core.exceptions import ObjectDoesNotExist
>>> try:

```

```
>>> p2.restaurant
>>> except ObjectDoesNotExist:
>>> print("There is no restaurant here.")
There is no restaurant here.
```

Vous pouvez aussi utiliser `hasattr` pour éviter de devoir intercepter une exception :

```
>>> hasattr(p2, 'restaurant')
False
```

Définissez l'objet `Place` en utilisant la notation d'assignation. Comme `place` est la clé primaire de `Restaurant`, l'enregistrement crée un nouvel objet `Restaurant`:

```
>>> r.place = p2
>>> r.save()
>>> p2.restaurant
<Restaurant: Ace Hardware the restaurant>
>>> r.place
<Place: Ace Hardware the place>
```

Redéfinissez `place` en utilisant l'assignation dans la direction inverse :

```
>>> p1.restaurant = r
>>> p1.restaurant
<Restaurant: Demon Dogs the restaurant>
```

Notez que vous devez enregistrer un objet avant de pouvoir l'attribuer à une relation un-à-un. Par exemple, la création d'un `Restaurant` avec une instance `Place` non enregistrée produit une erreur `ValueError`:

```
>>> p3 = Place(name='Demon Dogs', address='944 W. Fullerton')
>>> Restaurant.objects.create(place=p3, serves_hot_dogs=True, serves_pizza=False)
Traceback (most recent call last):
...
ValueError: save() prohibited to prevent data loss due to unsaved related object
'place'.
```

Changed in Django 1.8.4:

Précédemment, l'enregistrement d'un objet avec des objets liés non enregistrés ne produisait pas d'erreur et pouvait aboutir à des pertes de données silencieuses. Dans les versions 1.8-1.8.3, des instances de modèles non enregistrées ne pouvaient plus être attribuées à des champs de liaison, mais cette restriction a été levée pour permettre une utilisation facilitée de modèles uniquement en mémoire.

`Restaurant.objects.all()` renvoie uniquement les objets `Restaurant`, pas les objets `Places`. Notez qu'il y a maintenant deux restaurants, « Ace Hardware » ayant été créé lors de l'instruction `r.place = p2`:

```
>>> Restaurant.objects.all()
[<Restaurant: Demon Dogs the restaurant>, <Restaurant: Ace Hardware the
restaurant>]
```

`Place.objects.all()` renvoie tous les objets `Place`, qu'ils aient un objet `Restaurant` ou non :

```
>>> Place.objects.order_by('name')
[<Place: Ace Hardware the place>, <Place: Demon Dogs the place>]
```

Vous pouvez interroger les modèles en utilisant les [recherches traversant les relations](#):

```
>>> Restaurant.objects.get(place=p1)
<Restaurant: Demon Dogs the restaurant>
>>> Restaurant.objects.get(place__pk=1)
<Restaurant: Demon Dogs the restaurant>
>>> Restaurant.objects.filter(place__name__startswith="Demon")
[<Restaurant: Demon Dogs the restaurant>]
>>> Restaurant.objects.exclude(place__address__contains="Ashland")
[<Restaurant: Demon Dogs the restaurant>]
```

Ceci fonctionne évidemment aussi dans le sens inverse :

```
>>> Place.objects.get(pk=1)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant__place=p1)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant=r)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant__place__name__startswith="Demon")
<Place: Demon Dogs the place>
```

Ajoutez un objet `Waiter` à un `Restaurant`:

```
>>> w = r.waiter_set.create(name='Joe')
>>> w
<Waiter: Joe the waiter at Demon Dogs the restaurant>
```

Interrogez les objets `Waiter`:

```
>>> Waiter.objects.filter(restaurant__place=p1)
[<Waiter: Joe the waiter at Demon Dogs the restaurant>]
>>> Waiter.objects.filter(restaurant__place__name__startswith="Demon")
[<Waiter: Joe the waiter at Demon Dogs the restaurant>]
```

[Relations plusieurs-à-un](#)
[Gestion des requêtes HTTP](#)

Distribution des URL

Une organisation propre et élégante des URL est un aspect important dans une application Web de qualité. Django vous laisse organiser vos URL comme bon vous semble, sans restriction imposée par le système.

Pas besoin de `.php` ou de `.cgi`, et encore moins de 0, 2097, 1-1-1928, 00.

Lisez [Cool URIs don't change](#), par le père du World Wide Web Tim Berners-Lee, pour savoir pourquoi les URL devraient être claires et conviviales.

Aperçu

Pour organiser les URL d'une application, il s'agit de créer un module Python appelé communément **URLconf** (configuration d'URL). Le code de ce module est en Python pur et est une simple correspondance entre motifs d'URL (de banales expressions régulières) et fonctions Python (vos vues).

Cette correspondance peut être autant courte que longue. Elle peut faire référence à d'autres correspondances. Et étant donné qu'il s'agit de code Python pur, elle peut être construite dynamiquement.

Django fournit également une façon de traduire les URL en fonction de la langue active. Voir la [documentation sur l'internationalisation](#) pour plus d'informations.

Processus de traitement des requêtes par Django

Quand un utilisateur accède à une page de votre site Django, voici l'algorithme que le système suit pour déterminer quel code Python doit être exécuté :

1. Django identifie le module URLconf racine à utiliser. Par défaut, c'est la valeur attribuée au réglage `ROOT_URLCONF`, mais si la requête `HttpRequest` entrante possède un attribut `urlconf` (défini par l'intergiciel de [traitement de requête](#)), sa valeur sera utilisée en lieu et place du réglage `ROOT_URLCONF`.
2. Django charge ce module Python et cherche la variable `urlpatterns`. Ce devrait être une liste Python d'instances `django.conf.urls.url()`.
3. Django parcourt chaque motif d'URL dans l'ordre et s'arrête dès la première correspondance avec l'URL demandée.
4. Une fois qu'une des expressions régulières correspond, Django importe et appelle la vue correspondante, qui est une simple fonction Python (ou une [vue reposant sur une classe](#)). La vue se voit passer les paramètres suivants :
 - Une instance `HttpRequest`.
 - Si l'expression régulière correspondante n'a pas renvoyé de groupes nommés, les correspondances de l'expression régulière sont transmises comme paramètres positionnels.

- Les paramètres nommés sont formés des groupes nommés résultant de la correspondance par expression régulière, surchargés par tout paramètre ajouté dans le paramètre facultatif `kwargs` de [django.conf.urls.url\(\)](#).
5. Si aucune expression régulière ne correspond, ou si une exception est levée durant ce processus, Django appelle une vue d'erreur appropriée. Voir [Gestion d'erreur](#) plus bas.

Exemple

Voici un exemple d'URLconf :

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/([0-9]{4})/$', views.year_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.article_detail),
]
```

Notes :

- Pour capturer une valeur contenue dans l'URL, entourez-la simplement de parenthèses.
- Inutile de préfixer vos URL d'une barre oblique, elles en ont toutes une. Par exemple, écrivez `^articles` et non `^/articles`.
- Le 'r' devant chaque expression régulière est facultatif mais recommandé. Il signale à Python qu'une chaîne est « brute », aucun caractère de la chaîne ne doit être échappé. Voir [l'explication de Dive Into Python](#).

Exemples de requêtes :

- Une requête vers `/articles/2005/03/` correspondrait à la troisième entrée dans la liste. Django appellerait la fonction `views.month_archive(request, '2005', '03')`.
- `/articles/2005/3/` ne correspondrait à aucun motif d'URL, car la troisième entrée dans la liste nécessite deux chiffres pour le mois.
- `/articles/2003/` correspondrait au premier motif de la liste, et non le deuxième, car les motifs sont évalués dans l'ordre, et le premier est le premier à correspondre. Libre à vous d'utiliser l'ordre de définition pour traiter des cas spéciaux comme ici. Ici, Django appellerait la fonction `views.special_case_2003(request)`.
- `/articles/2003` ne correspondrait à aucun motif, car chaque motif nécessite que l'URL se termine par une barre oblique.

- `/articles/2003/03/03/` correspondrait au dernier motif. Django appellerait la fonction `views.article_detail(request, '2003', '03', '03')`.

Groupes nommés

L'exemple ci-dessus utilise de simples groupes d'expressions régulières *non nommés* (via les parenthèses) pour capturer des bouts d'URL et les passer en tant que paramètres *positionnels* à une vue. Dans des situations plus poussées, il est possible d'utiliser des groupes de capture *nommés* pour capturer des bouts d'URL et les passer en tant que paramètres *nommés* à une vue.

La syntaxe des expressions régulières en Python pour les groupes de captures nommés est `(?P<nom>motif)`, où `nom` est le nom du groupe et `motif` est le motif à faire correspondre.

Voici le même exemple d'URLconf que ci-dessus, réécrit en utilisant les groupes nommés :

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$',
views.month_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<day>[0-9]{2})/$',
views.article_detail),
]
```

Le résultat est strictement identique qu'avec le précédent exemple, avec une subtile différence : les valeurs capturées sont passées aux vues en tant que paramètres nommés plutôt qu'en tant que paramètres positionnels. Par exemple :

- Une requête sur `/articles/2005/03/` appellerait la fonction `views.month_archive(request, year='2005', month='03')`, plutôt que `views.month_archive(request, '2005', '03')`.
- Une requête sur `/articles/2003/03/03/` appellerait la fonction `views.article_detail(request, year='2003', month='03', day='03')`.

En pratique, cela signifie que vos modules URLconf sont légèrement plus explicites et moins sujets aux bogues d'ordre des paramètres, et vous pouvez réordonner les paramètres des définitions de vos vues. Bien sûr, ces avantages viennent au prix de la brièveté ; certains développeurs trouvent la syntaxe des groupes nommés vilaine et trop verbeuse.

L'algorithme de correspondance/groupement

Voici l'algorithme que suit l'analyseur d'URLconf, en faisant la distinction entre les groupes nommés et les groupes non nommés dans une expression régulière :

1. S'il y a des paramètres nommés, il les utilise et ignore les paramètres anonymes.
2. Sinon, il passe tous les paramètres anonymes en tant que paramètres positionnels.

Dans les deux cas, tout paramètre nommé supplémentaire passé comme décrit dans [Transmission de paramètres supplémentaires à une vue](#) (plus bas) sera aussi passé à la vue.

Configurations d'URL et contenu des comparaisons

L'URLconf utilise l'URL demandée, en tant que simple chaîne Python. Ni les paramètres GET ou POST, ni le nom de domaine ne sont inclus.

Par exemple, pour une requête sur `https://www.example.com/myapp/`, l'URLconf cherchera `myapp/`.

Pour une requête sur `https://www.example.com/myapp/?page=3`, l'URLconf cherchera `myapp/`.

L'URLconf ignore la méthode de la requête. En d'autres termes, toutes les méthodes de requêtes – POST, GET, HEAD, etc. – seront acheminées vers la même fonction de la même URL.

Les paramètres capturés sont toujours des chaînes de caractères

Chaque paramètre capturé est envoyé à la vue en tant que simple chaîne de caractères Python, peu importe la correspondance qu'effectue l'expression régulière. Par exemple, dans cette ligne de configuration d'URL :

```
url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
```

... le paramètre `year` transmis à `views.year_archive()` sera une chaîne de caractères,

et non pas un entier, même si `[0-9]{4}` ne correspondra qu'à des chaînes représentant des entiers.

Valeurs par défaut des paramètres de vue

Une astuce pratique est de définir des valeurs par défaut pour les paramètres de vos vues. Voici un exemple de configuration d'URL et de vue :

```
# URLconf
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^blog/$', views.page),
    url(r'^blog/page(?P<num>[0-9]+)/$', views.page),
]
```

```
# View (in blog/views.py)
def page(request, num="1"):
    # Output the appropriate page of blog entries, according to num.
    ...
```

Dans l'exemple ci-dessus, les deux motifs d'URLs pointe vers la même vue – `views.page` – mais le premier motif ne capture aucun élément de l'URL. Si le premier motif correspond, la fonction `page()` utilisera la valeur par défaut de son paramètre `num`, "1". Si le second motif correspond, `page()` utilisera la valeur capturée par l'expression régulière.

Performance

Chaque expression régulière d'un `urlpatterns` est compilée la première fois qu'elle est utilisée. Cela rend le système extrêmement rapide.

Syntaxe de la variable `urlpatterns`

`urlpatterns` devrait être une liste Python d'instances [`url\(\)`](#).

Gestion d'erreur

Quand Django ne trouve pas d'expression régulière correspondant à l'URL demandée, ou quand une exception est générée, Django appelle une vue de gestion d'erreur.

Les vues à utiliser pour ces situations sont définies par quatre variables. Leur valeur par défaut devraient convenir à la plupart des projets, mais il reste possible de les personnaliser en surchargeant leurs valeurs par défaut.

Voir la documentation sur [la personnalisation des vues d'erreur](#) pour plus de détails.

Ces valeurs doivent être définies dans l'`URLCONF` racine. La définition de ces variables dans n'importe quel autre `URLCONF` n'aura aucun effet.

Les valeurs doivent pouvoir être appelées, ou être des chaînes de caractères représentant le chemin d'importation Python complet vers une vue qui doit être appelée pour traiter l'erreur actuelle.

Les variables sont :

- `handler400` – Voir [django.conf.urls.handler400](#).
- `handler403` – Voir [django.conf.urls.handler403](#).
- `handler404` – Voir [django.conf.urls.handler404](#).
- `handler500` – Voir [django.conf.urls.handler500](#).

Inclusion d'autres URLconfs

À tout moment, votre `urlpatterns` peut « inclure » d'autres modules `URLconf`.

Fondamentalement, cela permet de rassembler un ensemble d'URL les unes sous les autres.

Par exemple, voici un extrait de la configuration d'URL du [site Web Django](#) lui-même. Il inclut un certain nombre d'autres configurations d'URL :

```
from django.conf.urls import include, url

urlpatterns = [
    # ... snip ...
    url(r'^community/', include('django_website.aggregator.urls')),
    url(r'^contact/', include('django_website.contact.urls')),
    # ... snip ...
]
```

Remarquez que les expressions régulières de cet exemple n'ont pas de \$ (caractère de fin de correspondance) mais ont bien une barre oblique finale. Quand Django rencontre un `include()` ([`django.conf.urls.include\(\)`](#)), il tronque le bout d'URL qui correspondait jusque là et passe la chaîne de caractères restante à la configuration d'URL incluse pour continuer le traitement.

Une autre possibilité est d'inclure des motifs d'URL supplémentaires en utilisant une liste d'instances [`url\(\)`](#). Par exemple, dans cet configuration d'URL :

```
from django.conf.urls import include, url

from apps.main import views as main_views
from credit import views as credit_views

extra_patterns = [
    url(r'^reports/$', credit_views.report),
    url(r'^reports/(?P<id>[0-9]+)/$', credit_views.report),
    url(r'^charge/$', credit_views.charge),
]

urlpatterns = [
    url(r'^$', main_views.homepage),
    url(r'^help/', include('apps.help.urls')),
    url(r'^credit/', include(extra_patterns)),
]
```

Dans cet exemple, l'URL `/credit/reports/` sera traitée par la vue Django `credit_views.report()`.

Cela peut être utilisé pour supprimer de la redondance dans des configurations d'URL où un préfixe de motif est employé de manière répétitive. Par exemple, considérez cette configuration d'URL :

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^(?P<page_slug>[w-]+)-(?P<page_id>w+)/history/$', views.history),
    url(r'^(?P<page_slug>[w-]+)-(?P<page_id>w+)/edit/$', views.edit),
]
```

```

        url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/discuss/$', views.discuss),
        url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/permissions/$',
views.permissions),
]

```

Nous pouvons l'améliorer en indiquant une seule fois le préfixe de chemin commun et en regroupant les suffixes qui diffèrent :

```

from django.conf.urls import include, url
from . import views

urlpatterns = [
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/', include([
        url(r'^history/$', views.history),
        url(r'^edit/$', views.edit),
        url(r'^discuss/$', views.discuss),
        url(r'^permissions/$', views.permissions),
    ])),
]

```

Les paramètres capturés

Un URLConf inclus reçoit tous les paramètres capturés par ses URLConf parents, ainsi l'exemple suivant est valide :

```

# In settings/urls/main.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
]

# In foo/urls/blog.py
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.blog.index),
    url(r'^archive/$', views.blog.archive),
]

```

Dans l'exemple ci-dessus, la variable capturée "username" est passée à l'URLConf inclus, comme on peut s'y attendre.

Paramètres imbriqués

Les expressions régulières autorisent les paramètres imbriqués et Django les résout et les transmet à la vue. Lors de l'inversion d'URL, Django essaie de remplir tous les paramètres capturés externes, en ignorant les paramètres capturés imbriqués. Considérez les motifs d'URL suivants qui acceptent de manière facultative un paramètre page:

```

from django.conf.urls import url

```

```
urlpatterns = [
    url(r'blog/(page-(\d+)/)?$', blog_articles),      # bad
    url(r'comments/(?P<page-(?P<page_number>\d+)/)?$', comments), # good
]
```

Les deux motifs emploient des paramètres imbriqués et seront résolus : par exemple, `blog/page-2/` va correspondre à `blog_articles` avec deux paramètres positionnels : `page-2/` et `2`. Le second motif pour `comments` va correspondre à `comments/page-2/` avec le paramètre nommé `page_number` défini à `2`. Le paramètre extérieur dans ce cas est un paramètre non capturant (`?:...`).

La vue `blog_articles` a besoin du paramètre capturé extérieur afin d'être résolue, `page-2/` ou aucun paramètre, dans ce cas, alors que `comments` peut être résolue avec soit aucun paramètre, soit une valeur pour `page_number`.

Les paramètres capturés imbriqués créent un couplage fort entre les paramètres de la vue et l'URL, comme le démontre `blog_articles`: la vue reçoit une partie de l'URL (`page-2/`) au lieu de ne recevoir que la valeur qui l'intéresse réellement. Ce couplage est encore plus prononcé lors de la résolution, car pour résoudre la vue, il est nécessaire de transmettre un bout d'URL au lieu d'un simple numéro de page.

En règle générale, capturez uniquement les valeurs que la vue doit gérer et employez des paramètres sans capture lorsque l'expression régulière a besoin d'un paramètre mais que la vue l'ignore.

Transmission de paramètres supplémentaires à une vue

Les configurations d'URL ont un point d'entrée qui permet de passer des paramètres supplémentaires à vos vues, via un dictionnaire Python.

La fonction [django.conf.urls.url\(\)](#) accepte un troisième paramètre facultatif qui doit correspondre à un dictionnaire de paramètres nommés supplémentaires à transmettre à la fonction de vue.

Par exemple :

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^blog/(?P<year>[0-9]{4})/$', views.year_archive, {'foo': 'bar'}),
]
```

Dans cet exemple, si une requête demande `/blog/2005/`, Django appelle `views.year_archive(request, year='2005', foo='bar')`.

Cette technique est utilisée dans le [système de syndication](#) pour passer des métadonnées et des options aux vues.

Gestion des conflits

Il est possible d'avoir un motif d'URL qui capture des paramètres nommés, et qui passe aussi des paramètres ayant les mêmes noms dans son dictionnaire de paramètres supplémentaires. Quand c'est le cas, les paramètres du dictionnaire seront utilisés à la place des paramètres capturés dans l'URL.

Transmission de paramètres supplémentaires à `include()`

De la même façon, vous pouvez passer des paramètres supplémentaires à la fonction `include()`. Dans ce cas, *chacune* des lignes de la configuration d'URL incluse recevra les paramètres supplémentaires.

Par exemple, ces deux ensembles de configurations d'URL sont fonctionnellement identiques :

Premier ensemble :

```
# main.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^blog/', include('inner'), {'blogid': 3}),
]

# inner.py
from django.conf.urls import url
from mysite import views

urlpatterns = [
    url(r'^archive/$', views.archive),
    url(r'^about/$', views.about),
]
```

Second ensemble :

```
# main.py
from django.conf.urls import include, url
from mysite import views

urlpatterns = [
    url(r'^blog/', include('inner')),
]

# inner.py
from django.conf.urls import url

urlpatterns = [
    url(r'^archive/$', views.archive, {'blogid': 3}),
    url(r'^about/$', views.about, {'blogid': 3}),
]
```

Remarquez que les paramètres supplémentaires seront *toujours* passés à *chaque* élément de la configuration d'URL incluse, peu importe si la vue associée à l'élément accepte ces paramètres comme étant valides. C'est pour cette raison que cette technique est utile seulement lorsque vous êtes certain

que toutes les vues de la configuration d'URL incluse acceptent les paramètres supplémentaires que vous passez.

Résolution inversée des URL

Un besoin courant quand on travaille sur un projet Django est d'avoir la possibilité d'obtenir des URL dans leur forme finale soit pour les incorporer dans du contenu généré (des URL vers des vues et des ressources, des URL montrées à l'utilisateur, etc.) ou pour gérer le flux de navigation côté serveur (redirections, etc.).

Il est très fortement suggéré d'éviter d'écrire des URL figées (une approche laborieuse, non extensible et sujette à l'erreur). Il est tout aussi dangereux de concevoir des mécanismes ad hoc capables de générer des URL identiques à celles décrites dans la configuration d'URL, risquant d'aboutir avec le temps à l'obtention d'URL obsolètes.

En d'autres termes, ce dont nous avons besoin est un mécanisme DRY (« ne pas se répéter »). L'un de ses avantages est de permettre l'évolution de l'organisation des URL sans avoir à parcourir le code source du projet pour remplacer les URL obsolètes.

L'information principale dont nous disposons pour récupérer une URL est une identification de la vue qui est chargée de la traiter (par exemple, son nom). Les autres informations nécessaires à la recherche de la bonne URL sont les types (positionnel, nommé) et les valeurs des paramètres de la vue.

Django met à disposition une solution qui fait en sorte que le système d'abstraction d'URL est le seul à connaître l'organisation des URL. Vous lui donnez votre configuration d'URL et vous pouvez ensuite l'utiliser dans les deux sens :

- À partir d'une URL demandée par un utilisateur/navigateur, il appelle la vue Django concernée en lui passant tous les paramètres dont elle a besoin en extrayant les valeurs telles quelles de l'URL.
- À partir de l'identification de la vue Django concernée ainsi que les valeurs des paramètres qui lui seraient passés, il renvoie l'URL associée.

Le premier correspond à l'utilisation dont nous avons parlé dans les sections précédentes. Le second correspond à ce qu'on appelle *la résolution inversée des URL*, *la correspondance inverse d'URL*, *la recherche inverse d'URL*, ou plus simplement *l'inversion d'URL*.

Django met à disposition des outils pour réaliser une inversion d'URL dans les différentes couches où les URL sont nécessaires :

- Dans les gabarits : en utilisant la balise de gabarit [url](#).
- Dans le code Python : en utilisant la fonction [django.core.urlresolvers.reverse\(\)](#).

- Dans le code plus abstrait amené à manipuler les URL des instances de modèle Django : la méthode `get_absolute_url()`.

Exemples

Reprenons comme exemple cette ligne de configuration d'URL :

```
from django.conf.urls import url

from . import views

urlpatterns = [
    #...
    url(r'^articles/([0-9]{4})/$', views.year_archive, name='news-year-archive'),
    #...
]
```

D'après ce schéma, l'URL de l'archive correspondant à l'année *nnnn* est `/articles/nnnn/`.

Vous pouvez obtenir ces URL dans les gabarits en utilisant :

```
<a href="{% url 'news-year-archive' 2012 %}">2012 Archive</a>
{# Or with the year in a template context variable: #}
<ul>
{% for yearvar in year_list %}
<li><a href="{% url 'news-year-archive' yearvar %}">{{ yearvar }} Archive</a></li>
{% endfor %}
</ul>
```

Ou en Python :

```
from django.core.urlresolvers import reverse
from django.http import HttpResponseRedirect

def redirect_to_year(request):
    # ...
    year = 2006
    # ...
    return HttpResponseRedirect(reverse('news-year-archive', args=(year,)))
```

Si, pour quelque raison que ce soit, vous décidez de changer l'URL à laquelle les archives annuelles d'articles sont sauvegardées, vous aurez seulement besoin de changer la ligne concernée dans la configuration d'URL.

Dans le cas où les vues ont un comportement générique, une relation N:1 peut exister entre les URL et les vues. Dans ces cas-là, le nom de la vue n'est pas un identifiant suffisant au moment où il s'agit d'utiliser l'inversion d'URL. Lisez la section suivante pour découvrir la solution qu'apporte Django à ce problème.

Nommage des motifs d'URL

Afin de pouvoir effectuer des inversions d'URL, il est nécessaire d'utiliser des **motifs d'URL nommés** comme vous pouvez le voir dans les exemples ci-dessus. La chaîne utilisée pour les noms des URL peut contenir tous les caractères que vous voulez. Vous n'êtes pas tenus d'utiliser des noms Python valables.

Lorsque vous nommez vos motifs d'URL, assurez-vous d'utiliser des noms qui ont peu de chances d'entrer en conflit avec ceux d'une autre application. Si vous appelez votre motif d'URL `comment`, et qu'une autre application fait la même chose, il n'y a aucune garantie que la bonne URL sera insérée dans votre gabarit quand vous utilisez ce nom.

L'ajout d'un préfixe à vos noms d'URL, peut-être dérivé du nom de l'application, permet de réduire les risques de collision. Nous recommandons quelque chose comme `myapp-comment` plutôt que `comment`.

Les espaces de noms d'URL

Introduction

Les espaces de noms d'URL permettent la résolution inverse de [motifs d'URL nommés](#) de manière unique, même si différentes applications utilisent les mêmes noms d'URL. L'utilisation systématique d'URL avec espaces de noms est une bonne pratique que les applications tierces devraient adopter (comme cela a été fait dans le tutoriel). C'est aussi un moyen qui permet la résolution inverse quand plusieurs instances d'une application sont déployées. En d'autres termes, comme plusieurs instances d'une même application partagent les URL nommées, les espaces de noms fournissent un moyen de distinguer ces URL nommées.

Django applications that make proper use of URL namespacing can be deployed more than once for a particular site. For example [django.contrib.admin](#) has an [AdminSite](#) class which allows you to easily [deploy more than one instance of the admin](#). In a later example, we'll discuss the idea of deploying the polls application from the tutorial in two different locations so we can serve the same functionality to two different audiences (authors and publishers).

Un espace de noms d'URL se compose de deux parties, qui sont les deux des chaînes de caractères :

espace de noms d'application

Ceci définit le nom de l'application déployée. Chaque instance d'une même application possède le même espace de noms d'application. Par exemple, l'application d'administration de Django possède l'espace de noms d'application `'admin'`, comme l'on peut s'y attendre.

espace de noms d'instance

Ceci identifie une instance particulière d'une application. Les espaces de noms d'instance doivent être uniques dans tout un projet. Cependant, un espace de noms d'instance peut être identique à l'espace de noms d'application. C'est utilisé pour identifier l'instance par défaut d'une application. Par exemple, l'instance par défaut du site d'administration de Django possède l'espace de noms d'instance 'admin'.

Les URL avec espace de noms sont identifiées par l'opérateur ': '. Par exemple, la page d'accueil principale de l'application d'administration est référencée par 'admin:index'. Cela indique une URL nommée 'index' dans un espace de noms 'admin'.

Les espaces de noms peuvent aussi être imbriqués. L'URL nommée 'sports:polls:index' recherche un motif nommé 'index' dans l'espace de noms 'polls', lui-même défini à l'intérieur de l'espace de noms de premier niveau 'sports'.

Résolution inverse des URL avec espaces de noms

Lorsqu'il reçoit une URL avec espace de noms à résoudre (par ex. 'polls:index'), Django partage le nom complet en segments et effectue la recherche suivante :

1. Premièrement, Django cherche un [espace de noms d'application](#) correspondant (dans cet exemple, 'polls'). Cela produira une liste d'instances de cette application.
2. Si une application courante est définie, Django trouve et renvoie le résolveur d'URL de cette instance. L'application courante peut être définie avec le paramètre `current_app` de la fonction [reverse\(\)](#).

La balise de gabarit [url](#) utilise l'espace de noms de la vue actuellement résolue comme application courante dans un [RequestContext](#). Vous pouvez outrepasser cette valeur par défaut en définissant l'application courante dans l'attribut [request.current_app](#).

Changed in Django 1.8:

Dans les versions précédentes de Django, il fallait définir l'attribut `current_app` pour tout objet [Context](#) ou [RequestContext](#) utilisé pour faire le rendu d'un gabarit.

Changed in Django 1.9:

Précédemment, la balise de gabarit [url](#) n'utilisait pas l'espace de nom de la vue en cours et il fallait définir l'attribut `current_app` de la requête.

3. S'il n'existe pas d'application courante, Django cherche une instance d'application par défaut. Celle-ci correspond à l'instance dont l'[espace de noms d'instance](#) est identique à l'[espace de noms d'application](#) (dans cet exemple, une instance de `polls` appelée 'polls').
4. S'il n'existe pas d'instance d'application par défaut, Django sélectionne la dernière instance déployée de l'application, quel que soit son nom d'instance.

5. Si l'espace de noms indiqué ne correspond pas à un [espace de noms d'application](#) à l'étape 1, Django effectue une recherche directe d'espace de noms en tant qu'[espace de noms d'instance](#).

S'il existe des espaces de noms imbriqués, ces étapes sont répétées pour chaque partie de l'espace de noms jusqu'à ce qu'il ne reste plus que le nom de la vue comme élément non résolu. Le nom de vue est ensuite résolu comme URL dans l'espace de noms qui a été trouvé.

Exemple

Pour voir en action cette stratégie de résolution, considérez l'exemple de deux instances de l'application `polls` du tutoriel : l'une appelée `'author-polls'` et l'autre `'publisher-polls'`. Supposons que l'on ait amélioré cette application afin qu'elle prenne en considération l'espace de noms de l'instance lors de la création et de l'affichage des sondages.

`urls.py`

```
from django.conf.urls import include, url

urlpatterns = [
    url(r'^author-polls/', include('polls.urls', namespace='author-polls')),
    url(r'^publisher-polls/', include('polls.urls', namespace='publisher-polls')),
]
```

`polls/urls.py`

```
from django.conf.urls import url

from . import views

app_name = 'polls'
urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='detail'),
    ...
]
```

Avec cette configuration, les recherches suivantes sont possibles :

- Si l'une des instances est courante, c'est-à-dire que nous construisons la page de détail dans l'instance `'author-polls'`, `'polls:index'` est résolu à la page d'accueil de l'instance `'author-polls'`. C'est-à-dire que les deux expressions suivantes produiront `"/author-polls/`.

Dans la méthode d'une vue fondée sur les classes :

```
reverse('polls:index', current_app=self.request.resolver_match.namespace)
```

et dans le gabarit :

```
{% url 'polls:index' %}
```

- S'il n'existe pas d'instance courante, par exemple si nous produisons une page quelque part ailleurs sur le site, `'polls:index'` est résolu en rapport avec l'instance `polls` inscrite en dernier. Comme il n'y a pas d'instance par défaut (espace de noms d'instance `'polls'`), c'est la dernière instance inscrite de `polls` qui est utilisée. Cela sera `'publisher-polls'` car c'est celle qui est déclarée en dernier dans les motifs d'URL `urlpatterns`.
- `'author-polls:index'` est toujours résolu comme page d'accueil de l'instance `'author-polls'` (et de même pour `'publisher-polls'`).

S'il y avait aussi une instance par défaut, nommée `'polls'`, la seule chose qui change par rapport à ci-dessus est que quand il n'y a pas d'instance courante (le deuxième cas ci-dessus), `'polls:index'` est résolu à la page d'accueil de l'instance par défaut, et non pas de l'instance déclarée en dernier dans les motifs d'URL `urlpatterns`.

Espaces de noms d'URL et configurations d'URL incluses

Les espaces de noms d'application des configurations d'URL incluses peuvent être définis de deux manières.

Premièrement, vous pouvez définir un attribut `app_name` dans le module de configuration d'URL inclus, au même niveau que l'attribut `urlpatterns`. Il est nécessaire d'indiquer à [`include\(\)`](#) le module réel ou une référence textuelle au module, non pas la liste ou l'élément `urlpatterns` lui-même.

`polls/urls.py`

```
from django.conf.urls import url

from . import views

app_name = 'polls'
urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='detail'),
    ...
]
```

`urls.py`

```
from django.conf.urls import include, url

urlpatterns = [
    url(r'^polls/', include('polls.urls')),
]
```

Les URL définies dans `polls.urls` auront un espace de noms d'application `polls`.

Deuxièmement, vous pouvez inclure un objet contenant des données d'espace de noms intégrées. Si vous incluez par `include()` une liste d'instances [`url\(\)`](#), les URL contenues dans cet objet seront

ajoutées à l'espace de noms global. Cependant, vous pouvez aussi inclure un tuple de 2 éléments contenant :

```
(<list of url() instances>, <application namespace>)
```

Par exemple :

```
from django.conf.urls import include, url

from . import views

polls_patterns = ([
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='detail'),
], 'polls')

urlpatterns = [
    url(r'^polls/', include(polls_patterns)),
]
```

Cela va inclure les motifs d'URL nommés dans l'espace de noms d'application donné.

L'espace de noms d'instance peut être précisé en utilisant le paramètre `namespace` de [include\(\)](#). Si l'espace de noms d'instance n'est pas défini, il contiendra par défaut l'espace de noms d'application de la configuration d'URL incluse. Cela signifie qu'il s'agira aussi de l'instance par défaut pour cet espace de noms.

Changed in Django 1.9:

Dans les versions précédentes, il fallait indiquer à la fois l'espace de noms d'application et l'espace de noms d'instance à un seul endroit, soit en les passant comme paramètres à [include\(\)](#) ou en incluant un tuple de 3 éléments contenant (<liste d'instances d'url()>, <espace de noms d'application>, <espace de noms d'instance>).

[Gestion des requêtes HTTP](#)

[Écriture de vues](#)

Écriture de vues

Une fonction de vue, ou *vue* pour faire simple, est une simple fonction Python acceptant une requête Web et renvoyant une réponse Web. Cette réponse peut contenir le contenu HTML d'une page Web, une redirection, une erreur 404, un document XML, une image... ou vraiment n'importe quoi d'autre. La vue elle-même contient la logique nécessaire pour renvoyer une réponse. Ce code peut se trouver à l'emplacement de votre choix, pour autant qu'il soit dans le chemin Python. Il n'y a pas d'autres exigences, pas de « magie » comme on dit. Mais comme il faut bien mettre ce code *quelque part*, la

convention est de placer les vues dans un fichier nommé `views.py` se trouvant dans un projet ou un répertoire d'application.

Une vue simple

Voici une vue qui renvoie l'heure et la date actuelle, sous forme d'un document HTML :

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Passons ce code en revue ligne par ligne :

- Nous importons d'abord la classe [HttpResponse](#) à partir du module [django.http](#), ainsi que la bibliothèque `datetime` de Python.
- Puis, nous définissons une fonction nommée `current_datetime`. C'est la fonction de vue. Chaque fonction de vue accepte un objet [HttpRequest](#) comme premier paramètre, typiquement nommé `request`.

Remarquez que le nom de la fonction de vue n'a aucune importance ; la vue ne doit pas être nommée d'une certaine manière pour que Django puisse la reconnaître. Nous l'avons appelée `current_datetime` ici car ce nom indique clairement ce qu'elle fait.

- La vue renvoie un objet [HttpResponse](#) contenant la réponse générée. Chaque fonction de vue est chargée de renvoyer un objet [HttpResponse](#) (il y a des exceptions, mais nous les aborderons plus tard).

Le fuseau horaire de Django

Django contient un réglage [TIME_ZONE](#) (fuseau horaire) valant par défaut `America/Chicago`. Vous ne vivez probablement pas à cet endroit, vous allez donc la modifier dans votre fichier de réglages.

Correspondance entre URL et vues

Ainsi, pour résumer, cette fonction de vue renvoie une page HTML contenant la date et l'heure actuelles. Pour afficher cette vue avec une URL particulière, il est nécessaire de créer une *configuration d'URL*. Voir [Distribution des URL](#) pour les instructions.

Renvoi d'erreurs

Le renvoi de codes d'erreur HTTP est simple dans Django. Il existe des sous-classes de [HttpResponse](#) pour tous les codes de statut HTTP les plus courants autres que 200 (qui signifie « OK »). La liste complète des sous-classes disponibles se trouve dans la documentation des [requêtes/réponses](#). Il suffit de renvoyer une instance de l'une de ces sous-classes au lieu d'une réponse [HttpResponse](#) normale afin de signaler une erreur. Par exemple :

```
from django.http import HttpResponse, HttpResponseNotFound

def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</h1>')
```

Il n'existe pas de sous-classe spécialisée pour chaque code de réponse HTTP possible, car la plupart ne se rencontrent que très rarement. Cependant, comme décrit dans la documentation de [HttpResponse](#), il est aussi possible de transmettre le code de statut HTTP au constructeur de [HttpResponse](#) pour créer une classe à renvoyer pour n'importe quel code de statut. Par exemple :

```
from django.http import HttpResponse

def my_view(request):
    # ...

    # Return a "created" (201) response code.
    return HttpResponse(status=201)
```

Comme les erreurs 404 sont de loin les erreurs HTTP les plus fréquentes, il existe une façon simplifiée de les traiter.

L'exception `Http404`

```
class django.http.Http404
```

Lorsque vous renvoyez une erreur telle que [HttpResponseNotFound](#), vous êtes chargé de définir le code HTML de la page d'erreur résultante :

```
return HttpResponseNotFound('<h1>Page not found</h1>')
```

Par commodité et parce qu'il est conseillé d'avoir une page d'erreur 404 cohérente pour tout un site, Django fournit une exception `Http404`. Si vous générez `Http404` quelque part dans une fonction de vue, Django l'intercepte et renvoie la page d'erreur standard de l'application, en compagnie du code d'erreur HTTP 404.

Exemple d'utilisation :

```
from django.http import Http404
```



```
from django.shortcuts import render
from polls.models import Poll

def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist")
    return render(request, 'polls/detail.html', {'poll': p})
```

Afin de pouvoir afficher du HTML personnalisé lorsque Django répond avec une page 404, il est possible de créer un gabarit HTML nommé `404.html` et de le placer au premier niveau de l'arborescence des gabarits. Ce gabarit sera alors utilisé lorsque le réglage [DEBUG](#) est défini à `False`.

Lorsque [DEBUG](#) vaut `True`, vous pouvez fournir un message à `Http404` et il apparaîtra dans le gabarit 404 standard de débogage. Utilisez ces messages à des fins de débogage ; ils ne sont généralement pas adaptés aux gabarits 404 de production.

Personnalisation des vues d'erreur

Les vues d'erreur par défaut de Django conviennent bien à la majorité des applications Web, mais elles peuvent aisément être surchargées en cas de besoin particulier. Il suffit de redéfinir les gestionnaires correspondants comme démontré ci-dessous, dans la configuration d'URL racine (leur définition à tout autre endroit est sans effet).

La vue [page_not_found\(\)](#) est surchargée par [handler404](#):

```
handler404 = 'mysite.views.my_custom_page_not_found_view'
```

La vue [server_error\(\)](#) est surchargée par [handler500](#):

```
handler500 = 'mysite.views.my_custom_error_view'
```

La vue [permission_denied\(\)](#) est surchargée par [handler403](#):

```
handler403 = 'mysite.views.my_custom_permission_denied_view'
```

La vue [bad_request\(\)](#) est surchargée par [handler400](#):

```
handler400 = 'mysite.views.my_custom_bad_request_view'
```

[Distribution des URL](#)

[Décorateurs de vue](#)

Décorateurs de vue

Django fournit plusieurs décorateurs afin de permettre aux vues de prendre en charge différentes fonctionnalités HTTP.

Méthodes HTTP autorisées

Les décorateurs présents dans [django.views.decorators.http](#) peuvent être utilisés pour restreindre l'accès à une vue en se basant sur la méthode utilisée lors de la requête. Ces décorateurs renvoient une instance de [django.http.HttpResponseNotAllowed](#) si les conditions ne sont pas remplies.

`require_http_methods(request_method_list)`[\[source\]](#)

Décorateur exigeant que l'accès à une vue se fasse via certaines méthodes HTTP seulement.
Exemple :

```
from django.views.decorators.http import require_http_methods

@require_http_methods(["GET", "POST"])
def my_view(request):
    # I can assume now that only GET or POST requests make it this far
    # ...
    pass
```

Remarquez que les noms de méthodes HTTP doivent être en majuscules.

`require_GET()`

Décorateur exigeant que la vue n'accepte que la méthode GET.

`require_POST()`

Décorateur exigeant que la vue n'accepte que la méthode POST.

`require_safe()`

Décorateur exigeant que la vue n'accepte que les méthodes GET et HEAD. Ces méthodes sont généralement considérées « sans risque » car elles ne devraient rien faire d'autre mis à part retourner la ressource demandée.

Note

Django retire automatiquement le contenu des réponses aux requêtes HEAD en prenant soin de laisser les en-têtes inchangés, vous pouvez ainsi traiter les requêtes HEAD comme des requêtes GET dans vos vues. Étant donné que certains programmes, tels que les vérificateurs d'hyperliens,

se basent sur les requêtes HEAD, il peut être préférable d'utiliser `require_safe` à la place de `require_GET`.

Traitement conditionnel de vue

Les décorateurs suivants présents dans [django.views.decorators.http](#) peuvent être utilisés pour contrôler le comportement du cache sur certaines vues.

`condition(etag_func=None, last_modified_func=None)`[\[source\]](#)

`etag(etag_func)`[\[source\]](#)

`last_modified(last_modified_func)`[\[source\]](#)

Ces décorateurs peuvent être utilisés pour générer des en-têtes ETag et Last-Modified. Voir [traitement conditionnel de vue](#).

Compression GZip

Les décorateurs présents dans [django.views.decorators.gzip](#) contrôlent la compression du contenu vue par vue.

`gzip_page()`

Ce décorateur compresse le contenu si le navigateur autorise la compression gzip. Il déclare l'en-tête Vary en conséquence, de façon à ce que la mise en cache dépende de l'en-tête Accept-Encoding.

En-têtes Vary

Les décorateurs présents dans [django.views.decorators.vary](#) peuvent être utilisés pour contrôler la mise en cache reposant sur certains en-têtes des requêtes.

`vary_on_cookie(func)`[\[source\]](#)

`vary_on_headers(*headers)`[\[source\]](#)

L'en-tête Vary définit quels en-têtes de requête un mécanisme de cache doit prendre en considération pour générer sa clé de cache.

Voir [utilisation des en-têtes vary](#).

Mise en cache

Les décorateurs présents dans [django.views.decorators.cache](#) contrôlent la mise en cache aux niveaux serveur et client.

`never_cache(view_func)`[\[source\]](#)

Ce décorateur ajoute un en-tête `Cache-Control: max-age=0, no-cache, no-store, must-revalidate` à une réponse pour indiquer que la page ne doit jamais être mise en cache.

Changed in Django 1.9:

Avant Django 1.9, l'en-tête envoyé contenait `Cache-Control: max-age=0`. Cela n'empêchait pas le cache de manière fiable dans tous les navigateurs.

[Écriture de vues](#)

[Envoi de fichiers](#)

Envoi de fichiers

Lorsque Django gère un téléversement de fichier, les données du fichier aboutissent dans [request.FILES](#) (plus de détails sur l'objet `request` se trouvent dans la documentation des [objets request et response](#)). Ce document explique comment les fichiers sont stockés sur disque et en mémoire, et comment personnaliser le comportement par défaut.

Avertissement

Il existe des risques de sécurité si vous acceptez du contenu téléversé de la part d'utilisateurs non fiables. Consultez le sujet [Contenu envoyé par les utilisateurs](#) du guide de sécurité pour des détails sur les mesures de réduction des risques.

Envoi simple de fichiers

Consider a simple form containing a [FileField](#):

forms.py

```
from django import forms
```

```
class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file = forms.FileField()
```

Une vue gérant ce formulaire recevra les données de fichier dans `request.FILES` qui est un dictionnaire contenant une clé pour chaque `FileField` (ou `ImageField`, ou toute autre sous-classe de `FileField`) du formulaire. Ainsi, les données du formulaire ci-dessus seraient accessibles dans `request.FILES['file']`.

Notez que `request.FILES` ne contient les données que lorsque la méthode de requête est POST et que le formulaire ``<form>`` à l'origine de la requête possède l'attribut `enctype="multipart/form-data"`. Sinon, `request.FILES` est vide.

Most of the time, you'll simply pass the file data from `request` into the form as described in [Liaison de fichiers téléversés avec un formulaire](#). This would look something like:

views.py

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm

# Imaginary function to handle an uploaded file.
from somewhere import handle_uploaded_file

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render(request, 'upload.html', {'form': form})
```

Remarquez que nous devons transmettre `request.FILES` au constructeur du formulaire ; c'est comme cela que les données de fichiers sont liées à un formulaire.

Voici une manière habituelle de gérer un fichier téléversé :

```
def handle_uploaded_file(f):
    with open('some/file/name.txt', 'wb+') as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

En bouclant sur `UploadedFile.chunks()` au lieu d'appeler `read()`, on peut s'assurer que les gros fichiers ne saturent pas la mémoire du système.

Il existe quelques autres méthodes et attributs accessibles pour les objets `UploadedFile`. Voir [UploadedFile](#) pour une référence complète.

Téléversement de fichiers liés à un modèle

Si vous enregistrez un fichier dans un `Model` contenant un champ `FileField`, l'emploi d'un formulaire `ModelForm` simplifie le processus. L'objet fichier sera enregistré à l'emplacement indiqué

par le paramètre [upload_to](#) du champ [FileField](#) correspondant lors de l'appel à `form.save()`:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import ModelFormWithFileField

def upload_file(request):
    if request.method == 'POST':
        form = ModelFormWithFileField(request.POST, request.FILES)
        if form.is_valid():
            # file is saved
            form.save()
            return HttpResponseRedirect('/success/url/')
    else:
        form = ModelFormWithFileField()
    return render(request, 'upload.html', {'form': form})
```

Si vous construisez manuellement un objet, vous pouvez simplement attribuer l'objet fichier provenant de [request.FILES](#) au champ de fichier du modèle :

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm
from .models import ModelWithFileField

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            instance = ModelWithFileField(file_field=request.FILES['file'])
            instance.save()
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render(request, 'upload.html', {'form': form})
```

Uploading multiple files

If you want to upload multiple files using one form field, set the `multiple` HTML attribute of field's widget:

forms.py

```
from django import forms

class FileFieldForm(forms.Form):
    file_field = forms.FileField(widget=forms.ClearableFileInput(attrs={'multiple': True}))
```

Then override the `post` method of your [FormView](#) subclass to handle multiple file uploads:

views.py

```
from django.views.generic.edit import FormView
from .forms import FileFieldForm
```

```

class FileFieldView(FormView):
    form_class = FileFieldForm
    template_name = 'upload.html' # Replace with your template.
    success_url = '...' # Replace with your URL or reverse().

    def post(self, request, *args, **kwargs):
        form_class = self.get_form_class()
        form = self.get_form(form_class)
        files = request.FILES.getlist('file_field')
        if form.is_valid():
            for f in files:
                ... # Do something with each file.
            return self.form_valid(form)
        else:
            return self.form_invalid(form)

```

Gestionnaires de téléversement

Lorsqu'un utilisateur envoie un fichier, Django transmet les données du fichier à un *gestionnaire de téléversement*, une petite classe qui gère les données du fichier reçu. Les gestionnaires de téléversement sont définis initialement dans le réglage [FILE_UPLOAD_HANDLERS](#) dont le contenu par défaut est :

```

["django.core.files.uploadhandler.MemoryFileUploadHandler",
 "django.core.files.uploadhandler.TemporaryFileUploadHandler"]

```

Conjointement, [MemoryFileUploadHandler](#) et [TemporaryFileUploadHandler](#) définissent le comportement Django par défaut de téléversement de fichier en plaçant les petits fichiers en mémoire et les plus gros sur le disque.

Vous pouvez écrire des gestionnaires personnalisée qui modifient la façon de gérer les fichiers. Vous pourriez par exemple utiliser des gestionnaires personnalisés pour limiter la quantité de données par utilisateur, compresser les données à la volée, afficher des barres de progression ou même envoyer directement des données vers un autre emplacement de stockage sans les stocker localement. Voir [Écriture de gestionnaires de téléversement personnalisés](#) pour plus de détails sur la manière de personnaliser ou de remplacer complètement le comportement de téléversement.

Emplacement de stockage des données

Avant d'enregistrer des fichiers téléversés, les données doivent bien être stockées quelque part.

Par défaut, si un fichier téléversé est plus petit que 2,5 Mio, Django place la totalité du fichier en mémoire. Cela signifie que l'enregistrement du fichier correspond à une simple lecture en mémoire et à une écriture sur le disque, ce qui est très rapide.

Cependant, si un fichier téléversé est trop gros, Django écrit le fichier dans un fichier temporaire stocké dans le répertoire des fichiers temporaires du système. Sur une plate-forme de type Unix, Django va en principe générer un fichier dont le chemin correspond à quelque chose comme

`/tmp/tmpzfp6I6.upload`. Si le fichier est suffisamment gros, vous pouvez même voir la taille du fichier augmenter au fur et à mesure de son flux d'enregistrement vers le disque.

Ces paramètres spécifiques, 2,5 Mio, `/tmp`, etc. ne sont que des valeurs par défaut « raisonnables » qui peuvent être personnalisés comme expliqué dans la section suivante.

Modification de la gestion des téléversements

Certains réglages permettent de contrôler le comportement de téléversement de fichiers de Django. Voir [Réglages de téléversement de fichiers](#) pour plus de détails.

Modification des gestionnaires de téléversement à la volée

Parfois, certaines vues ont besoin d'un comportement de téléversement différent. Dans ces situations, il est possible de surcharger les gestionnaires de téléversement par requête en modifiant `request.upload_handlers`. Par défaut, cette liste contient les gestionnaires de téléversement énumérés dans [FILE_UPLOAD_HANDLERS](#), mais cette liste est modifiable comme n'importe quelle autre liste.

Par exemple, supposons que vous ayez écrit un gestionnaire `ProgressBarUploadHandler` qui renvoie des informations de progression de téléversement d'un composant AJAX quelconque. Voici comment ajouter ce gestionnaire à la liste des gestionnaires de téléversement :

```
request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
```

Dans ce cas, `list.insert()` peut être plus approprié que `append()` parce qu'un gestionnaire de barre de progression devrait être exécuté *avant* tout autre gestionnaire. Souvenez-vous que les gestionnaires de téléversement sont appelés dans l'ordre.

Si vous vouliez totalement remplacer les gestionnaires de téléversement, il suffirait d'attribuer une nouvelle liste :

```
request.upload_handlers = [ProgressBarUploadHandler(request)]
```

Note

Il n'est possible de modifier les gestionnaires de téléversement qu'*avant* d'accéder à `request.POST` ou `request.FILES`, car ça n'aurait pas de sens de modifier les gestionnaires de téléversement après que la gestion des téléversements a déjà démarré. Si vous le faites tout de même, Django générera une erreur.

Ceci dit, la modification des gestionnaires de téléversement devrait toujours intervenir aussi tôt que possible dans les vues.

De plus, [CsrfViewMiddleware](#) qui est un intergiciel activé par défaut accède à `request.POST`. Cela signifie qu'il est nécessaire de décorer avec [csrf_exempt\(\)](#) les vues dans lesquelles vous souhaitez changer les gestionnaires de téléversement. Il faut ensuite utiliser [csrf_protect\(\)](#) sur la

fonction qui traite effectivement la requête. Remarquez qu'il se pourrait que les gestionnaires commencent à recevoir un fichier envoyé avant que les contrôles CSRF aient été effectués. Exemple de code :

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect

@csrf_exempt
def upload_file_view(request):
    request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
    return _upload_file_view(request)

@csrf_protect
def _upload_file_view(request):
    ... # Process request
```

[Décorateurs de vue](#)

[Les fonctions raccourcis de Django](#)

Les fonctions raccourcis de Django

Le paquet `django.shortcuts` rassemble des fonctions et des classes utilitaires qui recouvrent plusieurs niveaux de l'architecture MVC. En d'autres termes, ces fonctions/classes introduisent un couplage contrôlé à des fins de commodité.

render ()

`render(request, template_name, context=None, context_instance=_context_instance_undefined, content_type=None, status=None, current_app=_current_app_undefined, dirs=_dirs_undefined, using=None)`[\[source\]](#)

Combine un gabarit donné avec un dictionnaire contexte donné et renvoie un objet [HttpResponse](#) avec le texte résultant.

Django ne met pas à disposition de fonction raccourci renvoyant une réponse [TemplateResponse](#) parce que le constructeur de [TemplateResponse](#) présente le même niveau de flexibilité que [render\(\)](#).

Paramètres obligatoires

`request`

L'objet requête utilisé pour générer la réponse.

`template_name`

Le nom complet d'un gabarit à utiliser ou une liste de noms de gabarits. Si une liste est donnée, le premier gabarit de la liste qui est trouvé sera utilisé. Consultez la [documentation du chargement de gabarits](#) pour plus d'informations sur la façon dont les gabarits sont recherchés.

Paramètres facultatifs

`context`

Un dictionnaire de valeurs à ajouter un contexte du gabarit. Par défaut, ce dictionnaire est vide. Si une des valeurs du dictionnaire est exécutable, la vue l'appellera immédiatement avant de faire le rendu du gabarit.

Changed in Django 1.8:

Le paramètre `context` était appelé `dictionary`. Ce nom est obsolète dans Django 1.8 et sera supprimé dans Django 1.10.

`context_instance`

L'instance de contexte avec laquelle effectuer le rendu du gabarit. Par défaut, le gabarit sera rendu avec une instance `RequestContext` (complétée par des valeurs de `request` et `context`).

Obsolète depuis la version 1.8: Le paramètre `context_instance` est obsolète. Utilisez simplement `context`.

`content_type`

Le type MIME à utiliser pour le document produit. La valeur par défaut est celle du réglage [DEFAULT_CONTENT_TYPE](#).

`status`

Le code d'état de la réponse. La valeur par défaut est 200.

`current_app`

Une indication de l'application contenant la vue actuelle. Consultez la [stratégie de résolution des URL en espaces de noms](#) pour plus d'informations.

Obsolète depuis la version 1.8: Le paramètre `current_app` est obsolète. Vous devriez plutôt définir `request.current_app`.

`using`

Le nom [NAME](#) d'un moteur de gabarit à utiliser pour charger le gabarit.

Changed in Django 1.8:

Le paramètre `using` a été ajouté.

Obsolète depuis la version 1.8: Le paramètre `dirs` a été rendu obsolète.

Exemple

L'exemple suivant effectue le rendu du gabarit `myapp/index.html` avec le type MIME `application/xhtml+xml`:

```
from django.shortcuts import render

def my_view(request):
    # View code here...
    return render(request, 'myapp/index.html', {
        'foo': 'bar',
    }, content_type='application/xhtml+xml')
```

Cet exemple est équivalent à :

```
from django.http import HttpResponse
from django.template import loader

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/index.html')
    c = {'foo': 'bar'}
    return HttpResponse(t.render(c, request), content_type='application/xhtml+xml')
```

`render_to_response()`

```
render_to_response(template_name, context=None,
context_instance=_context_instance_undefined, content_type=None, status=None,
dirs=_dirs_undefined, using=None)\[source\]
```

Cette fonction a précédé l'introduction de [render\(\)](#) et fonctionne de manière similaire, sauf qu'elle ne rend pas la requête disponible dans la réponse. Elle n'est plus recommandée et va probablement être rendue obsolète à l'avenir.

Paramètres obligatoires

`template_name`

Le nom complet d'un gabarit à utiliser ou une liste de noms de gabarits. Si une liste est donnée, le premier gabarit de la liste qui est trouvé sera utilisé. Consultez la [documentation du chargement de gabarits](#) pour plus d'informations sur la façon dont les gabarits sont recherchés.

Paramètres facultatifs

`context`

Un dictionnaire de valeurs à ajouter un contexte du gabarit. Par défaut, ce dictionnaire est vide. Si une des valeurs du dictionnaire est exécutable, la vue l'appellera immédiatement avant de faire le rendu du gabarit.

Changed in Django 1.8:

Le paramètre `context` était appelé `dictionary`. Ce nom est obsolète dans Django 1.8 et sera supprimé dans Django 1.10.

`context_instance`

L'instance de contexte avec laquelle effectuer le rendu du gabarit. Par défaut, le gabarit sera rendu avec une instance [Context](#) (complétée par des valeurs de `context`). Si vous avez besoin d'utiliser des [processeurs de contexte](#), produisez plutôt le gabarit avec une instance [RequestContext](#). Votre code pourrait ressembler à ceci :

```
return render_to_response('my_template.html',
                          my_context,
                          context_instance=RequestContext(request))
```

Obsolète depuis la version 1.8: Le paramètre `context_instance` est obsolète. Utilisez plutôt la fonction [render\(\)](#) qui rend toujours disponible l'objet `RequestContext`.

`content_type`

Le type MIME à utiliser pour le document produit. La valeur par défaut est celle du réglage [DEFAULT_CONTENT_TYPE](#).

`status`

Le code d'état de la réponse. La valeur par défaut est 200.

`using`

Le nom [NAME](#) d'un moteur de gabarit à utiliser pour charger le gabarit.

Changed in Django 1.8:

Les paramètres `status` et `using` ont été ajoutés.

Obsolète depuis la version 1.8: Le paramètre `dirs` a été rendu obsolète.

`redirect()`

`redirect(to, permanent=False, *args, **kwargs)`[\[source\]](#)

Renvoie une réponse [HttpResponseRedirect](#) à l'URL correspondant aux paramètres transmis.

Les paramètres peuvent être :

- Un modèle : la fonction [`get_absolute_url\(\)`](#) du modèle sera appelée.
- Un nom de vue, et potentiellement des paramètres : [`urlresolvers.reverse`](#) sera utilisée pour résoudre le nom.
- Une URL absolue ou relative qui sera utilisée telle quelle comme emplacement de redirection.

Produit par défaut une redirection temporaire ; indiquez `permanent=True` pour produire une redirection permanente.

Exemples

La fonction [`redirect\(\)`](#) peut être utilisée de différentes manières.

1. En lui passant un objet ; la méthode [`get_absolute_url\(\)`](#) de l'objet est appelée pour produire l'URL de redirection :

```
from django.shortcuts import redirect

def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object)
```

2. En lui passant le nom d'une vue et, en option, des paramètres positionnels ou nommés ; l'URL sera résolue en utilisant la méthode [`reverse\(\)`](#):

```
def my_view(request):
    ...
    return redirect('some-view-name', foo='bar')
```

3. En lui passant une URL de redirection fixe :

```
def my_view(request):
    ...
    return redirect('/some/url/')
```

Cela fonctionne aussi avec des URL complètes :

```
def my_view(request):
    ...
    return redirect('https://example.com/')
```

Par défaut, [`redirect\(\)`](#) renvoie une redirection temporaire. Toutes les formes ci-dessus acceptent un paramètre `permanent` qui, s'il est défini à `True`, produit une redirection permanente :

```
def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object, permanent=True)
```

get_object_or_404()

`get_object_or_404(klass, *args, **kwargs)`[\[source\]](#)

Appelle [get\(\)](#) d'un gestionnaire de modèle donné, mais génère une exception [Http404](#) au lieu de l'exception [DoesNotExist](#) du modèle.

Paramètres obligatoires

`klass`

Une classe [Model](#), un [Manager](#) ou une instance [QuerySet](#) comme source de l'objet.

`**kwargs`

Paramètres de recherche, dans un format accepté par `get()` et `filter()`.

Exemple

L'exemple suivant obtient l'objet ``MyModel`` ayant la clé primaire 1 :

```
from django.shortcuts import get_object_or_404

def my_view(request):
    my_object = get_object_or_404(MyModel, pk=1)
```

Cet exemple est équivalent à :

```
from django.http import Http404

def my_view(request):
    try:
        my_object = MyModel.objects.get(pk=1)
    except MyModel.DoesNotExist:
        raise Http404("No MyModel matches the given query.")
```

Le cas le plus courant est de transmettre un [Model](#), comme montré ci-dessus. Cependant, vous pouvez aussi transmettre une instance [QuerySet](#):

```
queryset = Book.objects.filter(title__startswith='M')
get_object_or_404(queryset, pk=1)
```

L'exemple ci-dessus est un peu tiré par les cheveux, car il est équivalent à :

```
get_object_or_404(Book, title__startswith='M', pk=1)
```

mais cela reste utile dans les cas où vous recevez une variable `queryset` par un autre moyen.

Finalement, vous pouvez aussi utiliser un gestionnaire ([Manager](#)). C'est utile par exemple si vous avez un [gestionnaire personnalisé](#)

```
get_object_or_404(Book.dahl_objects, title='Matilda')
```

Il est aussi possible d'utiliser des [gestionnaires de liaison](#):

```
author = Author.objects.get(name='Roald Dahl')
get_object_or_404(author.book_set, title='Matilda')
```

Note : comme pour `get()`, une exception [MultipleObjectsReturned](#) sera levée si plus d'un objet est renvoyé.

get_list_or_404()

`get_list_or_404(klass, *args, **kwargs)`[\[source\]](#)

Renvoie le résultat de [filter\(\)](#) sur un gestionnaire de modèle donné transformé en liste, générant [Http404](#) si la liste résultante est vide.

Paramètres obligatoires

`klass`

Une instance de [Model](#), [Manager](#) ou [QuerySet](#) comme source de la liste d'objets.

`**kwargs`

Paramètres de recherche, dans un format accepté par `get()` et `filter()`.

Exemple

L'exemple suivant récupère tous les objets publiés de

```
from django.shortcuts import get_list_or_404
```

```
def my_view(request):
    my_objects = get_list_or_404(MyModel, published=True)
```

Cet exemple est équivalent à :

```
from django.http import Http404
```

```
def my_view(request):
    my_objects = list(MyModel.objects.filter(published=True))
    if not my_objects:
        raise Http404("No MyModel matches the given query.")
```

[Envoi de fichiers](#)

[Vues génériques](#)

Utilisation des sessions

Django prend complètement en charge les sessions anonymes. L'infrastructure des sessions permet de stocker et de récupérer des données arbitraires sur la base « un site-un visiteur ». Les données sont stockées côté serveur ; l'envoi et la réception des cookies sont transparents. Les cookies contiennent un identifiant de session, et non pas les données elles-mêmes (sauf si vous utilisez le [moteur basé sur les cookies](#)).

Activation des sessions

Les sessions sont implémentées par l'intermédiaire d'un composant [middleware](#).

Pour activer la fonctionnalité des sessions, faites ce qui suit :

- Contrôlez que le réglage [MIDDLEWARE_CLASSES](#) contient bien `'django.contrib.sessions.middleware.SessionMiddleware'`. Le fichier `settings.py` créé par défaut par `django-admin startproject` active `SessionMiddleware`.

Si vous ne souhaitez pas utiliser les sessions, vous pouvez très bien enlever la ligne

`SessionMiddleware` de [MIDDLEWARE_CLASSES](#) et la ligne

`'django.contrib.sessions'` de [INSTALLED_APPS](#). Cela économisera un peu de travail à Django.

Configuration du moteur de sessions

Par défaut, Django stocke les sessions dans la base de données (avec le modèle `django.contrib.sessions.models.Session`). Bien que cela puisse être pratique, il est plus rapide dans certaines configurations de stocker les données de sessions ailleurs ; il est donc possible de configurer Django pour stocker les données de sessions sur le système de fichiers ou dans un cache.

Utilisation des sessions en base de données

Si vous souhaitez placer le contenu des sessions en base de données, vous devez ajouter

`'django.contrib.sessions'` dans le réglage [INSTALLED_APPS](#).

Après avoir configuré votre installation, lancez `manage.py migrate` pour installer l'unique table de base de données qui stocke les données des sessions.

Utilisation des sessions en cache

Pour de meilleures performances, il est recommandé d'utiliser le moteur de sessions basé sur le cache.

Pour stocker les données de sessions en utilisant le système de cache de Django, il s'agit d'abord de s'assurer que le cache est configuré ; voir la [documentation du cache](#) pour plus de détails.

Avertissement

Les données de sessions ne devraient être stockées en cache que quand le moteur de cache Memcached est utilisé. Le moteur de cache en mémoire locale ne retient pas assez longtemps les données pour qu'il représente un bon choix, et il est plus rapide d'utiliser directement le moteur de sessions basé sur des fichiers ou sur la base de données que de faire transiter le tout par les moteurs de cache basés sur des fichiers ou sur la base de données. De plus, le moteur de cache en mémoire locale ne gère PAS correctement le mode multi-processus et n'est donc probablement pas un bon choix dans un environnement de production.

Si plusieurs caches sont définis dans [CACHES](#), Django utilise le cache par défaut. Pour utiliser un autre cache, définissez [SESSION_CACHE_ALIAS](#) avec le nom de ce cache.

Après avoir configuré le cache, il vous reste deux possibilités pour stocker les données en cache :

- Définissez [SESSION_ENGINE](#) à `"django.contrib.sessions.backends.cache"` pour un système de stockage de sessions simple. Les données de session seront stockées directement dans le cache. Cependant, la persistance de ces données n'est pas garantie : les données en cache peuvent être vidées si le cache est rempli ou si le serveur de cache est redémarré.
- Pour obtenir des données de sessions à la fois en cache et persistantes, définir [SESSION_ENGINE](#) à `"django.contrib.sessions.backends.cached_db"`. Il s'agit là d'un cache à écriture directe, chaque écriture dans le cache sera également écrite en base de données. Les sessions ne sont lues dans la base de données que si elles ne sont pas déjà dans le cache.

Les deux stockages de sessions sont assez rapides, mais le cache simple est le plus rapide car il ne s'occupe pas de persistance. Dans la plupart des cas, le moteur `cached_db` sera suffisamment rapide, mais si vous avez besoin de performances optimales et qu'il est acceptable pour vous que les données de sessions soient effacées de temps en temps, le moteur `cache` est fait pour vous.

Si vous utilisez le moteur de session `cached_db`, vous devrez aussi suivre les instructions de configuration décrites dans [Utilisation des sessions en base de données](#).

Utilisation des sessions basées sur des fichiers

Pour utiliser les sessions basées sur des fichiers, définissez le réglage [SESSION_ENGINE](#) à `"django.contrib.sessions.backends.file"`.

Il peut aussi être souhaitable de définir le réglage [SESSION_FILE_PATH](#) (dont la valeur par défaut est le résultat de `tempfile.gettempdir()`, généralement `/tmp`) pour contrôler l'emplacement où Django stocke les fichiers de sessions. Vérifiez que le serveur Web dispose des permissions en lecture et écriture à cet endroit.

Utilisation des sessions basées sur les cookies

Pour utiliser les sessions basées sur les cookies, définissez le réglage [SESSION_ENGINE](#) à `"django.contrib.sessions.backends.signed_cookies"`. Les données de sessions seront stockées en utilisant les outils de Django pour la [signature cryptographique](#) et le réglage [SECRET_KEY](#).

Note

Il est recommandé de laisser le réglage [SESSION_COOKIE_HTTPONLY](#) à la valeur `True` pour empêcher l'accès à des données stockées à partir du JavaScript.

Avertissement

**** Si la clé secrète `SECRET_KEY` n'est plus secrète et que vous utilisez** [PickleSerializer](#), cela peut aboutir à de l'exécution arbitraire de code à distance.**

Un attaquant en possession de la clé secrète [SECRET_KEY](#) peut non seulement générer des données de session falsifiées qui seront admises par votre site, mais peut aussi exécuter du code arbitraire à distance, dans la mesure où les données sont sérialisées par `pickle`.

Si vous utilisez des sessions basées sur les cookies, il faut faire d'autant plus attention à toujours conserver la clé secrète absolument secrète, pour tout système accessible à distance.

**** les données de sessions sont signées mais non chiffrées****

Lors de l'utilisation du moteur basé sur les cookies, les données de sessions peuvent être lues par le client.

Un code MAC (Message Authentication Code) est utilisé pour protéger les données contre des modifications par le client, afin que les données de sessions soient caduques quand elles ont été modifiées. La même protection intervient si le client qui stocke le cookie (par ex. le navigateur de l'utilisateur) ne peut pas stocker tout le cookie de session et tronque des données. Même si Django compresse les données, il est toujours possible de dépasser la [limite habituelle de 4096 octets](#) par cookie.

Aucune garantie de fraîcheur

Il faut également relever que bien que le code MAC puisse garantir l'authenticité des données (qu'elles ont bien été générées par votre site) ainsi que l'intégrité de ces données (qu'elles sont complètes et correctes), il ne peut pas garantir leur fraîcheur, c'est-à-dire que vous recevez la dernière version que vous avez envoyée au client. Cela signifie que pour certaines utilisations des données de session, le moteur par cookie pourrait vous rendre vulnérable à des [attaques par rejeu](#). Au contraire d'autres moteurs de session qui conservent une version de chaque session du côté serveur et qui les infirment lorsqu'un utilisateur se déconnecte, les sessions basées sur les cookies ne sont pas infirmées lorsqu'un utilisateur se déconnecte. Ainsi, si un attaquant vole un cookie d'utilisateur, il peut utiliser celui-ci pour

se connecter sous le nom de cet utilisateur même quand ce dernier s'est déconnecté. Les cookies ne sont considérées comme obsolètes qu'au moment où leur âge dépasse [SESSION_COOKIE_AGE](#).

Performance

Finalement, la taille d'un cookie peut influencer la [rapidité de votre site](#).

Utilisation des sessions dans les vues

Lorsque `SessionMiddleware` est actif, chaque objet [HttpRequest](#), le premier paramètre de toute fonction vue de Django, possède un attribut `session` qui est un objet de type dictionnaire.

Vous pouvez lire et écrire dans `request.session` à n'importe quel moment dans votre vue. Vous pouvez le modifier plusieurs fois.

`class backends.base.SessionBase`

C'est la classe de base pour tous les objets session. Elle possède les méthodes de dictionnaire standard suivantes :

`__getitem__(key)`

Exemple :

`__setitem__(key, value)`

Exemple : `request.session['fav_color'] = 'blue'`

`__delitem__(key)`

Exemple : `del request.session['fav_color']`. Si la clé indiquée ne se trouve pas déjà dans l'objet session, une exception `KeyError` est levée.

`__contains__(key)`

Exemple : `'fav_color' in request.session`

`get(key, default=None)`

Exemple : `fav_color = request.session.get('fav_color', 'red')`

`pop(key, default=__not_given)`

Exemple : `fav_color = request.session.pop('fav_color', 'blue')`

`keys()`

`items()`

`setdefault()`
`clear()`

Elle possède aussi ces méthodes :

`flush()`

Supprime les données de la session actuelle et supprime le cookie de session. C'est utile pour s'assurer que les données de session précédentes ne puissent plus être relues par le navigateur de l'utilisateur (par exemple, c'est ce que fait [django.contrib.auth.logout\(\)](#)).

Changed in Django 1.8:

La suppression du cookie de session est un nouveau comportement de Django 1.8. Précédemment, le comportement était de régénérer la valeur de clé de session qui était renvoyée à l'utilisateur dans le cookie.

`set_test_cookie()`

Crée un cookie de test pour déterminer si le navigateur de l'utilisateur prend en charge les cookies. En raison du fonctionnement des cookies, vous ne pourrez pas déterminer cela avant la prochaine requête de l'utilisateur. Consultez [Génération de cookies de test](#) ci-dessous pour obtenir plus d'informations.

`test_cookie_worked()`

Renvoie `True` ou `False`, selon que le navigateur de l'utilisateur a accepté le cookie de test ou non. En raison du fonctionnement des cookies, vous aurez dû appeler `set_test_cookie()` sur une requête de page précédente. Consultez [Génération de cookies de test](#) ci-dessous pour obtenir plus d'informations.

`delete_test_cookie()`

Supprime le cookie de test. Utilisé pour faire le ménage derrière soi.

`set_expiry(value)`

Définit la durée d'expiration de la session. Il est possible de passer plusieurs valeurs différentes :

- Si `value` est un nombre entier, la session va expirer après ce nombre de secondes d'inactivité. Par exemple, l'appel de `request.session.set_expiry(300)` va provoquer l'expiration de la session dans 5 minutes.

- Si `value` est un objet `datetime` ou `timedelta`, la session va expirer à cette date et heure spécifique. Notez que les valeurs `datetime` et `timedelta` ne sont sérialisables que si vous utilisez la sérialisation [PickleSerializer](#).
- Si la `value` est `0`, le cookie de session de l'utilisateur va expirer au moment où celui-ci ferme son navigateur Web.
- Si `value` est `None`, la session se fie à nouveau à la politique d'expiration globale des sessions.

La lecture d'une session n'est pas considérée comme une activité en regard de la notion d'expiration. L'expiration de la session est calculée à partir du moment de la dernière *modification* de la session.

`get_expiry_age()`

Renvoie le nombre de secondes restant jusqu'à l'expiration de la session. Pour les sessions sans expiration particulière (ou celles qui expirent à la fermeture du navigateur), cette valeur sera égale à [SESSION_COOKIE_AGE](#).

Cette fonction accepte deux paramètres nommés facultatifs :

- `modification`: dernière modification de la session, comme objet [datetime](#). Par défaut, c'est l'heure actuelle.
- `expiry`: information d'expiration de la session, comme objet [datetime](#), valeur [int](#) (en secondes) ou `None`. Par défaut, cette valeur est égale à celle stockée dans la session par [set_expiry\(\)](#), le cas échéant, sinon `None`.

`get_expiry_date()`

Renvoie la date à laquelle la session expire. Pour les sessions sans expiration particulière (ou celles qui expirent à la fermeture du navigateur), cette valeur sera égale à [SESSION_COOKIE_AGE](#) secondes à partir de maintenant.

Cette fonction accepte les mêmes paramètres mots-clés que [get_expiry_age\(\)](#).

`get_expire_at_browser_close()`

Renvoie `True` ou `False` selon que le cookie de session utilisateur expire au moment de la fermeture du navigateur Web de l'utilisateur ou non.

`clear_expired()`

Supprime les sessions expirées du stockage des sessions. Cette méthode de classe est appelée par [clearsessions](#).

cycle_key()

Crée une nouvelle clé de session tout en conservant les données de session actuelles. [django.contrib.auth.login\(\)](#) appelle cette méthode pour lutter contre les attaques par fixation de session.

Sérialisation des sessions

Par défaut, Django sérialise les données de session en JSON. Vous pouvez utiliser le réglage [SESSION_SERIALIZER](#) pour personnaliser le format de sérialisation de session. Même en considérant les limites décrites dans [Écriture de son propre sérialiseur](#), nous recommandons fortement d'en rester à la sérialisation JSON, *d'autant plus si vous utilisez le moteur cookie*.

Par exemple, voici un scénario d'attaque si vous utilisez [pickle](#) pour sérialiser les données de session. Si vous utilisez le [moteur de session basé sur des cookies signés](#) et que la clé [SECRET_KEY](#) est connue d'un attaquant (il n'y a aucune vulnérabilité connue dans Django qui divulguerait cette information), cet attaquant pourrait insérer une chaîne dans la session, ce qui pourrait conduire, au moment de la désérialisation (« unpickling »), à une exécution de code arbitraire sur le serveur. La technique pour faire cela est simple et aisément disponible sur Internet. Même si le stockage de sessions par cookies signe les données de session pour empêcher leur falsification, une divulgation de la clé [SECRET_KEY](#) crée immédiatement une vulnérabilité d'exécution de code à distance.

Sérialiseurs intégrés

`class serializers.JSONSerializer`

Un adaptateur du sérialiseur JSON provenant de [django.core.signing](#). Ne peut sérialiser que des types de données simples.

De plus, comme JSON ne gère que des clés textuelles, il faut savoir que des clés non textuelles dans `request.session` ne fonctionneront pas comme prévu :

```
>>> # initial assignment
>>> request.session[0] = 'bar'
>>> # subsequent requests following serialization & deserialization
>>> # of session data
>>> request.session[0] # KeyError
>>> request.session['0']
'bar'
```

Similarly, data that can't be encoded in JSON, such as non-UTF8 bytes like `'\xd9'` (which raises [UnicodeDecodeError](#)), can't be stored.

Consultez la section [Écriture de son propre sérialiseur](#) pour plus de détails sur les limites de la sérialisation JSON.

`class serializers.PickleSerializer`

Accepter n'importe quel objet Python, mais, comme expliqué ci-dessus, peut conduire à des vulnérabilités d'exécution de code à distance si la clé [`SECRET_KEY`](#) est révélée à un attaquant.

Écriture de son propre sérialiseur

Notez qu'au contraire de [`PickleSerializer`](#), [`JSONSerializer`](#) ne sait pas traiter n'importe quel type de donnée Python. Comme c'est souvent le cas, il faut contrebalancer commodité et sécurité. Si vous souhaitez stocker des types de données plus spécialisées comme `datetime` ou `Decimal` dans des sessions en JSON, vous devrez écrire votre propre sérialiseur (ou convertir ces valeurs en objets sérialisables en JSON avant de les stocker dans `request.session`). Bien que la sérialisation de telles valeurs soit relativement simple à réaliser (`django.core.serializers.json.DateTimeAwareJSONEncoder` donne des pistes), l'écriture d'un décodeur qui puisse reconstruire la valeur initiale de manière fiable est plus délicate. Par exemple, vous risquez de créer un objet `datetime` pour ce qui était en fait une chaîne dont le format ressemblait à celui que vous avez choisi pour représenter les objets `datetime`.

Une classe de sérialisation doit implémenter deux méthodes, `dumps(self, obj)` et `loads(self, data)`, pour respectivement sérialiser et désérialiser le dictionnaire des données de session.

Lignes directrices des objets sessions

- Utilisez des chaînes Python normales comme clés de dictionnaire pour `request.session`. Il s'agit plus d'une convention que d'une règle stricte.
- Les clés de dictionnaire de session commençant par un soulignement sont réservés pour l'usage interne de Django.
- Ne surchargez pas `request.session` avec un nouvel objet et n'accédez pas ou ne modifiez pas ses attributs. Utilisez-le comme un dictionnaire Python.

Exemples

Cette vue très simple définit une variable `has_commented` à `True` après qu'un utilisateur a soumis un commentaire. Elle n'autorise pas plus d'un commentaire par utilisateur :

```
def post_comment(request, new_comment):
    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

Cette vue très simple connecte un « membre » du site :

```
def login(request):
    m = Member.objects.get(username=request.POST['username'])
```

```

if m.password == request.POST['password']:
    request.session['member_id'] = m.id
    return HttpResponse("You're logged in.")
else:
    return HttpResponse("Your username and password didn't match.")

```

...et celle-ci déconnecte un membre, compte tenu de l'exemple `login()` ci-dessus :

```

def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
        pass
    return HttpResponse("You're logged out.")

```

La fonction standard [`django.contrib.auth.logout\(\)`](#) fait en réalité un peu plus que cela pour empêcher des divulgations de données fortuites. Elle appelle la méthode [`flush\(\)`](#) de `request.session`. Nous utilisons cet exemple comme démonstration du fonctionnement des objets sessions, il ne faut pas le considérer comme une implémentation complète de `logout()`.

Génération de cookies de test

Par commodité, Django fournit un moyen simple de tester si le navigateur de l'utilisateur accepte les cookies. Il suffit d'appeler la méthode [`set_test_cookie\(\)`](#) de `request.session` dans une vue, et d'appeler [`test_cookie_worked\(\)`](#) dans une des vues suivantes, mais pas dans le même appel de vue.

Cette séparation bizarre entre `set_test_cookie()` et `test_cookie_worked()` est nécessaire en raison du fonctionnement des cookies. Lorsque vous définissez un cookie, il n'est en fait pas possible de savoir si le navigateur l'a accepté avant de recevoir la prochaine requête du navigateur.

Il est recommandé d'utiliser [`delete_test_cookie\(\)`](#) pour faire le ménage derrière soi. Faites-le après avoir vérifié que le cookie de test a fonctionné.

Voici un exemple d'utilisation typique :

```

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

def login(request):
    if request.method == 'POST':
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            return HttpResponse("You're logged in.")
        else:
            return HttpResponse("Please enable cookies and try again.")
    request.session.set_test_cookie()
    return render(request, 'foo/login_form.html')

```


Utilisation des sessions en dehors des vues

Note

Les exemples de cette section importent directement l'objet `SessionStore` à partir du moteur `django.contrib.sessions.backends.db`. Dans votre propre code, il est préférable d'importer `SessionStore` à partir du moteur de sessions désigné par [SESSION_ENGINE](#), comme ci-dessous :

```
>>> from importlib import import_module
>>> from django.conf import settings
>>> SessionStore = import_module(settings.SESSION_ENGINE).SessionStore
```

Une API est disponible pour manipuler les données de sessions en dehors des vues :

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore()
>>> # stored as seconds since epoch since datetimes are not serializable in JSON.
>>> s['last_login'] = 1376587691
>>> s.create()
>>> s.session_key
'2b1189a188b44ad18c35e113ac6ceead'
>>> s = SessionStore(session_key='2b1189a188b44ad18c35e113ac6ceead')
>>> s['last_login']
1376587691
```

`SessionStore.create()` is designed to create a new session (i.e. one not loaded from the session store and with `session_key=None`). `save()` is designed to save an existing session (i.e. one loaded from the session store). Calling `save()` on a new session may also work but has a small chance of generating a `session_key` that collides with an existing one. `create()` calls `save()` and loops until an unused `session_key` is generated.

Si vous utilisez le moteur `django.contrib.sessions.backends.db`, chaque session n'est en fait qu'un modèle Django normal. Le modèle `Session` est défini dans `django/contrib/sessions/models.py`. Comme c'est un modèle normal, vous pouvez accéder aux sessions en utilisant l'API habituelle de base de données Django :

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

Notez que vous devrez appeler [get_decoded\(\)](#) pour obtenir le dictionnaire de session. Cela est nécessaire car le dictionnaire est stocké dans un format codé :

```
>>> s.session_data
'KGRwMQpTJ19hdXR0X3VzZXJfawQnCnAyCkKxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

Enregistrement des sessions

Par défaut, Django n'enregistre la session dans la base de données qu'au moment où la session a été modifiée, c'est-à-dire qu'au moins une des valeurs de son dictionnaire a été définie ou supprimée :

```
# Session is modified.
request.session['foo'] = 'bar'

# Session is modified.
del request.session['foo']

# Session is modified.
request.session['foo'] = {}

# Gotcha: Session is NOT modified, because this alters
# request.session['foo'] instead of request.session.
request.session['foo']['bar'] = 'baz'
```

Dans le dernier cas de l'exemple ci-dessus, nous pouvons indiquer explicitement à l'objet session qu'il a été modifié en définissant son attribut `modified`:

```
request.session.modified = True
```

Pour modifier ce comportement par défaut, définissez le réglage [`SESSION_SAVE_EVERY_REQUEST`](#) à `True`. Dans ce cas, Django enregistre la session dans la base de données lors de chaque requête.

Notez que le cookie de session n'est envoyé que lorsqu'une session a été créée ou modifiée. Si [`SESSION_SAVE_EVERY_REQUEST`](#) vaut `True`, le cookie de session sera envoyé lors de chaque requête.

De même, la partie `expires` d'un cookie de session est mis à jour chaque fois que le cookie de session est envoyé.

La session n'est pas enregistrée si le code de statut de la réponse est 500.

Expiration ou persistance des sessions

Le réglage [`SESSION_EXPIRE_AT_BROWSER_CLOSE`](#) permet de contrôler le type des sessions utilisées par l'infrastructure des sessions, soit les sessions expirant à la fermeture du navigateur ou les sessions persistantes.

Par défaut, [`SESSION_EXPIRE_AT_BROWSER_CLOSE`](#) est défini à `False`, ce qui signifie que les cookies de session seront stockés dans les navigateurs des utilisateurs pour une durée de [`SESSION_COOKIE_AGE`](#). Utilisé quand on ne souhaite pas que les utilisateurs doivent se connecter chaque fois qu'ils ouvrent leur navigateur.

Si [`SESSION_EXPIRE_AT_BROWSER_CLOSE`](#) est défini à `True`, Django utilise des cookies expirant à la fermeture du navigateur. Utilisé quand on souhaite que les utilisateurs doivent se connecter chaque fois qu'ils ouvrent leur navigateur.

Ce réglage est une valeur par défaut globale et peut être surchargé au niveau de chaque session en appelant explicitement la méthode [`set_expiry\(\)`](#) de `request.session` comme décrit ci-dessus dans [utilisation des sessions dans les vues](#).

Note

Certains navigateurs (Chrome, par exemple) peuvent être réglés afin de ne pas fermer les sessions de navigation entre fermeture et réouverture du navigateur. Dans certains cas, cela peut interférer avec le réglage [`SESSION_EXPIRE_AT_BROWSER_CLOSE`](#) et empêcher les sessions d'expirer à la fermeture du navigateur. Il est important de le savoir lors des tests d'applications Django qui ont le réglage [`SESSION_EXPIRE_AT_BROWSER_CLOSE`](#) actif.

Effacement du stockage des sessions

Au fur et à mesure que des utilisateurs créent de nouvelles sessions sur votre site Web, les données de sessions s'accumulent dans votre stockage de sessions. Si vous utilisez le moteur en base de données, la table de base de données `django_session` prend du volume. Si vous utilisez le moteur basé sur des fichiers, le nombre des fichiers dans le répertoire temporaire va augmenter constamment.

Pour comprendre ce problème, examinons ce qui se passe avec le moteur en base de données.

Lorsqu'un utilisateur se connecte, Django ajoute un enregistrement dans la table `django_session`.

Il met à jour cet enregistrement chaque fois que les données de la session sont modifiées. Si l'utilisateur se déconnecte explicitement, Django efface l'enregistrement. Mais si l'utilisateur ne le fait *pas*, l'enregistrement n'est jamais effacé. Le principe est identique avec le moteur basé sur les fichiers.

Django ne fournit *pas* de système automatique d'effacement des sessions expirées. C'est donc à vous de le faire à intervalle régulier. Django met à disposition une commande de gestion de nettoyage dans ce but : [`clearsessions`](#). Il est recommandé de lancer cette commande régulièrement, par exemple dans une tâche « cron » journalière.

Notez que le moteur basé sur le cache n'est pas sujet à ce problème, car les caches suppriment automatiquement les données obsolètes. Pareil pour le moteur basé sur les cookies, car les données de sessions sont stockées par les navigateurs des utilisateurs.

Réglages

Quelques [réglages Django](#) permettent de contrôler le comportement des sessions :

- [`SESSION_CACHE_ALIAS`](#)
- [`SESSION_COOKIE_AGE`](#)
- [`SESSION_COOKIE_DOMAIN`](#)
- [`SESSION_COOKIE_HTTPONLY`](#)
- [`SESSION_COOKIE_NAME`](#)
- [`SESSION_COOKIE_PATH`](#)

- [SESSION COOKIE SECURE](#)
- [SESSION ENGINE](#)
- [SESSION EXPIRE AT BROWSER CLOSE](#)
- [SESSION FILE PATH](#)
- [SESSION SAVE EVERY REQUEST](#)
- [SESSION SERIALIZER](#)

Sécurité des sessions

Les sous-domaines d'un site peuvent définir des cookies pour le client valables pour tout le domaine. Cela rend possible les attaques par fixation de session si des cookies peuvent être créés par des sous-domaines qui ne sont pas sous contrôle de personnes de confiance.

Par exemple, un attaquant pourrait se connecter à `good.example.com` et obtenir une session valable pour son compte. Si l'attaquant contrôle `bad.example.com`, il peut l'utiliser pour vous envoyer sa clé de session puisqu'un sous-domaine a le droit de définir des cookies pour `*.example.com`. Lorsque vous visitez `good.example.com`, vous serez connecté sous le compte de l'attaquant et vous pourriez saisir involontairement des données personnelles sensibles (par ex. des données de carte de crédit) enregistrées dans le compte de l'attaquant.

Une autre attaque possible apparaît dans le cas où `good.example.com` définit son réglage [SESSION_COOKIE_DOMAIN](#) à `".example.com"`, ce qui pourrait avoir comme conséquence d'envoyer les cookies de session à `bad.example.com`.

Détails techniques

- Le dictionnaire de session accepte n'importe quelle données sérialisable en [json](#) lors de l'utilisation de [JSONSerializer](#) ou n'importe quel objet Python sérialisable par `pickle` en cas d'utilisation de [PickleSerializer](#). Consultez la documentation du module [pickle](#) pour plus d'informations.
- Les données des sessions sont stockées dans une table de base de données nommée `django_session`.
- Django n'envoie un cookie qu'en cas de nécessité. Si vous ne définissez pas de données de session, le cookie de session ne sera pas envoyé.

L'objet SessionStore

Lorsque Django manipule les sessions en interne, il emploie un objet de stockage de session provenant du moteur de sessions correspondant. Par convention, la classe d'objet de stockage de session est appelée `SessionStore` et se trouve dans le module désigné par [SESSION_ENGINE](#).

Toutes les classes `SessionStore` disponibles dans Django héritent de [`SessionBase`](#) et implémentent des méthodes de manipulation de données, en l'occurrence :

- `exists()`
- `create()`
- `save()`
- `delete()`
- `load()`
- [`clear_expired\(\)`](#)

Dans l'optique de créer un moteur de sessions personnalisé ou d'en adapter un existant, vous pouvez créer une nouvelle classe héritant de [`SessionBase`](#) ou de toute autre classe `SessionStore` existante.

Il est relativement simple d'étendre la plupart des moteurs de sessions existants, à l'exception des moteurs se basant sur une base de données qui exigent généralement un peu plus de travail (lisez la section suivante pour plus de détails).

Extension des moteurs de sessions s'appuyant sur une base de données

New in Django 1.9.

La création d'un moteur de sessions personnalisé s'appuyant sur une base de données en se basant sur ceux qui sont inclus dans Django (c'est-à-dire `db` et `cached_db`) peut se faire en héritant de [`AbstractBaseSession`](#) et de l'une des classes `SessionStore`.

`AbstractBaseSession` et `BaseSessionManager` peuvent être importés à partir de `django.contrib.sessions.base_session` afin qu'ils puissent être importés sans devoir inclure `django.contrib.sessions` dans [`INSTALLED_APPS`](#).

```
class base_session.AbstractBaseSession
```

New in Django 1.9.

Le modèle abstrait de base d'une session.

`session_key`

La clé primaire. Le champ lui-même peut contenir jusqu'à 40 caractères. L'implémentation actuelle génère une chaîne de 32 caractères (une séquence aléatoire de chiffres et de lettre ASCII minuscules).

`session_data`

Une chaîne contenant un dictionnaire de session codé et sérialisé.

`expire_date`

Une date/heure indiquant la date d'expiration de la session.

Les sessions qui ont expiré ne sont plus accessibles aux utilisateurs ; elles peuvent tout de même rester stockées dans la base de données jusqu'au lancement de la commande d'administration [clearsessions](#).

classmethod `get_session_store_class()`

Renvoie une classe de stockage de session à utiliser avec ce modèle de session.

`get_decoded()`

Renvoie les données de session décodées.

Le décodage est effectué par la classe de stockage de session.

Vous pouvez aussi personnaliser le gestionnaire du modèle en héritant de [BaseSessionManager](#):

class `base_session.BaseSessionManager`

New in Django 1.9.

`encode(session_dict)`

Renvoie le dictionnaire de session indiqué sous forme de chaîne codée et sérialisée.

Le codage est effectué par la classe de stockage de session en lien à la classe de modèle.

`save(session_key, session_dict, expire_date)`

Enregistre les données de session correspondant à la clé de session indiquée ou supprimer la session dans le cas où les données sont vides.

La personnalisation des classes `SessionStore` se fait en surchargeant les méthodes et les propriétés décrites ci-après :

class `backends.db.SessionStore`

Implémentation du stockage de session s'appuyant sur une base de données.

classmethod `get_model_class()`

New in Django 1.9.

Surchargez cette méthode afin de renvoyer un modèle de session personnalisé en cas de besoin.

`create_model_instance(data)`

New in Django 1.9.

Renvoie une nouvelle instance d'objet modèle de session, qui représente l'état actuel de la session.

En surchargeant cette méthode, on peut modifier les données du modèle de session avant qu'elles ne soient enregistrées dans la base de données.

`class backends.cached_db.SessionStore`

Implémentation du stockage de session s'appuyant sur une base de données et du cache.

`cache_key_prefix`
New in Django 1.9.

Un préfixe ajouté à une clé de session pour construire une chaîne de clé de cache.

Exemple

L'exemple ci-dessous montre un moteur de session personnalisé s'appuyant sur une base de données, incluant une colonne de base de données supplémentaire pour stocker un identifiant de compte (et ainsi fournissant une option pour interroger la base de données pour toutes les sessions actives d'un compte) :

```
from django.contrib.sessions.backends.db import SessionStore as DBStore
from django.contrib.sessions.base_session import AbstractBaseSession
from django.db import models

class CustomSession(AbstractBaseSession):
    account_id = models.IntegerField(null=True, db_index=True)

    @classmethod
    def get_session_store_class(cls):
        return SessionStore

class SessionStore(DBStore):
    @classmethod
    def get_model_class(cls):
        return CustomSession

    def create_model_instance(self, data):
        obj = super(SessionStore, self).create_model_instance(data)
        try:
            account_id = int(data.get('_auth_user_id'))
        except (ValueError, TypeError):
            account_id = None
        obj.account_id = account_id
        return obj
```

Si vous faites une migration à partir du stockage de sessions `cached_db` intégré à Django vers un stockage personnalisé basé sur `cached_db`, il est conseillé de surcharger le préfixe de clé de cache afin d'éviter d'éventuels conflits d'espace de noms :

```
class SessionStore(CachedDBStore):
```

```
cache_key_prefix = 'mysessions.custom_cached_db_backend'  
  
# ...
```

Identifiants de session dans les URL

L'infrastructure des sessions de Django est complètement et uniquement basé sur les cookies. Il ne se rabat pas sur le système des identifiants de session dans les URL comme le fait PHP. C'est une décision conceptuelle volontaire. Cette technique défigure les URL et rend votre site vulnérable aux vols d'identifiants de session par l'intermédiaire de l'en-tête « Referer ».

[Intergiciels \(« Middleware »\)](#)
[Utilisation des formulaires](#)

Utilisation des formulaires

À propos de ce document

Ce document présente une introduction aux bases des formulaires web et à la manière dont ils sont traités par Django. Pour plus de détails sur des aspects spécifiques de l'API des formulaires, consultez [L'API des formulaires](#), [Champs de formulaires](#), et [La validation de formulaires et de champs](#).

À moins que vous n'ayez l'intention de construire des sites Web et des applications qui ne font rien d'autre que de publier du contenu, et n'acceptant aucune saisie de la part de ses visiteurs, vous allez avoir besoin de comprendre et d'utiliser des formulaires.

Django fournit un certain nombre d'outils et de bibliothèques pour vous aider à créer des formulaires de saisie pour les visiteurs du site, puis pour aider à traiter et à répondre à ces saisies.

Formulaires HTML

En HTML, un formulaire est un ensemble d'éléments à l'intérieur des balises `<form>...</form>` qui permettent à un visiteur d'effectuer des actions comme saisir du texte, sélectionner des options, manipuler des objets ou des contrôles, et ainsi de suite, puis d'envoyer ces informations au serveur.

Certains de ces éléments d'interface de formulaires - boîtes de saisie de texte ou cases à cocher - sont assez simples et intégrés dans le HTML lui-même. D'autres éléments sont beaucoup plus complexes ; une interface qui affiche un sélecteur de date ou qui permet de déplacer un curseur ou de contrôler un autre élément graphique utilise généralement du code JavaScript et CSS, ainsi que des éléments `<input>` de formulaires HTML pour produire ce genre d'effets.

En plus de ses éléments `<input>`, un formulaire doit préciser deux choses :

- *où* : l'URL vers laquelle les données correspondant à la saisie de l'utilisateur doivent être renvoyées
- *comment* : la méthode HTTP utilisée pour renvoyer les données

Par exemple, le formulaire de connexion pour l'administration de Django contient plusieurs éléments `<input>`: un de `type="text"` pour le nom d'utilisateur, un de `type="password"` pour le mot de passe, et un de `type="submit"` pour le bouton « Connexion ». Il contient également des champs de texte cachés que l'utilisateur ne voit pas, mais que Django utilise pour déterminer ce qu'il faut faire après la connexion.

Il indique également au navigateur que les données du formulaire doivent être envoyées à l'URL spécifiée dans l'attribut `action` de la balise `<form>` (`/admin/`) et qu'elles doivent être envoyées en utilisant le mécanisme HTTP spécifié par l'attribut `method` (`post`).

Lorsque le bouton `<input type="submit" value="Connexion">` est déclenché, les données sont renvoyées à `/admin/`.

GET et POST

GET et POST sont les seules méthodes HTTP à utiliser dans les formulaires.

Le formulaire de connexion de Django est renvoyé en utilisant la méthode **POST**, par laquelle le navigateur regroupe les données du formulaire, les encode pour la transmission, les envoie au serveur, puis reçoit la réponse en retour.

GET, en revanche, regroupe les données fournies dans une chaîne et l'utilise pour composer une URL. L'URL contient l'adresse à laquelle les données doivent être envoyées, ainsi que les clés et les valeurs fournies. Vous pouvez en voir un exemple si vous effectuez une recherche dans la documentation de Django, car cela va produire une URL de la forme `https://docs.djangoproject.com/search/?q=forms&release=1`.

GET et POST sont généralement utilisées à des fins différentes.

Toute requête pouvant être utilisée pour modifier l'état du système, par exemple une requête qui applique des changements en base de données, devrait utiliser **POST**. GET ne devrait être employée que pour des requêtes qui n'affectent pas l'état du système.

GET ne conviendrait pas non plus pour un formulaire de changement de mot de passe, car le mot de passe apparaîtrait en clair dans l'URL ainsi que dans l'historique du navigateur et les journaux du serveur. Elle ne conviendrait pas non plus pour des données volumineuses ou pour des données binaires telles que pour une image. Une application Web utilisant des requêtes GET pour des formulaires administratifs ne serait pas sécurisée : il est facile pour un attaquant de simuler une requête de formulaire pour obtenir l'accès à des parties sensibles du système. Les requêtes **POST** accompagnées

par d'autres protections comme la [protection CSRF](#) de Django offrent un bien meilleur contrôle sur les accès.

D'autre part, GET convient pour des interfaces comme un formulaire de recherche Web, car les URL qui représentent une requête GET peuvent être facilement conservées dans un signet, partagées ou ré-envoyées.

Le rôle de Django dans les formulaires

La gestion des formulaires est une affaire complexe. Considérez l'application d'administration de Django où de nombreuses données de différents types doivent être préparées pour être affichées dans un formulaire, produites en HTML, éditées par une interface conviviale, renvoyées vers le serveur, validées et nettoyées, et finalement enregistrées ou transmises pour traitement ultérieur.

La fonctionnalité des formulaires de Django peut simplifier et automatiser de larges portions de ce travail et peut aussi le faire de manière plus sécurisée que la plupart des programmeurs ne pourraient le faire en écrivant leur propre code.

Django gère trois parties distinctes du travail induit par les formulaires :

- préparation et restructuration des données en vue de leur présentation
- création des formulaires HTML pour les données
- réception et traitement des formulaires et des données envoyés par le client

Il est *possible* d'écrire du code qui fait tout cela manuellement, mais Django peut s'en charger à votre place.

Les formulaires dans Django

Nous avons brièvement décrit les formulaires HTML, mais un objet HTML `<form>` n'est qu'un élément de la machinerie nécessaire.

Dans le contexte d'une application Web, « formulaire » peut se référer à cette balise HTML `<form>`, à la classe [Form](#) de Django qui la produit, aux données structurées renvoyées lorsque le formulaire est soumis, ou encore au fonctionnement de bout en bout de ces différentes parties.

La classe [Form](#) de Django

La classe [Form](#) de Django se situe au cœur de ce système de composants. De la même façon qu'un modèle Django décrit la structure logique d'un objet, son comportement et la manière dont ses parties nous sont présentées, une classe [Form](#) décrit un formulaire et détermine son fonctionnement et son apparence.

De la même façon que les champs d'une classe de modèle correspondent à des champs de base de données, les champs d'une classe de formulaire correspondent aux éléments `<input>` d'un formulaire

HTML (un [ModelForm](#) fait correspondre les champs d'une classe de modèle à des éléments `<input>` d'un formulaire HTML via un formulaire [Form](#); cela constitue la base de l'interface d'administration de Django).

Les champs d'un formulaire sont eux-mêmes des classes ; elles gèrent les données de formulaire et s'occupent de la validation lorsque des données de formulaires sont reçues. Un champ [DateField](#) et un champ [FileField](#) gèrent des types de données très différents et doivent procéder à des actions différentes.

Un champ de formulaire est présenté dans une interface de navigateur sous forme de « composant » HTML, un élément d'interface utilisateur. Chaque type de champ contient une [classe Widget](#) appropriée par défaut, mais celles-ci peuvent être remplacées au besoin.

Instanciation, traitement et rendu des formulaires

Lors du rendu d'un objet Django pour son affichage, il s'agit généralement de :

1. obtenir l'élément dans la vue (le récupérer à partir de la base de données, par exemple)
2. le transmettre au contexte de gabarit
3. le transformer en balisage HTML en utilisant des variables de gabarit

L'affichage d'un formulaire dans un gabarit suit à peu près le même processus que pour tout autre type d'objet, mais il y a tout de même certaines différences notables.

Dans le cas d'une instance de modèle ne contenant aucune donnée, il n'y aura pas beaucoup de sens d'en faire quoi que ce soit dans un gabarit. En revanche, il est totalement raisonnable de vouloir afficher un formulaire vierge, c'est ce qu'il faut faire lorsqu'on veut que quelqu'un le remplisse.

Ainsi, lorsque nous manipulons une instance de modèle dans une vue, nous la récupérons typiquement à partir de la base de données. Lorsque nous manipulons un formulaire, nous en créons typiquement une instance dans la vue.

Lors de la création d'une instance de formulaire, on peut choisir de le laisser vide ou de le pré-remplir, par exemple avec :

- des données provenant d'une instance de modèle enregistrée (comme dans le cas des formulaires d'administration pour l'édition)
- des données que nous avons obtenues à partir d'autres sources
- des données reçues d'un envoi de formulaire HTML

Le dernier de ces cas est le plus intéressant, car c'est ce qui permet aux utilisateurs de ne pas seulement lire un site Web, mais aussi de renvoyer des informations aux sites Web.

Construction d'un formulaire

Le travail à effectuer

Supposons que vous vouliez créer un formulaire simple sur votre site dans le but d'obtenir le nom de l'utilisateur. Voici ce qu'il faudrait obtenir dans le navigateur :

```
<form action="/your-name/" method="post">
  <label for="your_name">Your name: </label>
  <input id="your_name" type="text" name="your_name" value="{{ current_name }}">
  <input type="submit" value="OK">
</form>
```

Le navigateur saura alors qu'il doit renvoyer les données du formulaire vers l'URL `/your-name/`, en utilisant la méthode POST. Il affichera un champ de texte libellé « Your name: » et un bouton intitulé « OK ». Si le contexte de gabarit contient une variable `current_name`, celle-ci sera utilisée pour pré-remplir le champ `your_name`.

Il faudra une vue qui produit le gabarit contenant le formulaire HTML et qui puisse fournir le champ `current_name` si nécessaire.

Lorsque le formulaire aura été envoyé, la requête POST envoyée au serveur contiendra les données du formulaire.

Mais il sera également nécessaire d'avoir une vue répondant à l'URL `/your-name/` qui se chargera de trouver les paires clé-valeur dans la requête, puis de les traiter.

Il s'agit d'un formulaire très simple. En pratique, un formulaire peut contenir des dizaines ou centaines de champs, dont beaucoup peuvent être pré-remplis, et il est possible que l'utilisateur doive passer par plusieurs cycles d'envoi/édition avant que l'opération globale ne soit terminée.

On peut attendre du navigateur qu'il s'occupe d'une partie de la validation, même avant que le formulaire ne soit envoyé. Peut-être que certains champs seront bien plus complexes, permettant par exemple de choisir des dates dans un calendrier.

Compte tenu de tout ceci, il est bien plus avantageux que Django se charge lui-même de la plupart du travail.

Construction d'un formulaire dans Django

La classe [Form](#)

Nous savons déjà à quoi notre formulaire HTML doit ressembler. Notre point de départ dans Django est le suivant :

```
forms.py
```

```
from django import forms
```

```
class NameForm(forms.Form):
```

```
your_name = forms.CharField(label='Your name', max_length=100)
```

Cela définit une classe [Form](#) comportant un seul champ (`your_name`). Nous avons attribué une étiquette conviviale à ce champ ; celle-ci apparaîtra dans la balise `<label>` au moment de l’affichage (bien que dans ce cas, le contenu indiqué dans [label](#) est en réalité identique à celui qui aurait été généré automatiquement si nous n’avions rien fait).

La longueur maximale du champ est définie par [max_length](#). Cela fait deux choses : la balise HTML `<input>` reçoit l’attribut `maxlength="100"` (afin que le navigateur empêche lui-même la saisie d’un plus grand nombre de caractères), et ultérieurement lorsque Django reçoit en retour les données de formulaire, il valide que les données respectent cette longueur.

Une instance de [Form](#) possède une méthode [is_valid\(\)](#) qui procède aux routines de validation pour tous ses champs. Lorsque la méthode est appelée et que tous les champs contiennent des données valables, celle-ci :

- renvoie `True`;
- insère les données du formulaire dans l’attribut [cleaned_data](#).

Le formulaire complet, lorsqu’il est affiché pour la première fois, ressemble à ceci :

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100">
```

Notez qu’il n’inclut **pas** les balises `<form>` ni de bouton d’envoi. Ces éléments doivent être ajoutés par celui qui rédige le gabarit.

La vue

Les données de formulaire renvoyés à un site Web Django sont traitées par une vue, en principe la même qui a servi à produire le formulaire. Cela permet de réutiliser une partie de la même logique.

Pour traiter le formulaire, nous devons en créer une instance dans la vue à destination de l’URL à laquelle il doit apparaître :

views.py

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

from .forms import NameForm

def get_name(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the request:
        form = NameForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required
            # ...
```

```

        # redirect to a new URL:
        return HttpResponseRedirect('/thanks/')

    # if a GET (or any other method) we'll create a blank form
    else:
        form = NameForm()

    return render(request, 'name.html', {'form': form})

```

Si nous arrivons dans cette vue avec une requête **GET**, la vue crée une instance de formulaire vierge et la place dans le contexte de gabarit en vue de son affichage. C'est à quoi l'on peut s'attendre lors du premier accès à l'URL en question.

Si le formulaire est envoyé par une requête **POST**, la vue crée également une instance de formulaire et la complète avec les données reçues à partir de la requête : `form = NameForm(request.POST)`. C'est ce qu'on appelle « lier les données au formulaire » (il s'agit maintenant d'un formulaire *lié*, « bound » en anglais).

La méthode `is_valid()` est appelée ; si elle ne renvoie pas `True`, on se retrouve au stade du gabarit contenant le formulaire. Mais cette fois, le formulaire n'est plus vierge (*non lié* ou *unbound*) ce qui fait que le formulaire HTML sera rempli avec les données saisies précédemment, données qui peuvent être modifiées et corrigées selon les besoins.

Si `is_valid()` renvoie `True`, toutes les données validées du formulaire sont alors accessibles dans l'attribut `cleaned_data`. Ces données peuvent être utilisées pour mettre à jour la base de données ou effectuer d'autres opérations avant de renvoyer au navigateur une redirection HTTP lui indiquant l'URL à recharger.

Le gabarit

Il n'y a pas grand chose à faire dans le gabarit `name.html`. L'exemple le plus simple est :

```

<form action="/your-name/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit" />
</form>

```

Tous les champs de formulaire et leurs attributs seront convertis en balises HTML à partir de `{{ form }}` par le langage de gabarit de Django.

Formulaires et protection contre le « Cross site request forgery » (CSRF)

Django est livré avec une protection simple d'emploi [contre les attaques de type Cross Site Request Forgeries](#). Lors de l'envoi d'un formulaire par la méthode **POST** et la protection CSRF active, vous devez utiliser la balise de gabarit `csrf_token` comme dans l'exemple précédent. Cependant, comme la protection CSRF n'est pas directement liée aux formulaires dans les gabarits, cette balise est omise dans les exemples suivants de ce document.

Types de composants HTML5 et validation par le navigateur

Si un formulaire contient un champ de type [URLField](#), [EmailField](#) ou tout type de champ nombre entier, Django fait appel aux composants HTML5 (input) de type `url`, `email` et `number`. Par défaut, les navigateurs appliquent souvent leur propre validation de ces champs, qui peut être plus stricte que celle de Django. Si vous voulez échapper à cette validation, ajoutez l'attribut `novalidate` sur la balise `form` ou définissez un autre composant de formulaire pour le champ, comme par exemple [TextInput](#).

Nous possédons maintenant un formulaire Web fonctionnel, défini par une instance [Form](#) de Django, traité par une vue et affiché sous forme de balise HTML `<form>`.

C'est tout ce dont vous avez besoin pour commencer, mais l'infrastructure des formulaires a beaucoup plus à vous offrir. À partir du moment où vous comprenez les bases du processus décrit ci-dessus, il est bon de connaître les autres fonctionnalités mises à disposition par le système des formulaires et d'en savoir un peu plus au sujet de la machinerie sous-jacente.

Plus de détails sur les classes [Form](#) de Django

Toutes les classes de formulaires héritent de [django.forms.Form](#), y compris les classes [ModelForm](#) que vous pouvez rencontrer dans l'administration de Django.

Modèles et formulaires

En fait, si un formulaire est destiné à ajouter ou modifier directement un modèle Django, un formulaire [ModelForm](#) peut vous économiser beaucoup de temps, d'effort et de code, car il s'occupe de construire un formulaire avec tous les champs et attributs appropriés à partir d'une classe `Model`.

Instances de formulaires liées et non liées

La distinction entre [Formulaires liés et non liés](#) est importante :

- Un formulaire non renseigné n'a aucune donnée associée. Lorsqu'il est présenté à l'utilisateur, il sera vide ou ne contiendra que des valeurs par défaut.
- Un formulaire renseigné contient des données envoyées et il est donc possible de lui demander si ces données sont valides. Si un formulaire renseigné non valide est affiché, il peut contenir des messages d'erreur intégrés indiquant à l'utilisateur quelles sont les données à corriger.

L'attribut [is_bound](#) d'un formulaire indique si des données ont été liées au formulaire ou pas.

Plus de détails sur les champs

Voici un formulaire un peu plus utile que notre exemple minimal ci-dessus, que nous pourrions utiliser pour implémenter une fonctionnalité « Contactez-moi » sur un site Web personnel :

forms.py

```

from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)

```

Le formulaire précédent comportait un seul champ, `your_name`, de type [CharField](#). Dans ce cas, le formulaire possède quatre champs : `subject`, `message`, `sender` et `cc_myself`. [CharField](#), [EmailField](#) et [BooleanField](#) ne sont que trois des types de champs disponibles ; une liste complète se trouve dans [Champs de formulaires](#).

Composants de formulaires (« widgets »)

À chaque champ de formulaire correspond une [classe Widget](#), qui elle-même se réfère à un composant de formulaire HTML comme par exemple `<input type="text">`.

Dans la plupart des cas, le composant par défaut du champ convient bien. Par exemple, le composant par défaut du champ [CharField](#) est [TextInput](#), qui produit une balise `<input type="text">` en HTML. Si au contraire vous aviez souhaité une balise `<textarea>`, il aurait fallu définir le composant approprié lors de la définition du champ de formulaire, comme nous l'avons fait pour le champ `message`.

Données de champ

Quelles que soient les données envoyées avec un formulaire, au moment où elles ont été validées avec succès suite à l'appel de `is_valid()` (et que `is_valid()` a renvoyé `True`), les données de formulaire validées se trouvent dans le dictionnaire `form.cleaned_data`. Ces données auront été gracieusement converties en types Python pour vous.

Note

À ce stade, vous pouvez toujours accéder directement aux données non validées dans `request.POST`, mais les données validées sont plus adéquates.

Dans l'exemple ci-dessus du formulaire de contact, `cc_myself` sera une valeur booléenne. De la même manière, des champs de type [IntegerField](#) et [FloatField](#) convertissent les valeurs en types Python `int` et `float`, respectivement.

Voici comment les données de formulaire pourraient être traitées dans la vue qui gère ce formulaire :

`views.py`

```

from django.core.mail import send_mail

if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']

```



```
cc_myself = form.cleaned_data['cc_myself']

recipients = ['info@example.com']
if cc_myself:
    recipients.append(sender)

send_mail(subject, message, sender, recipients)
return HttpResponseRedirect('/thanks/')
```

Astuce

Pour en savoir plus sur l'envoi de courriels à partir de Django, consultez [Envoi de messages électroniques](#).

Certains types de champ ont besoin de traitement supplémentaire. Par exemple, les fichiers téléversés via un formulaire doivent être traités différemment (ils sont accessibles par `request.FILES`, et non pas par `request.POST`). Pour plus de détails sur la façon de gérer des téléversements de fichiers avec un formulaire, consultez [Liaison de fichiers téléversés avec un formulaire](#).

Formulaires et gabarits

Tout ce qui est nécessaire pour qu'un formulaire soit inclus dans un gabarit est de placer l'instance de formulaire dans le contexte du gabarit. Ainsi, si le formulaire est appelé `form` dans le contexte, `{{ form }}` suffira à afficher ses éléments `<label>` et `<input>` de manière appropriée.

Options d'affichage des formulaires

Compléments HTML pour les formulaires dans les gabarits

N'oubliez pas que le rendu HTML d'un formulaire n'inclut *pas* la balise `<form>` englobante, ni le composant `submit` pour l'envoi du formulaire. C'est à vous d'ajouter ces éléments.

Il existe toutefois d'autres options pour les paires `<label>/<input>`:

- `{{ form.as_table }}` affiche les composants sous forme de cellules de tableau à l'intérieur de balises `<tr>`
- `{{ form.as_p }}` affiche les composants dans des balises `<p>`
- `{{ form.as_ul }}` affiche les composants dans des balises ``

Notez que vous devrez ajouter vous-même les éléments `<table>` ou `` autour du contenu produit.

Voici ce qu'affiche `{{ form.as_p }}` pour notre instance de `ContactForm`:

```
<p><label for="id_subject">Subject:</label>
  <input id="id_subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="id_message">Message:</label>
  <textarea name="message" id="id_message"></textarea></p>
<p><label for="id_sender">Sender:</label>
  <input type="email" name="sender" id="id_sender" /></p>
```

```
<p><label for="id_cc_myself">Cc myself:</label>
  <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
```

Notez que chaque champ de formulaire reçoit un attribut ID défini à `id_<nom-de-champ>` qui est référencé par la balise `label` correspondante. C'est important pour garantir que les formulaires sont accessibles aux aides techniques du genre logiciel de lecture d'écran. Vous pouvez aussi [personnaliser la façon dont les éléments label et id sont générés](#).

Voir [Affichage des formulaires en HTML](#) pour plus d'informations à ce sujet.

Affichage manuel des champs

Laisser Django afficher un à un les champs du formulaire n'est pas la seule possibilité ; on peut tout à fait afficher soi-même les champs (par exemple pour changer leur ordre d'apparition). Chaque champ est accessible en tant qu'attribut du formulaire avec la syntaxe `{{ form.nom_du_champ }}`, ce qui, dans un gabarit Django, produira son affichage de manière appropriée. Par exemple :

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email subject:</label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">
    {{ form.message.errors }}
    <label for="{{ form.message.id_for_label }}">Your message:</label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email address:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC yourself?</label>
    {{ form.cc_myself }}
</div>
```

Une balise `<label>` complète peut aussi être générée au moyen de [label_tag\(\)](#). Par exemple :

```
<div class="fieldWrapper">
    {{ form.subject.errors }}
    {{ form.subject.label_tag }}
    {{ form.subject }}
</div>
```

Affichage des messages d'erreur de formulaires

Naturellement, le prix de cette flexibilité est un plus grand effort. Jusqu'à ce stade, nous n'avions pas à nous préoccuper de l'affichage des erreurs du formulaire, car cela se faisait automatiquement. Dans cet exemple, il a fallu s'assurer que les erreurs de chaque champ et les erreurs globales du formulaire

soient effectivement affichées. Remarquez `{{ form.non_field_errors }}` au sommet du formulaire ainsi que l'expression de gabarit concernant les erreurs de chaque champ.

La syntaxe `{{ form.nom_du_champ.errors }}` affiche une liste des erreurs du formulaire, sous forme de liste non ordonnée. Cela pourrait ressembler à ceci :

```
<ul class="errorlist">
  <li>Sender is required.</li>
</ul>
```

La liste possède une classe CSS `errorlist` pour vous permettre de mettre en forme son apparence. Si vous souhaitez personnaliser davantage l'affichage des erreurs, vous pouvez le faire par une boucle :

```
{% if form.subject.errors %}
  <ol>
    {% for error in form.subject.errors %}
      <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
  </ol>
{% endif %}
```

Les erreurs non liées à un champ (ou les erreurs de champs cachés qui sont affichées au sommet du formulaire lorsqu'on utilise des utilitaires comme `form.as_p()`) seront dotées d'une classe supplémentaire `nonfield` pour aider à faire la distinction avec les erreurs spécifiques à un champ. Par exemple, `{{ form.non_field_errors }}` ressemblerait à :

```
<ul class="errorlist nonfield">
  <li>Generic validation error</li>
</ul>
```

Changed in Django 1.8:

La classe `nonfield` telle que décrite dans l'exemple ci-dessus a été ajoutée.

Voir [L'API des formulaires](#) pour plus de détails sur les erreurs, la mise en forme et la manipulation des attributs de formulaire dans les gabarits.

Boucles sur champs de formulaires

Si vous utilisez le même code HTML pour tous vos champs de formulaire, vous pouvez réduire la duplication de code en effectuant une boucle sur chaque champ en utilisant l'opérateur `{% for %}`:

```
{% for field in form %}
  <div class="fieldWrapper">
    {{ field.errors }}
    {{ field.label_tag }} {{ field }}
    {% if field.help_text %}
      <p class="help">{{ field.help_text|safe }}</p>
    {% endif %}
  </div>
{% endfor %}
```

Voici quelques attributs de champ utiles :

```
{{ field.label }}
```

L'étiquette du champ, par exemple Adresse de courriel.

```
{{ field.label_tag }}
```

L'intitulé du champ placé dans la balise HTML `<label>` appropriée. Cela comprend le paramètre [`label_suffix`](#) du formulaire. Par exemple, la valeur `label_suffix` par défaut est un caractère deux-points :

```
<label for="id_email">Email address:</label>
```

```
{{ field.id_for_label }}
```

L'attribut `id` utilisé pour ce champ (`id_email` dans l'exemple ci-dessus). Si vous construisez vous-même l'étiquette du champ, il peut être avantageux d'utiliser ceci à la place de `label_tag`. C'est aussi utile par exemple si vous avez du code JavaScript en ligne et que vous vouliez éviter de figer l'identifiant du champ.

```
{{ field.value }}
```

La valeur du champ, par exemple `quelqu-un@example.com`.

```
{{ field.html_name }}
```

Le nom du champ tel qu'il sera utilisé dans le nom de champ de la balise `input`. Il prend en compte le préfixe de formulaire, si celui-ci est défini.

```
{{ field.help_text }}
```

Tout texte d'aide associé au champ.

```
{{ field.errors }}
```

Affiche une liste `<ul class="errorlist">` contenant toute erreur de validation correspondant à ce champ. Vous pouvez personnaliser la présentation des erreurs avec une boucle `{% for error in field.errors %}`. Dans ce cas, chaque objet de la boucle est une simple chaîne de caractères contenant le message d'erreur.

```
{{ field.is_hidden }}
```

Cet attribut vaut `True` si le champ de formulaire est un champ masqué, sinon il vaut `False`. Ce n'est pas particulièrement utile comme variable de gabarit, mais pourrait être utile dans des tests conditionnels tels que :

```
{% if field.is_hidden %}
```

```

    {%# Do something special %}
{% endif %}

{{ field.field }}

```

L'instance [Field](#) de la classe de formulaire que cet objet [BoundField](#) adapte. Vous pouvez l'utiliser pour accéder aux attributs de [Field](#), par exemple `{{ char_field.field.max_length }}`.

Voir aussi

Pour une liste complète des attributs et méthodes, voir [BoundField](#).

Boucles sur les champs masqués et visibles

Si vous affichez manuellement un formulaire dans un gabarit, sans faire appel à l'affichage par défaut des formulaires de Django, il peut être nécessaire de traiter différemment les champs `<input type="hidden">` des autres champs non masqués. Par exemple, comme les champs masqués ne produisent pas de contenu visible, l'affichage de messages d'erreur « à côté » du champ pourrait générer de la confusion pour les utilisateurs ; il faut donc gérer différemment les erreurs de ce type de champ.

Django fournit deux méthodes qui permettent de tourner en boucle indépendamment sur les champs visibles et masqués : `hidden_fields()` et `visible_fields()`. Voici une modification d'un exemple précédent en utilisant ces deux méthodes :

```

{%# Include the hidden fields %}
{% for hidden in form.hidden_fields %}
{{ hidden }}
{% endfor %}
{%# Include the visible fields %}
{% for field in form.visible_fields %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}

```

Cet exemple ne gère pas du tout les erreurs des champs masqués. En principe, une erreur dans un champ masqué est un signe de manipulation abusive d'un formulaire, puisque l'interaction normale avec un formulaire ne les modifie pas. Cependant, vous pourriez facilement ajouter aussi une forme d'affichage pour ces erreurs de formulaire.

Gabarits de formulaire réutilisables

Si votre site utilise la même logique d'affichage des formulaires à plusieurs endroits, vous pouvez réduire la duplication en enregistrant la boucle de formulaire dans un gabarit autonome et en employant la balise [include](#) afin de réutiliser ce contenu dans d'autres gabarits :

```
# In your form template:
```

```
{% include "form_snippet.html" %}

# In form_snippet.html:
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

Si l'objet formulaire transmis au gabarit possède un nom différent dans le contexte, vous pouvez lui donner un alias en utilisant le paramètre `with` de la balise [include](#):

```
{% include "form_snippet.html" with form=comment_form %}
```

Si vous constatez que vous reproduisez souvent ce même code, il vous faut peut-être envisager la création d'une [balise d'inclusion](#) personnalisée.

Sujets complémentaires

Cette page aborde les généralités, mais les formulaires peuvent faire bien d'autres choses encore :

- [Formulaires groupés](#)
 - [Données initiales pour les formulaires groupés](#)
 - [Restriction du nombre maximum de formulaires](#)
 - [Validation des formulaires groupés](#)
 - [Validation du nombre de formulaires dans les formulaires groupés](#)
 - [Tri et suppression de formulaires](#)
 - [Ajout de champs supplémentaires à des formulaires groupés](#)
 - [Transmission de paramètres personnalisés aux formulaires de jeux de formulaires](#)
 - [Utilisation des formulaires groupés dans les vues et les gabarits](#)
- [Création de formulaires à partir de modèles](#)
 - [ModelForm](#)
 - [Formulaires groupés de modèles](#)
 - [Sous-formulaires groupés](#)
- [Fichiers annexes de formulaire \(classe Media\)](#)
 - [Fichiers annexes définis statiquement](#)
 - [Media comme propriété dynamique](#)
 - [Chemins dans les définitions de fichiers annexes](#)
 - [Objets Media](#)
 - [Media pour les formulaires](#)

Voir aussi

[La référence des formulaires](#)

Contient la référence d'API complète, y compris les champs de formulaire, les composants de formulaire et la validation des champs et des formulaires.

[Utilisation des sessions](#)
[Formulaires groupés](#)

Utilisation des formulaires

À propos de ce document

Ce document présente une introduction aux bases des formulaires web et à la manière dont ils sont traités par Django. Pour plus de détails sur des aspects spécifiques de l'API des formulaires, consultez [L'API des formulaires](#), [Champs de formulaires](#), et [La validation de formulaires et de champs](#).

À moins que vous n'ayez l'intention de construire des sites Web et des applications qui ne font rien d'autre que de publier du contenu, et n'acceptant aucune saisie de la part de ses visiteurs, vous allez avoir besoin de comprendre et d'utiliser des formulaires.

Django fournit un certain nombre d'outils et de bibliothèques pour vous aider à créer des formulaires de saisie pour les visiteurs du site, puis pour aider à traiter et à répondre à ces saisies.

Formulaires HTML

En HTML, un formulaire est un ensemble d'éléments à l'intérieur des balises `<form>...</form>` qui permettent à un visiteur d'effectuer des actions comme saisir du texte, sélectionner des options, manipuler des objets ou des contrôles, et ainsi de suite, puis d'envoyer ces informations au serveur.

Certains de ces éléments d'interface de formulaires - boîtes de saisie de texte ou cases à cocher - sont assez simples et intégrés dans le HTML lui-même. D'autres éléments sont beaucoup plus complexes ; une interface qui affiche un sélecteur de date ou qui permet de déplacer un curseur ou de contrôler un autre élément graphique utilise généralement du code JavaScript et CSS, ainsi que des éléments `<input>` de formulaires HTML pour produire ce genre d'effets.

En plus de ses éléments `<input>`, un formulaire doit préciser deux choses :

- *où* : l'URL vers laquelle les données correspondant à la saisie de l'utilisateur doivent être renvoyées
- *comment* : la méthode HTTP utilisée pour renvoyer les données

Par exemple, le formulaire de connexion pour l'administration de Django contient plusieurs éléments `<input>`: un de `type="text"` pour le nom d'utilisateur, un de `type="password"` pour le mot de passe, et un de `type="submit"` pour le bouton « Connexion ». Il contient également des champs

de texte cachés que l'utilisateur ne voit pas, mais que Django utilise pour déterminer ce qu'il faut faire après la connexion.

Il indique également au navigateur que les données du formulaire doivent être envoyées à l'URL spécifiée dans l'attribut `action` de la balise `<form>` (`/admin/`) et qu'elles doivent être envoyées en utilisant le mécanisme HTTP spécifié par l'attribut `method` (`post`).

Lorsque le bouton `<input type="submit" value="Connexion">` est déclenché, les données sont renvoyées à `/admin/`.

GET et POST

GET et POST sont les seules méthodes HTTP à utiliser dans les formulaires.

Le formulaire de connexion de Django est renvoyé en utilisant la méthode POST, par laquelle le navigateur regroupe les données du formulaire, les encode pour la transmission, les envoie au serveur, puis reçoit la réponse en retour.

GET, en revanche, regroupe les données fournies dans une chaîne et l'utilise pour composer une URL. L'URL contient l'adresse à laquelle les données doivent être envoyées, ainsi que les clés et les valeurs fournies. Vous pouvez en voir un exemple si vous effectuez une recherche dans la documentation de Django, car cela va produire une URL de la forme `https://docs.djangoproject.com/search/?q=forms&release=1`.

GET et POST sont généralement utilisées à des fins différentes.

Toute requête pouvant être utilisée pour modifier l'état du système, par exemple une requête qui applique des changements en base de données, devrait utiliser POST. GET ne devrait être employée que pour des requêtes qui n'affectent pas l'état du système.

GET ne conviendrait pas non plus pour un formulaire de changement de mot de passe, car le mot de passe apparaîtrait en clair dans l'URL ainsi que dans l'historique du navigateur et les journaux du serveur. Elle ne conviendrait pas non plus pour des données volumineuses ou pour des données binaires telles que pour une image. Une application Web utilisant des requêtes GET pour des formulaires administratifs ne serait pas sécurisée : il est facile pour un attaquant de simuler une requête de formulaire pour obtenir l'accès à des parties sensibles du système. Les requêtes POST accompagnées par d'autres protections comme la [protection CSRF](#) de Django offrent un bien meilleur contrôle sur les accès.

D'autre part, GET convient pour des interfaces comme un formulaire de recherche Web, car les URL qui représentent une requête GET peuvent être facilement conservées dans un signet, partagées ou ré-envoyées.

Le rôle de Django dans les formulaires

La gestion des formulaires est une affaire complexe. Considérez l'application d'administration de Django où de nombreuses données de différents types doivent être préparées pour être affichées dans un formulaire, produites en HTML, éditées par une interface conviviale, renvoyées vers le serveur, validées et nettoyées, et finalement enregistrées ou transmises pour traitement ultérieur.

La fonctionnalité des formulaires de Django peut simplifier et automatiser de larges portions de ce travail et peut aussi le faire de manière plus sécurisée que la plupart des programmeurs ne pourraient le faire en écrivant leur propre code.

Django gère trois parties distinctes du travail induit par les formulaires :

- préparation et restructuration des données en vue de leur présentation
- création des formulaires HTML pour les données
- réception et traitement des formulaires et des données envoyés par le client

Il est *possible* d'écrire du code qui fait tout cela manuellement, mais Django peut s'en charger à votre place.

Les formulaires dans Django

Nous avons brièvement décrit les formulaires HTML, mais un objet HTML `<form>` n'est qu'un élément de la machinerie nécessaire.

Dans le contexte d'une application Web, « formulaire » peut se référer à cette balise HTML `<form>`, à la classe [Form](#) de Django qui la produit, aux données structurées renvoyées lorsque le formulaire est soumis, ou encore au fonctionnement de bout en bout de ces différentes parties.

La classe [Form](#) de Django

La classe [Form](#) de Django se situe au cœur de ce système de composants. De la même façon qu'un modèle Django décrit la structure logique d'un objet, son comportement et la manière dont ses parties nous sont présentées, une classe [Form](#) décrit un formulaire et détermine son fonctionnement et son apparence.

De la même façon que les champs d'une classe de modèle correspondent à des champs de base de données, les champs d'une classe de formulaire correspondent aux éléments `<input>` d'un formulaire HTML (un [ModelForm](#) fait correspondre les champs d'une classe de modèle à des éléments `<input>` d'un formulaire HTML via un formulaire [Form](#); cela constitue la base de l'interface d'administration de Django).

Les champs d'un formulaire sont eux-mêmes des classes ; elles gèrent les données de formulaire et s'occupent de la validation lorsque des données de formulaires sont reçues. Un champ [DateField](#) et

un champ [FileField](#) gèrent des types de données très différents et doivent procéder à des actions différentes.

Un champ de formulaire est présenté dans une interface de navigateur sous forme de « composant » HTML, un élément d'interface utilisateur. Chaque type de champ contient une [classe Widget](#) appropriée par défaut, mais celles-ci peuvent être remplacées au besoin.

Instanciation, traitement et rendu des formulaires

Lors du rendu d'un objet Django pour son affichage, il s'agit généralement de :

1. obtenir l'élément dans la vue (le récupérer à partir de la base de données, par exemple)
2. le transmettre au contexte de gabarit
3. le transformer en balisage HTML en utilisant des variables de gabarit

L'affichage d'un formulaire dans un gabarit suit à peu près le même processus que pour tout autre type d'objet, mais il y a tout de même certaines différences notables.

Dans le cas d'une instance de modèle ne contenant aucune donnée, il n'y aura pas beaucoup de sens d'en faire quoi que ce soit dans un gabarit. En revanche, il est totalement raisonnable de vouloir afficher un formulaire vierge, c'est ce qu'il faut faire lorsqu'on veut que quelqu'un le remplisse.

Ainsi, lorsque nous manipulons une instance de modèle dans une vue, nous la récupérons typiquement à partir de la base de données. Lorsque nous manipulons un formulaire, nous en créons typiquement une instance dans la vue.

Lors de la création d'une instance de formulaire, on peut choisir de le laisser vide ou de le pré-remplir, par exemple avec :

- des données provenant d'une instance de modèle enregistrée (comme dans le cas des formulaires d'administration pour l'édition)
- des données que nous avons obtenues à partir d'autres sources
- des données reçues d'un envoi de formulaire HTML

Le dernier de ces cas est le plus intéressant, car c'est ce qui permet aux utilisateurs de ne pas seulement lire un site Web, mais aussi de renvoyer des informations aux sites Web.

Construction d'un formulaire

Le travail à effectuer

Supposons que vous vouliez créer un formulaire simple sur votre site dans le but d'obtenir le nom de l'utilisateur. Voici ce qu'il faudrait obtenir dans le navigateur :

```
<form action="/your-name/" method="post">
  <label for="your_name">Your name: </label>
```

```
<input id="your_name" type="text" name="your_name" value="{{ current_name }}">
<input type="submit" value="OK">
</form>
```

Le navigateur saura alors qu'il doit renvoyer les données du formulaire vers l'URL `/your-name/`, en utilisant la méthode `POST`. Il affichera un champ de texte libellé « Your name: » et un bouton intitulé « OK ». Si le contexte de gabarit contient une variable `current_name`, celle-ci sera utilisée pour pré-remplir le champ `your_name`.

Il faudra une vue qui produit le gabarit contenant le formulaire HTML et qui puisse fournir le champ `current_name` si nécessaire.

Lorsque le formulaire aura été envoyé, la requête `POST` envoyée au serveur contiendra les données du formulaire.

Mais il sera également nécessaire d'avoir une vue répondant à l'URL `/your-name/` qui se chargera de trouver les paires clé-valeur dans la requête, puis de les traiter.

Il s'agit d'un formulaire très simple. En pratique, un formulaire peut contenir des dizaines ou centaines de champs, dont beaucoup peuvent être pré-remplis, et il est possible que l'utilisateur doive passer par plusieurs cycles d'envoi/édition avant que l'opération globale ne soit terminée.

On peut attendre du navigateur qu'il s'occupe d'une partie de la validation, même avant que le formulaire ne soit envoyé. Peut-être que certains champs seront bien plus complexes, permettant par exemple de choisir des dates dans un calendrier.

Compte tenu de tout ceci, il est bien plus avantageux que Django se charge lui-même de la plupart du travail.

Construction d'un formulaire dans Django

La classe [Form](#)

Nous savons déjà à quoi notre formulaire HTML doit ressembler. Notre point de départ dans Django est le suivant :

```
forms.py
```

```
from django import forms
```

```
class NameForm(forms.Form):
    your_name = forms.CharField(label='Your name', max_length=100)
```

Cela définit une classe [Form](#) comportant un seul champ (`your_name`). Nous avons attribué une étiquette conviviale à ce champ ; celle-ci apparaîtra dans la balise `<label>` au moment de l'affichage (bien que dans ce cas, le contenu indiqué dans [label](#) est en réalité identique à celui qui aurait été généré automatiquement si nous n'avions rien fait).

La longueur maximale du champ est définie par [max_length](#). Cela fait deux choses : la balise HTML `<input>` reçoit l'attribut `maxlength="100"` (afin que le navigateur empêche lui-même la saisie d'un plus grand nombre de caractères), et ultérieurement lorsque Django reçoit en retour les données de formulaire, il valide que les données respectent cette longueur.

Une instance de [Form](#) possède une méthode [is_valid\(\)](#) qui procède aux routines de validation pour tous ses champs. Lorsque la méthode est appelée et que tous les champs contiennent des données valables, celle-ci :

- renvoie `True`;
- insère les données du formulaire dans l'attribut [cleaned_data](#).

Le formulaire complet, lorsqu'il est affiché pour la première fois, ressemble à ceci :

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100">
```

Notez qu'il n'inclut **pas** les balises `<form>` ni de bouton d'envoi. Ces éléments doivent être ajoutés par celui qui rédige le gabarit.

La vue

Les données de formulaire renvoyés à un site Web Django sont traitées par une vue, en principe la même qui a servi à produire le formulaire. Cela permet de réutiliser une partie de la même logique.

Pour traiter le formulaire, nous devons en créer une instance dans la vue à destination de l'URL à laquelle il doit apparaître :

views.py

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

from .forms import NameForm

def get_name(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the request:
        form = NameForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required
            # ...
            # redirect to a new URL:
            return HttpResponseRedirect('/thanks/')

    # if a GET (or any other method) we'll create a blank form
    else:
        form = NameForm()

    return render(request, 'name.html', {'form': form})
```

Si nous arrivons dans cette vue avec une requête **GET**, la vue crée une instance de formulaire vierge et la place dans le contexte de gabarit en vue de son affichage. C'est à quoi l'on peut s'attendre lors du premier accès à l'URL en question.

Si le formulaire est envoyé par une requête **POST**, la vue crée également une instance de formulaire et la complète avec les données reçues à partir de la requête : `form = NameForm(request.POST)`. C'est ce qu'on appelle « lier les données au formulaire » (il s'agit maintenant d'un formulaire *lié*, « bound » en anglais).

La méthode `is_valid()` est appelée ; si elle ne renvoie pas `True`, on se retrouve au stade du gabarit contenant le formulaire. Mais cette fois, le formulaire n'est plus vierge (*non lié* ou *unbound*) ce qui fait que le formulaire HTML sera rempli avec les données saisies précédemment, données qui peuvent être modifiées et corrigées selon les besoins.

Si `is_valid()` renvoie `True`, toutes les données validées du formulaire sont alors accessibles dans l'attribut `cleaned_data`. Ces données peuvent être utilisées pour mettre à jour la base de données ou effectuer d'autres opérations avant de renvoyer au navigateur une redirection HTTP lui indiquant l'URL à recharger.

Le gabarit

Il n'y a pas grand chose à faire dans le gabarit `name.html`. L'exemple le plus simple est :

```
<form action="/your-name/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit" />
</form>
```

Tous les champs de formulaire et leurs attributs seront convertis en balises HTML à partir de `{{ form }}` par le langage de gabarit de Django.

Formulaires et protection contre le « Cross site request forgery » (CSRF)

Django est livré avec une protection simple d'emploi [contre les attaques de type Cross Site Request Forgeries](#). Lors de l'envoi d'un formulaire par la méthode **POST** et la protection CSRF active, vous devez utiliser la balise de gabarit `csrf_token` comme dans l'exemple précédent. Cependant, comme la protection CSRF n'est pas directement liée aux formulaires dans les gabarits, cette balise est omise dans les exemples suivants de ce document.

Types de composants HTML5 et validation par le navigateur

Si un formulaire contient un champ de type `URLField`, `EmailField` ou tout type de champ nombre entier, Django fait appel aux composants HTML5 (input) de type `url`, `email` et `number`. Par défaut, les navigateurs appliquent souvent leur propre validation de ces champs, qui peut être plus stricte que celle de Django. Si vous voulez échapper à cette validation, ajoutez l'attribut `novalidate` sur la

balise `form` ou définissez un autre composant de formulaire pour le champ, comme par exemple [TextInput](#).

Nous possédons maintenant un formulaire Web fonctionnel, défini par une instance [Form](#) de Django, traité par une vue et affiché sous forme de balise HTML `<form>`.

C'est tout ce dont vous avez besoin pour commencer, mais l'infrastructure des formulaires a beaucoup plus à vous offrir. À partir du moment où vous comprenez les bases du processus décrit ci-dessus, il est bon de connaître les autres fonctionnalités mises à disposition par le système des formulaires et d'en savoir un peu plus au sujet de la machinerie sous-jacente.

Plus de détails sur les classes [Form](#) de Django

Toutes les classes de formulaires héritent de [django.forms.Form](#), y compris les classes [ModelForm](#) que vous pouvez rencontrer dans l'administration de Django.

Modèles et formulaires

En fait, si un formulaire est destiné à ajouter ou modifier directement un modèle Django, un formulaire [ModelForm](#) peut vous économiser beaucoup de temps, d'effort et de code, car il s'occupe de construire un formulaire avec tous les champs et attributs appropriés à partir d'une classe `Model`.

Instances de formulaires liées et non liées

La distinction entre [Formulaires liés et non liés](#) est importante :

- Un formulaire non renseigné n'a aucune donnée associée. Lorsqu'il est présenté à l'utilisateur, il sera vide ou ne contiendra que des valeurs par défaut.
- Un formulaire renseigné contient des données envoyées et il est donc possible de lui demander si ces données sont valides. Si un formulaire renseigné non valide est affiché, il peut contenir des messages d'erreur intégrés indiquant à l'utilisateur quelles sont les données à corriger.

L'attribut [is_bound](#) d'un formulaire indique si des données ont été liées au formulaire ou pas.

Plus de détails sur les champs

Voici un formulaire un peu plus utile que notre exemple minimal ci-dessus, que nous pourrions utiliser pour implémenter une fonctionnalité « Contactez-moi » sur un site Web personnel :

forms.py

```
from django import forms
```

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

Le formulaire précédent comportait un seul champ, `your_name`, de type [CharField](#). Dans ce cas, le formulaire possède quatre champs : `subject`, `message`, `sender` et `cc_myself`. [CharField](#), [EmailField](#) et [BooleanField](#) ne sont que trois des types de champs disponibles ; une liste complète se trouve dans [Champs de formulaires](#).

Composants de formulaires (« widgets »)

À chaque champ de formulaire correspond une [classe Widget](#), qui elle-même se réfère à un composant de formulaire HTML comme par exemple `<input type="text">`.

Dans la plupart des cas, le composant par défaut du champ convient bien. Par exemple, le composant par défaut du champ [CharField](#) est [TextInput](#), qui produit une balise `<input type="text">` en HTML. Si au contraire vous aviez souhaité une balise `<textarea>`, il aurait fallu définir le composant approprié lors de la définition du champ de formulaire, comme nous l'avons fait pour le champ `message`.

Données de champ

Quelles que soient les données envoyées avec un formulaire, au moment où elles ont été validées avec succès suite à l'appel de `is_valid()` (et que `is_valid()` a renvoyé `True`), les données de formulaire validées se trouvent dans le dictionnaire `form.cleaned_data`. Ces données auront été gracieusement converties en types Python pour vous.

Note

À ce stade, vous pouvez toujours accéder directement aux données non validées dans `request.POST`, mais les données validées sont plus adéquates.

Dans l'exemple ci-dessus du formulaire de contact, `cc_myself` sera une valeur booléenne. De la même manière, des champs de type [IntegerField](#) et [FloatField](#) convertissent les valeurs en types Python `int` et `float`, respectivement.

Voici comment les données de formulaire pourraient être traitées dans la vue qui gère ce formulaire :

`views.py`

```
from django.core.mail import send_mail

if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']

    recipients = ['info@example.com']
    if cc_myself:
        recipients.append(sender)

    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect('/thanks/')
```

Astuce

Pour en savoir plus sur l'envoi de courriels à partir de Django, consultez [Envoi de messages électroniques](#).

Certains types de champ ont besoin de traitement supplémentaire. Par exemple, les fichiers téléversés via un formulaire doivent être traités différemment (ils sont accessibles par `request.FILES`, et non pas par `request.POST`). Pour plus de détails sur la façon de gérer des téléversements de fichiers avec un formulaire, consultez [Liaison de fichiers téléversés avec un formulaire](#).

Formulaires et gabarits

Tout ce qui est nécessaire pour qu'un formulaire soit inclus dans un gabarit est de placer l'instance de formulaire dans le contexte du gabarit. Ainsi, si le formulaire est appelé `form` dans le contexte, `{{ form }}` suffira à afficher ses éléments `<label>` et `<input>` de manière appropriée.

Options d'affichage des formulaires

Compléments HTML pour les formulaires dans les gabarits

N'oubliez pas que le rendu HTML d'un formulaire n'inclut *pas* la balise `<form>` englobante, ni le composant `submit` pour l'envoi du formulaire. C'est à vous d'ajouter ces éléments.

Il existe toutefois d'autres options pour les paires `<label>/<input>`:

- `{{ form.as_table }}` affiche les composants sous forme de cellules de tableau à l'intérieur de balises `<tr>`
- `{{ form.as_p }}` affiche les composants dans des balises `<p>`
- `{{ form.as_ul }}` affiche les composants dans des balises ``

Notez que vous devrez ajouter vous-même les éléments `<table>` ou `` autour du contenu produit.

Voici ce qu'affiche `{{ form.as_p }}` pour notre instance de `ContactForm`:

```
<p><label for="id_subject">Subject:</label>
  <input id="id_subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="id_message">Message:</label>
  <textarea name="message" id="id_message"></textarea></p>
<p><label for="id_sender">Sender:</label>
  <input type="email" name="sender" id="id_sender" /></p>
<p><label for="id_cc_myself">Cc myself:</label>
  <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
```

Notez que chaque champ de formulaire reçoit un attribut ID défini à `id_<nom-de-champ>` qui est référencé par la balise `label` correspondante. C'est important pour garantir que les formulaires sont accessibles aux aides techniques du genre logiciel de lecture d'écran. Vous pouvez aussi [personnaliser la façon dont les éléments label et id sont générés](#).

Voir [Affichage des formulaires en HTML](#) pour plus d'informations à ce sujet.

Affichage manuel des champs

Laisser Django afficher un à un les champs du formulaire n'est pas la seule possibilité ; on peut tout à fait afficher soi-même les champs (par exemple pour changer leur ordre d'apparition). Chaque champ est accessible en tant qu'attribut du formulaire avec la syntaxe `{{ form.nom_du_champ }}`, ce qui, dans un gabarit Django, produira son affichage de manière appropriée. Par exemple :

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email subject:</label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">
    {{ form.message.errors }}
    <label for="{{ form.message.id_for_label }}">Your message:</label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email address:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC yourself?</label>
    {{ form.cc_myself }}
</div>
```

Une balise `<label>` complète peut aussi être générée au moyen de [label_tag\(\)](#). Par exemple :

```
<div class="fieldWrapper">
    {{ form.subject.errors }}
    {{ form.subject.label_tag }}
    {{ form.subject }}
</div>
```

Affichage des messages d'erreur de formulaires

Naturellement, le prix de cette flexibilité est un plus grand effort. Jusqu'à ce stade, nous n'avions pas à nous préoccuper de l'affichage des erreurs du formulaire, car cela se faisait automatiquement. Dans cet exemple, il a fallu s'assurer que les erreurs de chaque champ et les erreurs globales du formulaire soient effectivement affichées. Remarquez `{{ form.non_field_errors }}` au sommet du formulaire ainsi que l'expression de gabarit concernant les erreurs de chaque champ.

La syntaxe `{{ form.nom_du_champ.errors }}` affiche une liste des erreurs du formulaire, sous forme de liste non ordonnée. Cela pourrait ressembler à ceci :

```
<ul class="errorlist">
    <li>Sender is required.</li>
</ul>
```

La liste possède une classe CSS `errorlist` pour vous permettre de mettre en forme son apparence. Si vous souhaitez personnaliser davantage l’affichage des erreurs, vous pouvez le faire par une boucle :

```
{% if form.subject.errors %}
    <ol>
        {% for error in form.subject.errors %}
            <li><strong>{{ error|escape }}</strong></li>
        {% endfor %}
    </ol>
{% endif %}
```

Les erreurs non liées à un champ (ou les erreurs de champs cachés qui sont affichées au sommet du formulaire lorsqu’on utilise des utilitaires comme `form.as_p()`) seront dotées d’une classe supplémentaire `nonfield` pour aider à faire la distinction avec les erreurs spécifiques à un champ. Par exemple, `{{ form.non_field_errors }}` ressemblerait à :

```
<ul class="errorlist nonfield">
    <li>Generic validation error</li>
</ul>
```

Changed in Django 1.8:

La classe `nonfield` telle que décrite dans l’exemple ci-dessus a été ajoutée.

Voir [L’API des formulaires](#) pour plus de détails sur les erreurs, la mise en forme et la manipulation des attributs de formulaire dans les gabarits.

Boucles sur champs de formulaires

Si vous utilisez le même code HTML pour tous vos champs de formulaire, vous pouvez réduire la duplication de code en effectuant une boucle sur chaque champ en utilisant l’opérateur `{% for %}`:

```
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
        {% if field.help_text %}
            <p class="help">{{ field.help_text|safe }}</p>
        {% endif %}
    </div>
{% endfor %}
```

Voici quelques attributs de champ utiles :

```
{{ field.label }}
```

L’étiquette du champ, par exemple Adresse de courriel.

```
{{ field.label_tag }}
```

L'intitulé du champ placé dans la balise HTML `<label>` appropriée. Cela comprend le paramètre `label_suffix` du formulaire. Par exemple, la valeur `label_suffix` par défaut est un caractère deux-points :

```
<label for="id_email">Email address:</label>
```

```
{{ field.id_for_label }}
```

L'attribut `id` utilisé pour ce champ (`id_email` dans l'exemple ci-dessus). Si vous construisez vous-même l'étiquette du champ, il peut être avantageux d'utiliser ceci à la place de `label_tag`. C'est aussi utile par exemple si vous avez du code JavaScript en ligne et que vous vouliez éviter de figer l'identifiant du champ.

```
{{ field.value }}
```

La valeur du champ, par exemple `quelqu-un@example.com`.

```
{{ field.html_name }}
```

Le nom du champ tel qu'il sera utilisé dans le nom de champ de la balise `input`. Il prend en compte le préfixe de formulaire, si celui-ci est défini.

```
{{ field.help_text }}
```

Tout texte d'aide associé au champ.

```
{{ field.errors }}
```

Affiche une liste `<ul class="errorlist">` contenant toute erreur de validation correspondant à ce champ. Vous pouvez personnaliser la présentation des erreurs avec une boucle `{% for error in field.errors %}`. Dans ce cas, chaque objet de la boucle est une simple chaîne de caractères contenant le message d'erreur.

```
{{ field.is_hidden }}
```

Cet attribut vaut `True` si le champ de formulaire est un champ masqué, sinon il vaut `False`. Ce n'est pas particulièrement utile comme variable de gabarit, mais pourrait être utile dans des tests conditionnels tels que :

```
{% if field.is_hidden %}
    {# Do something special #}
{% endif %}
```

```
{{ field.field }}
```

L'instance [Field](#) de la classe de formulaire que cet objet [BoundField](#) adapte. Vous pouvez l'utiliser pour accéder aux attributs de [Field](#), par exemple `{{ char_field.field.max_length }}`.

Voir aussi

Pour une liste complète des attributs et méthodes, voir [BoundField](#).

Boucles sur les champs masqués et visibles

Si vous affichez manuellement un formulaire dans un gabarit, sans faire appel à l'affichage par défaut des formulaires de Django, il peut être nécessaire de traiter différemment les champs `<input type="hidden">` des autres champs non masqués. Par exemple, comme les champs masqués ne produisent pas de contenu visible, l'affichage de messages d'erreur « à côté » du champ pourrait générer de la confusion pour les utilisateurs ; il faut donc gérer différemment les erreurs de ce type de champ.

Django fournit deux méthodes qui permettent de tourner en boucle indépendamment sur les champs visibles et masqués : `hidden_fields()` et `visible_fields()`. Voici une modification d'un exemple précédent en utilisant ces deux méthodes :

```
{# Include the hidden fields #}
{% for hidden in form.hidden_fields %}
{{ hidden }}
{% endfor %}
{# Include the visible fields #}
{% for field in form.visible_fields %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

Cet exemple ne gère pas du tout les erreurs des champs masqués. En principe, une erreur dans un champ masqué est un signe de manipulation abusive d'un formulaire, puisque l'interaction normale avec un formulaire ne les modifie pas. Cependant, vous pourriez facilement ajouter aussi une forme d'affichage pour ces erreurs de formulaire.

Gabarits de formulaire réutilisables

Si votre site utilise la même logique d'affichage des formulaires à plusieurs endroits, vous pouvez réduire la duplication en enregistrant la boucle de formulaire dans un gabarit autonome et en employant la balise [include](#) afin de réutiliser ce contenu dans d'autres gabarits :

```
# In your form template:
{% include "form_snippet.html" %}

# In form_snippet.html:
{% for field in form %}
    <div class="fieldWrapper">
```

```
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

Si l'objet formulaire transmis au gabarit possède un nom différent dans le contexte, vous pouvez lui donner un alias en utilisant le paramètre `with` de la balise [include](#):

```
{% include "form_snippet.html" with form=comment_form %}
```

Si vous constatez que vous reproduisez souvent ce même code, il vous faut peut-être envisager la création d'une [balise d'inclusion](#) personnalisée.

Sujets complémentaires

Cette page aborde les généralités, mais les formulaires peuvent faire bien d'autres choses encore :

- [Formulaires groupés](#)
 - [Données initiales pour les formulaires groupés](#)
 - [Restriction du nombre maximum de formulaires](#)
 - [Validation des formulaires groupés](#)
 - [Validation du nombre de formulaires dans les formulaires groupés](#)
 - [Tri et suppression de formulaires](#)
 - [Ajout de champs supplémentaires à des formulaires groupés](#)
 - [Transmission de paramètres personnalisés aux formulaires de jeux de formulaires](#)
 - [Utilisation des formulaires groupés dans les vues et les gabarits](#)
- [Création de formulaires à partir de modèles](#)
 - [ModelForm](#)
 - [Formulaires groupés de modèles](#)
 - [Sous-formulaires groupés](#)
- [Fichiers annexes de formulaire \(classe Media\)](#)
 - [Fichiers annexes définis statiquement](#)
 - [Media comme propriété dynamique](#)
 - [Chemins dans les définitions de fichiers annexes](#)
 - [Objets Media](#)
 - [Media pour les formulaires](#)

Voir aussi

[La référence des formulaires](#)

Contient la référence d'API complète, y compris les champs de formulaire, les composants de formulaire et la validation des champs et des formulaires.

Formulaires groupés

`class BaseFormSet` [\[source\]](#)

Les formulaires groupés sont une couche d'abstraction pour travailler avec plusieurs formulaires sur une même page. On peut comparer cela à un tableau de données. Admettons que l'on dispose du formulaire suivant :

```
>>> from django import forms
>>> class ArticleForm(forms.Form):
...     title = forms.CharField()
...     pub_date = forms.DateField()
```

Il peut être souhaitable de permettre à l'utilisateur de créer plusieurs articles à la fois. Pour créer des formulaires groupés à partir d'un formulaire `ArticleForm`, voici comment procéder :

```
>>> from django.forms import formset_factory
>>> ArticleFormSet = formset_factory(ArticleForm)
```

Vous avez maintenant créé un objet de formulaires groupés nommé `ArticleFormSet`. Cet objet vous donne la possibilité d'effectuer une itération sur les formulaires du groupe et de les afficher tout comme vous le feriez pour un formulaire habituel :

```
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
name="form-0-title" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input
type="text" name="form-0-pub_date" id="id_form-0-pub_date" /></td></tr>
```

Comme vous le voyez, un seul formulaire vierge a été affiché. Le nombre de formulaires vierges affichés dépend du paramètre `extra`. Par défaut, [formset_factory\(\)](#) définit un seul formulaire supplémentaire ; l'exemple suivant affichera deux formulaires vierges :

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
```

En itérant sur l'objet `formset`, les formulaires sont affichés dans l'ordre où ils ont été créés. Vous pouvez modifier cet ordre en écrivant une implémentation différente de la méthode `__iter__()`.

Il est aussi possible d'accéder aux formulaires groupés par un numéro d'index, ce qui renvoie le formulaire correspondant. Si vous surchargez `__iter__`, il est aussi nécessaire de surcharger `__getitem__` afin de garantir un comportement cohérent.

Données initiales pour les formulaires groupés

Les données initiales sont le moteur essentiel qui fait l'intérêt des formulaires groupés. Comme on peut le voir ci-dessus, vous pouvez définir le nombre de formulaires supplémentaires. Cela signifie que vous pouvez indiquer aux formulaires groupés combien de formulaires supplémentaires doivent être affichés en plus des formulaires générés à partir des données initiales. Examinons un exemple plus en détails

```
>>> import datetime
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
>>> formset = ArticleFormSet(initial=[
...     {'title': 'Django is now open source',
...     'pub_date': datetime.date.today(),}
... ])

>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
name="form-0-title" value="Django is now open source" id="id_form-0-title"
/></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input
type="text" name="form-0-pub_date" value="2008-05-12" id="id_form-0-pub_date"
/></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text"
name="form-1-title" id="id_form-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input
type="text" name="form-1-pub_date" id="id_form-1-pub_date" /></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text"
name="form-2-title" id="id_form-2-title" /></td></tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input
type="text" name="form-2-pub_date" id="id_form-2-pub_date" /></td></tr>
```

Nous avons maintenant trois formulaires qui s'affichent. Un pour les données initiales que nous lui avons transmises et deux formulaires supplémentaires. Notez également que nous transmettons une liste de dictionnaires comme données initiales.

Voir aussi

[Création de formulaires groupés à partir de modèles avec les formulaires groupés de modèle.](#)

Restriction du nombre maximum de formulaires

Le paramètre `max_num` de [formset_factory\(\)](#) donne la possibilité de restreindre le nombre de formulaires que les formulaires groupés vont afficher

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
```

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2, max_num=1)
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
name="form-0-title" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input
type="text" name="form-0-pub_date" id="id_form-0-pub_date" /></td></tr>
```

Si la valeur de `max_num` est plus grande que le nombre d'objets existants dans les données initiales, un maximum de `extra` formulaires vierges supplémentaires seront ajoutés au formulaire groupé, tant que le nombre total de formulaires n'excède pas `max_num`. Par exemple, si `extra=2` et `max_num=2` et que le formulaire groupé est initialisé avec un élément `initial`, on verra s'afficher un formulaire pour l'élément initial et un formulaire vierge.

Si le nombre d'éléments dans les données initiales dépasse `max_num`, tous les formulaires correspondant aux données initiales seront affichés quelle que soit la valeur de `max_num` et aucun formulaire supplémentaire ne sera affiché. Par exemple, si `extra=3` et `max_num=1` et que le formulaire groupé est initialisé avec deux éléments, ce sont deux formulaires avec les données initiales qui seront affichés.

Une valeur `max_num` `NONE` (par défaut) place une limite élevée du nombre de formulaires affichés (1000). En pratique, cela équivaut à aucune limite.

Par défaut, `max_num` ne s'applique qu'au nombre de formulaires affichés et non pas à la validation. Si `validate_max=True` est transmis à [formset_factory\(\)](#), `max_num` s'applique alors à la validation. Voir [Validation du nombre de formulaires dans les formulaires groupés](#).

Validation des formulaires groupés

La validation des formulaires groupés est presque identique à celle d'un formulaire `Form` normal.

L'objet de formulaires groupés contient une méthode `is_valid` afin de fournir une manière agile de valider tous les formulaires du groupe :

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm)
>>> data = {
...     'form-TOTAL_FORMS': '1',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
True
```

Nous n'avons fourni aucune donnée aux formulaires groupés ce qui aboutit à un formulaire valide. Les formulaires groupés sont assez intelligents pour ne pas prendre en compte des formulaires supplémentaires qui n'ont pas été modifiés. Si nous passons un article non valable :


```
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test',
...     'form-1-pub_date': '', # <-- this date is missing but required
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {'pub_date': ['This field is required.']}]
```

Comme on peut le voir, `formset.errors` est une liste composée d'éléments correspondants à chaque formulaire du groupe. La validation s'est faite pour chacun des deux formulaires, et le message d'erreur attendu apparaît pour le second.

`BaseFormSet.total_error_count()`[\[source\]](#)

Pour contrôler le nombre d'erreurs dans les formulaires groupés, nous pouvons utiliser la méthode `total_error_count`:

```
>>> # Using the previous example
>>> formset.errors
[{}, {'pub_date': ['This field is required.']}]
>>> len(formset.errors)
2
>>> formset.total_error_count()
1
```

Nous pouvons aussi vérifier si les données des formulaires diffèrent des données initiales (c'est-à-dire si le formulaire a été envoyé sans aucune donnée) :

```
>>> data = {
...     'form-TOTAL_FORMS': '1',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': '',
...     'form-0-pub_date': '',
... }
>>> formset = ArticleFormSet(data)
>>> formset.has_changed()
False
```

Rôle du formulaire de gestion (ManagementForm)

Vous avez peut-être remarqué les données supplémentaires (`form-TOTAL_FORMS`, `form-INITIAL_FORMS` et `form-MAX_NUM_FORMS`) qui étaient requises dans les données de formulaires groupés ci-dessus. Ces données sont exigées par le formulaire de gestion. Celui-ci est utilisé par les formulaires groupés pour gérer l'ensemble des formulaires contenus dans le groupe. Si vous ne fournissez pas ces données de gestion, une exception sera générée :

```
>>> data = {
...     'form-0-title': 'Test',
...     'form-0-pub_date': '',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
Traceback (most recent call last):
...
django.forms.utils.ValidationError: ['ManagementForm data is missing or has been
tampered with']
```

Ces données servent à conserver la trace du nombre d’instances de formulaires qui sont affichées. Si vous ajoutez de nouveaux formulaires par JavaScript, il est aussi nécessaire d’incrémenter les champs de comptage de ce formulaire. D’un autre côté, si vous utilisez JavaScript pour permettre la suppression d’objets existants, vous devez alors vous assurer que ceux qui sont supprimés soient correctement marqués comme « à supprimer » en incluant le paramètre `form-#-DELETE` dans les données d’envoi POST. Il est prévu que tous les formulaires soient présents dans les données POST quoi qu’il arrive.

Le formulaire de gestion est disponible sous forme d’attribut de l’objet de formulaires groupés. Lorsque vous affichez des formulaires groupés dans un gabarit, vous pouvez inclure toutes les données de gestion en affichant `{{ my_formset.management_form }}` (en remplaçant le nom de vos formulaires groupés en conséquence).

total_form_count et initial_form_count

`BaseFormSet` possède quelques méthodes étroitement liées aux données `total_form_count` et `initial_form_count` du formulaire de gestion `ManagementForm`.

`total_form_count` renvoie le nombre total de formulaires du groupe de formulaires.

`initial_form_count` renvoie le nombre de formulaires du groupe qui ont été pré-remplis et est également utilisé pour déterminer le nombre de formulaires requis. Vous n’aurez probablement jamais à surcharger l’une de ces méthodes, mais prenez garde à bien comprendre ce qu’elles font si vous deviez le faire.

empty_form

`BaseFormSet` fournit un attribut supplémentaire `empty_form` qui renvoie une instance de formulaire avec le préfixe `__prefix__` pour faciliter l’utilisation de formulaires dynamiques avec JavaScript.

Validation personnalisée des formulaires groupés

Les formulaires groupés ont une méthode `clean` similaire à celle d’une classe `Form`. C’est là que vous pouvez définir votre propre validation agissant au niveau des formulaires groupés :

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory
```

```

>>> from myapp.forms import ArticleForm

>>> class BaseArticleFormSet(BaseFormSet):
...     def clean(self):
...         """Checks that no two articles have the same title."""
...         if any(self.errors):
...             # Don't bother validating the formset unless each form is valid on
its own
...             return
...         titles = []
...         for form in self.forms:
...             title = form.cleaned_data['title']
...             if title in titles:
...                 raise forms.ValidationError("Articles in a set must have
distinct titles.")
...             titles.append(title)

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseArticleFormSet)
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test',
...     'form-1-pub_date': '1912-06-23',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
['Articles in a set must have distinct titles.']

```

La méthode `clean` des formulaires groupés est appelée après les méthodes `Form.clean` des formulaires individuels. Les erreurs produites à ce niveau sont accessibles par la méthode `non_form_errors()` sur l'objet des formulaires groupés.

Validation du nombre de formulaires dans les formulaires groupés

Django fournit plusieurs manières de valider le nombre minimum ou maximum de formulaires soumis. Les applications ayant besoin d'une validation plus adaptée du nombre de formulaires peuvent utiliser la validation personnalisée de formulaires groupés.

validate_max

Si `validate_max=True` est transmis à [`formset_factory\(\)`](#), la validation vérifiera également que le nombre de formulaires dans les données reçues moins ceux marqués pour la suppression ne dépasse pas `max_num`.

```

>>> from django.forms import formset_factory

```

```

>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, max_num=1, validate_max=True)
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MIN_NUM_FORMS': '',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test 2',
...     'form-1-pub_date': '1912-06-23',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
['Please submit 1 or fewer forms.']

```

`validate_max=True` valide strictement la valeur de `max_num`, même si cette valeur a été dépassée en raison d'un nombre de données initiales trop important.

Note

Quelle que soit la valeur de `validate_max`, si le nombre de formulaires dans un jeu de données dépasse `max_num` de plus de 1000, la validation du formulaire échoue comme si `validate_max` avait été défini ; de plus, seuls les 1000 premiers formulaires dépassant `max_num` seront validés. Tout le reste est purement ignoré. Il s'agit d'une mesure de protection contre les attaques de remplissage de mémoire employant des requêtes POST contrefaites.

validate_min

Si `validate_min=True` est passé à [`formset_factory\(\)`](#), la validation vérifiera également que le nombre de formulaires dans les données reçues moins ceux marqués pour la suppression est supérieur ou égal à `min_num`.

```

>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, min_num=3, validate_min=True)
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MIN_NUM_FORMS': '',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test 2',
...     'form-1-pub_date': '1912-06-23',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors

```

```
[{}, {}]
>>> formset.non_form_errors()
['Please submit 3 or more forms.']
```

Tri et suppression de formulaires

La méthode `formset_factory()` fournit deux paramètres facultatifs `can_order` et `can_delete` pour faciliter le tri et la suppression de formulaires dans des formulaires groupés.

can_order

`BaseFormSet.can_order`

Valeur par défaut : `False`

Ce paramètre permet de créer des formulaires groupés qui peuvent être triés :

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, can_order=True)
>>> formset = ArticleFormSet(initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
name="form-0-title" value="Article #1" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input
type="text" name="form-0-pub_date" value="2008-05-10" id="id_form-0-pub_date"
/></td></tr>
<tr><th><label for="id_form-0-ORDER">Order:</label></th><td><input type="number"
name="form-0-ORDER" value="1" id="id_form-0-ORDER" /></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text"
name="form-1-title" value="Article #2" id="id_form-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input
type="text" name="form-1-pub_date" value="2008-05-11" id="id_form-1-pub_date"
/></td></tr>
<tr><th><label for="id_form-1-ORDER">Order:</label></th><td><input type="number"
name="form-1-ORDER" value="2" id="id_form-1-ORDER" /></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text"
name="form-2-title" id="id_form-2-title" /></td></tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input
type="text" name="form-2-pub_date" id="id_form-2-pub_date" /></td></tr>
<tr><th><label for="id_form-2-ORDER">Order:</label></th><td><input type="number"
name="form-2-ORDER" id="id_form-2-ORDER" /></td></tr>
```

Ceci ajoute un champ supplémentaire à chaque formulaire. Ce nouveau champ s'appelle `ORDER` et est de type `forms.IntegerField`. Pour les formulaires générés à partir des données initiales, ces champs reçoivent automatiquement une valeur numérique. Voyons ce qui se passe si l'utilisateur modifie ces valeurs :

```
>>> data = {
...     'form-TOTAL_FORMS': '3',
```

```

...     'form-INITIAL_FORMS': '2',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Article #1',
...     'form-0-pub_date': '2008-05-10',
...     'form-0-ORDER': '2',
...     'form-1-title': 'Article #2',
...     'form-1-pub_date': '2008-05-11',
...     'form-1-ORDER': '1',
...     'form-2-title': 'Article #3',
...     'form-2-pub_date': '2008-05-01',
...     'form-2-ORDER': '0',
... }

>>> formset = ArticleFormSet(data, initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> formset.is_valid()
True
>>> for form in formset.ordered_forms:
...     print(form.cleaned_data)
{'pub_date': datetime.date(2008, 5, 1), 'ORDER': 0, 'title': 'Article #3'}
{'pub_date': datetime.date(2008, 5, 11), 'ORDER': 1, 'title': 'Article #2'}
{'pub_date': datetime.date(2008, 5, 10), 'ORDER': 2, 'title': 'Article #1'}

```

can_delete

BaseFormSet.can_delete

Valeur par défaut : False

Ce paramètre permet de créer des formulaires groupés où l'on peut choisir des formulaires à supprimer :

```

>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, can_delete=True)
>>> formset = ArticleFormSet(initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
name="form-0-title" value="Article #1" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input
type="text" name="form-0-pub_date" value="2008-05-10" id="id_form-0-pub_date"
/></td></tr>
<tr><th><label for="id_form-0-DELETE">Delete:</label></th><td><input
type="checkbox" name="form-0-DELETE" id="id_form-0-DELETE" /></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text"
name="form-1-title" value="Article #2" id="id_form-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input
type="text" name="form-1-pub_date" value="2008-05-11" id="id_form-1-pub_date"
/></td></tr>
<tr><th><label for="id_form-1-DELETE">Delete:</label></th><td><input
type="checkbox" name="form-1-DELETE" id="id_form-1-DELETE" /></td></tr>

```

```

<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text"
name="form-2-title" id="id_form-2-title" /></td></tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input
type="text" name="form-2-pub_date" id="id_form-2-pub_date" /></td></tr>
<tr><th><label for="id_form-2-DELETE">Delete:</label></th><td><input
type="checkbox" name="form-2-DELETE" id="id_form-2-DELETE" /></td></tr>

```

Un peu comme pour `can_order`, ce paramètre ajoute un nouveau champ `DELETE` de type `forms.BooleanField` à chaque formulaire. Lorsque les données reviennent et que certains de ces champs ont une valeur « vrai », il est possible d'accéder aux formulaires concernés avec `deleted_forms`:

```

>>> data = {
...     'form-TOTAL_FORMS': '3',
...     'form-INITIAL_FORMS': '2',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Article #1',
...     'form-0-pub_date': '2008-05-10',
...     'form-0-DELETE': 'on',
...     'form-1-title': 'Article #2',
...     'form-1-pub_date': '2008-05-11',
...     'form-1-DELETE': '',
...     'form-2-title': '',
...     'form-2-pub_date': '',
...     'form-2-DELETE': '',
... }

>>> formset = ArticleFormSet(data, initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> [form.cleaned_data for form in formset.deleted_forms]
[{'DELETE': True, 'pub_date': datetime.date(2008, 5, 10), 'title': 'Article #1'}]

```

Si vous utilisez un [ModelFormSet](#), les instances de modèle correspondant aux formulaires supprimés seront supprimées lorsque `formset.save()` sera appelée.

Si vous appelez `formset.save(commit=False)`, les objets ne seront pas supprimés automatiquement. Vous devrez appeler `delete()` pour chacun des [formset.deleted_objects](#) afin que la suppression soit effective :

```

>>> instances = formset.save(commit=False)
>>> for obj in formset.deleted_objects:
...     obj.delete()

```

D'autre part, si vous utilisez un `FormSet` simple, c'est à vous de gérer `formset.deleted_forms`, par exemple dans la méthode `save()` de votre formulaire groupé, car il n'existe pas de notion générale sur ce qu'implique la suppression d'un formulaire.

Ajout de champs supplémentaires à des formulaires groupés

L'ajout de champs supplémentaires aux formulaires groupés est simple à effectuer. La classe de base des formulaires groupés offre une méthode `add_fields`. Vous pouvez simplement surcharger cette méthode pour ajouter vos propres champs ou même pour redéfinir les champs/attributs par défaut des champs de tri et de suppression :

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> class BaseArticleFormSet(BaseFormSet):
...     def add_fields(self, form, index):
...         super(BaseArticleFormSet, self).add_fields(form, index)
...         form.fields["my_field"] = forms.CharField()

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseArticleFormSet)
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
name="form-0-title" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input
type="text" name="form-0-pub_date" id="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-0-my_field">My field:</label></th><td><input
type="text" name="form-0-my_field" id="id_form-0-my_field" /></td></tr>
```

Transmission de paramètres personnalisés aux formulaires de jeux de formulaires

Il arrive parfois qu'une classe de formulaire accepte des paramètres personnalisés, comme `MyArticleForm`. Vous pouvez transmettre ce paramètre lors de l'instanciation du jeu de formulaires :

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm

>>> class MyArticleForm(ArticleForm):
...     def __init__(self, *args, **kwargs):
...         self.user = kwargs.pop('user')
...         super(MyArticleForm, self).__init__(*args, **kwargs)

>>> ArticleFormSet = formset_factory(MyArticleForm)
>>> formset = ArticleFormSet(form_kwargs={'user': request.user})
```

Le paramètre `form_kwargs` peut aussi dépendre de l'instance de formulaire spécifique. La classe de base du jeu de formulaires fournit une méthode `get_form_kwargs`. Celle-ci accepte un seul paramètre, l'indice du formulaire dans le jeu de formulaire. Cet indice vaut `NONE` pour le formulaire vide [empty form](#):

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory
```



```
>>> class BaseArticleFormSet(BaseFormSet):
...     def get_form_kwargs(self, index):
...         kwargs = super(BaseArticleFormSet, self).get_form_kwargs(index)
...         kwargs['custom_kwarg'] = index
...         return kwargs
```

New in Django 1.9:

Le paramètre `form_kwargs` a été ajouté.

Utilisation des formulaires groupés dans les vues et les gabarits

L'utilisation de formulaires groupés dans une vue est aussi simple que l'utilisation d'une classe `Form` habituelle. La seule chose à laquelle il faut être attentif est de ne pas oublier d'inclure le formulaire de gestion dans le gabarit. Examinons un exemple de vue :

```
from django.forms import formset_factory
from django.shortcuts import render
from myapp.forms import ArticleForm

def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    if request.method == 'POST':
        formset = ArticleFormSet(request.POST, request.FILES)
        if formset.is_valid():
            # do something with the formset.cleaned_data
            pass
    else:
        formset = ArticleFormSet()
    return render(request, 'manage_articles.html', {'formset': formset})
```

Le gabarit `manage_articles.html` pourrait ressembler à ceci :

```
<form method="post" action="">
    {{ formset.management_form }}
    <table>
        {% for form in formset %}
            {{ form }}
        {% endfor %}
    </table>
</form>
```

Toutefois, le code ci-dessus peut être légèrement raccourci en laissant les formulaires groupés se charger eux-mêmes du formulaire de gestion :

```
<form method="post" action="">
    <table>
        {{ formset }}
    </table>
</form>
```

Le résultat du code ci-dessus est que la classe de formulaires groupés va appeler sa méthode `as_table`.

Affichage manuel de `can_delete` et `can_order`

Si vous affichez manuellement les champs dans le gabarit, vous pouvez afficher le paramètre `can_delete` avec `{{ form.DELETE }}`:

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        <ul>
            <li>{{ form.title }}</li>
            <li>{{ form.pub_date }}</li>
            {% if formset.can_delete %}
                <li>{{ form.DELETE }}</li>
            {% endif %}
        </ul>
    {% endfor %}
</form>
```

De la même façon, si les formulaires groupés peuvent être triés (`can_order=True`), il est possible d'afficher le champ de tri avec `{{ form.ORDER }}`.

Utilisation de plus d'un objet de formulaires groupés dans une vue

Il est possible d'utiliser plus d'un objet de formulaires groupés dans une vue. Les formulaires groupés ont un comportement très semblable à celui des formulaires. Ceci dit, vous pouvez utiliser l'attribut `prefix` afin de préfixer les noms de champs des formulaires groupés avec une valeur donnée, ce qui permettra d'envoyer plusieurs ensembles de formulaires groupés à une vue sans conflit de nommage. Voyons un peu comment cela pourrait être fait :

```
from django.forms import formset_factory
from django.shortcuts import render
from myapp.forms import ArticleForm, BookForm

def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    BookFormSet = formset_factory(BookForm)
    if request.method == 'POST':
        article_formset = ArticleFormSet(request.POST, request.FILES,
prefix='articles')
        book_formset = BookFormSet(request.POST, request.FILES, prefix='books')
        if article_formset.is_valid() and book_formset.is_valid():
            # do something with the cleaned_data on the formsets.
            pass
    else:
        article_formset = ArticleFormSet(prefix='articles')
        book_formset = BookFormSet(prefix='books')
    return render(request, 'manage_articles.html', {
        'article_formset': article_formset,
        'book_formset': book_formset,
    })
```

Les formulaires groupés seraient ensuite affichés comme d'habitude. Il est important de relever que vous devez indiquer `prefix` dans tous les cas (POST ou non), afin que l'affichage et le traitement des données puissent se faire correctement.

Création de formulaires à partir de modèles

ModelForm

`class ModelForm`[\[source\]](#)

Si vous construisez une application basée sur une base de données, il y a des chances pour que vos formulaires correspondent étroitement avec les modèles Django. Par exemple, vous pourriez avoir un modèle `BlogComment` et vouloir créer un formulaire permettant d'envoyer des commentaires. Dans ce cas, il serait redondant de devoir définir les types de champs du formulaire, car vous avez déjà défini des champs au niveau du modèle.

C'est pour cette raison que Django fournit une classe utilitaire permettant de créer une classe de formulaire `Form` à partir d'un modèle Django.

Par exemple :

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'reporter']

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

Types de champs

La classe de formulaire générée contiendra un champ de formulaire pour chaque champ de modèle inclus, dans l'ordre indiqué par l'attribut `fields`.

Chaque champ de modèle possède un champ de formulaire par défaut. Par exemple, le champ `CharField` d'un modèle est représenté par un champ de formulaire `CharField`. Un champ `ManyToManyField` d'un modèle est représenté par un champ de formulaire `MultipleChoiceField`. Voici la liste complète des correspondances :

Champ de modèle

Champ de formulaire

[AutoField](#)

Non représenté dans le formulaire

[BigIntegerField](#)

[IntegerField](#) avec
min_value à
-9223372036854775808 et
max_value à
9223372036854775807.

[BooleanField](#)

[CharField](#)

[BooleanField](#)
[CharField](#) avec max_length
définie à la même valeur que
max_length du champ de modèle

[CommaSeparatedIntegerField](#)

[CharField](#)

[DateField](#)

[DateField](#)

[DateTimeField](#)

[DateTimeField](#)

[DecimalField](#)

[DecimalField](#)

[EmailField](#)

[EmailField](#)

[FileField](#)

[FileField](#)

[FilePathField](#)

[FilePathField](#)

[FloatField](#)

[FloatField](#)

[ForeignKey](#)

[ModelChoiceField](#) (voir ci-
dessous)

[ImageField](#)

[ImageField](#)

[IntegerField](#)

[IntegerField](#)

[IPAddressField](#)

[IPAddressField](#)

[GenericIPAddressField](#)

[GenericIPAddressField](#)

[ManyToManyField](#)

[ModelMultipleChoiceField](#)
(voir ci-dessous)

[NullBooleanField](#)

[NullBooleanField](#)

[PositiveIntegerField](#)

[IntegerField](#)

[PositiveSmallIntegerField](#)

[IntegerField](#)

[SlugField](#)

[SlugField](#)

[SmallIntegerField](#)

[IntegerField](#)

[TextField](#)

[CharField](#) avec
widget=forms.Textarea

[TimeField](#)

[TimeField](#)

[URLField](#)

[URLField](#)

Comme l'on peut s'y attendre, les champs de modèle de type `ForeignKey` et `ManyToManyField` sont des cas spéciaux :

- `ForeignKey` est représenté par `django.forms.ModelChoiceField`, qui est un champ `ChoiceField` dont les choix possibles sont définis par le `QuerySet` d'un modèle.
- `ManyToManyField` est représenté par un `django.forms.ModelMultipleChoiceField`, qui est un champ `MultipleChoiceField` dont les choix possibles sont définis par le `QuerySet` d'un modèle.

De plus, chaque champ de formulaire généré possède les attributs comme suit :

- Si le champ de modèle a `blank=True`, `required` est alors défini à `False` dans le champ de formulaire. Sinon, `required=True`.
- L'étiquette (`label`) du champ de formulaire est défini à la valeur `verbose_name` du champ de modèle, avec le premier caractère en majuscules.
- La valeur `help_text` du champ de formulaire est définie à la valeur `help_text` du champ de modèle.
- Si la valeur `choices` d'un champ de modèle est définie, le composant (`widget`) du champ de formulaire sera de type `Select` dont les choix proviendront de la valeur `choices` du champ de modèle. Ces choix incluent normalement le choix vierge qui est sélectionné par défaut. Si le champ est obligatoire, cela force l'utilisateur à faire un choix. Le choix vierge n'est pas inclus si le champ de modèle a `blank=False` ainsi qu'une valeur par défaut (`default`) explicite (c'est la valeur `default` qui sera initialement sélectionnée).

Pour terminer, notez que vous pouvez surcharger le champ de formulaire utilisé pour un champ de modèle donné. Consultez [Surcharge des champs par défaut](#) ci-dessous.

Un exemple complet

Considérez cet ensemble de modèles :

```
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = (
    ('MR', 'Mr.'),
    ('MRS', 'Mrs.'),
    ('MS', 'Ms.'),
)

class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)
```

```

def __str__(self):
    return self.name

class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ['name', 'title', 'birth_date']

class BookForm(ModelForm):
    class Meta:
        model = Book
        fields = ['name', 'authors']

```

Avec ces modèles, les sous-classes `ModelForm` ci-dessus seraient à peu près équivalentes à ceci (avec comme seule différence la méthode `save()` que nous aborderons dans un moment) :

```

from django import forms

class AuthorForm(forms.Form):
    name = forms.CharField(max_length=100)
    title = forms.CharField(
        max_length=3,
        widget=forms.Select(choices=TITLE_CHOICES),
    )
    birth_date = forms.DateField(required=False)

class BookForm(forms.Form):
    name = forms.CharField(max_length=100)
    authors = forms.ModelMultipleChoiceField(queryset=Author.objects.all())

```

Validation d'un ModelForm

La validation d'un `ModelForm` se distingue en deux étapes importantes :

1. La [validation du formulaire](#)
2. La [validation de l'instance de modèle](#)

Comme pour la validation de formulaire normale, la validation des formulaires de modèle est déclenchée implicitement lors de l'appel à `is_valid()` ou par l'accèsion à l'attribut `errors`, ou explicitement en appelant `full_clean()`, même si en pratique cette dernière méthode est rarement utilisée.

La validation de `Model` (`Model.full_clean()`) est déclenchée à l'intérieur de l'étape de validation de formulaire, juste après l'appel à la méthode `clean()` du formulaire.

Avertissement

Le processus de nettoyage modifie l'instance de modèle transmise au constructeur de `ModelForm` de plusieurs manières. Par exemple, toute valeur de champ date du modèle est convertie en vrai objet date.

Une validation qui échoue peut laisser l'instance de modèle sous-jacente dans un état non cohérent et il n'est donc pas conseillé de la réutiliser.

Surcharge de la méthode `clean()`

Vous pouvez surcharger la méthode `clean()` d'un formulaire de modèle pour procéder à des validations supplémentaires de la même manière que pour un formulaire normal.

Une instance de formulaire de modèle liée à un objet modèle contient un attribut `instance` qui donne à ses méthodes l'accès à cette instance de modèle spécifique.

Avertissement

La méthode `ModelForm.clean()` définit un drapeau qui fait que l'étape de [validation de modèle](#) valide l'unicité des champs de modèle marqués comme `unique`, `unique_together` ou `unique_for_date|month|year`.

Si vous souhaitez surcharger la méthode `clean()` et maintenir cette validation, vous devez appeler la méthode `clean()` de la classe parente.

Interactions avec la validation des modèles

Dans le cadre du processus de validation, `ModelForm` appelle la méthode `clean()` de chaque champ du modèle possédant un champ de formulaire correspondant. Si vous avez exclu des champs de modèle, la validation ne sera pas effectuée pour ces champs. Consultez la documentation de la [validation des formulaires](#) pour en savoir plus sur le fonctionnement du nettoyage et de la validation des champs.

La méthode `clean()` du modèle est appelée avant les contrôles d'unicité. Consultez la [validation des objets](#) pour plus d'informations sur la méthode `clean()` des modèles.

Considérations sur les messages d'erreur des modèles

Les messages d'erreur définis au niveau des [champs de formulaire](#) ou au niveau de la [classe Meta de formulaire](#) ont toujours la priorité sur les messages d'erreur définis au niveau des [champs de modèle](#).

Les messages d'erreur définis au niveau des [champs de modèle](#) ne sont utilisés que lorsqu'une erreur `ValidationError` est générée à l'étape de [validation du modèle](#) et qu'aucun message d'erreur correspondant n'est défini au niveau du formulaire.

Vous pouvez surcharger les messages d'erreur issus de `NON_FIELD_ERRORS` et générés par la validation des modèles en ajoutant la clé [NON_FIELD_ERRORS](#) au dictionnaire `error_messages` de la classe `Meta` interne des classes `ModelForm`:

```
from django.forms import ModelForm
from django.core.exceptions import NON_FIELD_ERRORS
```

```

class ArticleForm(ModelForm):
    class Meta:
        error_messages = {
            NON_FIELD_ERRORS: {
                'unique_together': "%(model_name)s's %(field_labels)s are not
unique.",
            }
        }

```

La méthode `save()`

Chaque `ModelForm` possède aussi une méthode `save()`. Celle-ci crée et enregistre un objet en base de données à partir des données saisies dans le formulaire. Une sous-classe de `ModelForm` peut accepter une instance existante de modèle par le paramètre nommé `instance`. Si celui-ci est présent, `save()` met à jour cette instance. Dans le cas contraire, `save()` crée une nouvelle instance du modèle concerné :

```

>>> from myapp.models import Article
>>> from myapp.forms import ArticleForm

# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()

```

Notez que si le formulaire [n'a pas été validé](#), l'appel à `save()` s'en occupera en contrôlant `form.errors`. Une erreur `ValueError` est générée si les données du formulaire ne sont pas valides, par exemple si `form.errors` est évalué à `True`.

Cette méthode `save()` accepte un paramètre nommé facultatif `commit`, qui accepte les valeurs `True` ou `False`. Si vous appelez `save()` avec `commit=False`, elle renvoie un objet qui n'aura pas encore été enregistré dans la base de données. Dans ce cas, c'est à vous d'appeler `save()` sur l'instance de modèle obtenue. C'est utile lorsque vous souhaitez effectuer un traitement particulier sur l'objet avant qu'il soit enregistré, ou si vous voulez utiliser l'une des [options d'enregistrement de modèle](#) spécialisées. `commit` vaut `True` par défaut.

Un autre effet de bord de l'utilisation de `commit=False` se voit lorsque le modèle possède une relation plusieurs-à-plusieurs vers un autre modèle. Dans ce cas, Django ne peut pas enregistrer immédiatement les données du formulaire de la relation plusieurs-à-plusieurs. La raison est qu'il n'est pas possible d'enregistrer des données plusieurs-à-plusieurs propres à une instance si celle-ci n'existe pas encore dans la base de données.

Pour contourner ce problème, chaque fois que vous enregistrez un formulaire avec `commit=False`, Django ajoute une méthode `save_m2m()` à la sous-classe de votre `ModelForm`. Après avoir manuellement enregistré l'instance dérivée du formulaire, vous pouvez appeler `save_m2m()` pour enregistrer les données de formulaire de type plusieurs-à-plusieurs. Par exemple :

```
# Create a form instance with POST data.
>>> f = AuthorForm(request.POST)

# Create, but don't save the new author instance.
>>> new_author = f.save(commit=False)

# Modify the author in some way.
>>> new_author.some_field = 'some_value'

# Save the new instance.
>>> new_author.save()

# Now, save the many-to-many data for the form.
>>> f.save_m2m()
```

Il n'est utile d'appeler `save_m2m()` que si vous utilisez `save(commit=False)`. Lorsque vous appelez simplement la méthode `save()` d'un formulaire, toutes les données, y compris les données plusieurs-à-plusieurs, sont enregistrées sans avoir besoin de faire quoi que ce soit d'autre. Par exemple :

```
# Create a form instance with POST data.
>>> a = Author()
>>> f = AuthorForm(request.POST, instance=a)

# Create and save the new author instance. There's no need to do anything else.
>>> new_author = f.save()
```

À part les méthodes `save()` et `save_m2m()`, un `ModelForm` fonctionne exactement de la même manière que tout autre formulaire `forms`. Par exemple, la méthode `is_valid()` est utilisée pour vérifier la validité, la méthode `is_multipart()` est utilisée pour déterminer si un formulaire nécessite un envoi de fichier `multipart` (et donc que `request.FILES` doit être transmis au formulaire), etc. Voir [Liaison de fichiers téléversés avec un formulaire](#) pour plus d'informations.

Sélection des champs à utiliser

Il est fortement recommandé de définir explicitement tous les champs qui doivent être présents dans le formulaire en utilisant l'attribut `fields`. Si vous ne le faites pas, cela peut facilement devenir source de problèmes de sécurité lorsqu'un formulaire permet à un utilisateur de définir certains champs de manière non prévue, particulièrement lorsqu'on ajoute de nouveaux champs à un modèle. En fonction de la façon dont le formulaire est affiché, il est même possible que le problème ne soit pas visible sur la page Web.

Une autre approche est d'inclure automatiquement tous les champs ou d'en exclure certains. Mais à la base, cette approche est notoirement moins sécurisée et a déjà abouti à des failles sévères sur des sites Web célèbres (par ex. [GitHub](#)).

Il existe toutefois deux raccourcis disponibles pour les cas où vous pouvez garantir que ces questions de sécurité ne s'appliquent pas à votre situation :

1. Définir l'attribut `fields` à la valeur spéciale `'__all__'` pour indiquer que tous les champs du modèle doivent être utilisés. Par exemple :

```
from django.forms import ModelForm

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = '__all__'
```

2. Définir l'attribut `exclude` de la classe interne `Meta` du `ModelForm` à une liste de champs à exclure du formulaire.

Par exemple :

```
class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        exclude = ['title']
```

Comme le modèle `Author` possède les trois champs `name`, `title` et `birth_date`, nous nous retrouvons avec les champs `name` et `birth_date` affichés dans le formulaire.

Si l'une de ces techniques est employée, l'ordre d'apparition des champs dans le formulaire respecte l'ordre de définition des champs du modèle, avec les instances `ManyToManyField` apparaissant en dernier.

De plus, Django applique la règle suivante : si vous définissez `editable=False` au niveau du champ de modèle, *tout* formulaire créé à partir du modèle via `ModelForm` ne contiendra pas ce champ.

Changed in Django 1.8:

Dans les versions précédentes, l'omission de `fields` et `exclude` aboutissait à un formulaire contenant tous les champs du modèle. Dans la version actuelle, ceci amène à une exception [`ImproperlyConfigured`](#).

Note

Tout champ non inclus dans un formulaire par la logique ci-dessus ne sera pas touché par la méthode `save()` du formulaire. De même, si vous ajoutez manuellement dans un formulaire un champ initialement exclu, ce champ ne sera pas initialisé avec les données de l'instance de modèle.

Django empêche toute tentative d'enregistrer un modèle incomplet. Si le modèle n'autorise pas les champs manquants à être vides et que ces champs n'ont pas de valeur par défaut, tout tentative d'appeler `save()` sur un `ModelForm` avec des champs manquants échouera. Pour éviter ce problème, vous devez créer une instance de modèle comportant des valeurs initiales pour les champs obligatoires manquants :

```
author = Author(title='Mr')
form = PartialAuthorForm(request.POST, instance=author)
form.save()
```

Une variante est d'utiliser `save(commit=False)` et de définir manuellement les champs obligatoires supplémentaires :

```
form = PartialAuthorForm(request.POST)
author = form.save(commit=False)
author.title = 'Mr'
author.save()
```

Consultez la [section sur l'enregistrement des formulaires](#) pour plus de détails sur l'utilisation de `save(commit=False)`.

Surcharge des champs par défaut

Les types de champs par défaut, tels que décrits dans le tableau ci-dessus [Types de champs](#), sont des choix raisonnables. Si votre modèle contient un champ `DateTimeField`, il est probable que vous vouliez représenter ce champ dans un formulaire par un champ de formulaire `DateTimeField`. Mais `ModelForm` apporte la souplesse de pouvoir modifier le champ de formulaire pour un modèle défini.

Pour indiquer un composant personnalisé pour un champ, utilisez l'attribut `widgets` de la classe `Meta` interne. Ce doit être un dictionnaire faisant correspondre les noms de champs à des classes ou des instances de composants.

Par exemple, si vous voulez que le champ `CharField` de l'attribut `name` du modèle `Author` soit représenté par un composant `<textarea>` au lieu du composant `<input type="text">` par défaut, vous pouvez surcharger le composant du champ :

```
from django.forms import ModelForm, Textarea
from myapp.models import Author

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

Le dictionnaire `widgets` accepte aussi bien des instances de composants (par ex. `Textarea(...)`) que des classes (par ex. `Textarea`).

De la même façon, vous pouvez indiquer les attributs `labels`, `help_texts` et `error_messages` de la classe interne `Meta` si vous avez besoin de personnaliser davantage un champ.

Par exemple, si vous souhaitez personnaliser la formulation de tous les textes visibles pour l'utilisateur concernant le champ `name`:

```
from django.utils.translation import ugettext_lazy as _

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title', 'birth_date')
        labels = {
            'name': _('Writer'),
        }
        help_texts = {
            'name': _('Some useful help text.'),
        }
        error_messages = {
            'name': {
                'max_length': _("This writer's name is too long."),
            },
        }
```

Vous pouvez aussi indiquer `field_classes` pour personnaliser le type des champs créés par le formulaire.

Par exemple, si vous souhaitez utiliser `MySlugFormField` pour le champ `slug`, vous pourriez procéder ainsi :

```
from django.forms import ModelForm
from myapp.models import Article

class ArticleForm(ModelForm):
    class Meta:
        model = Article
        fields = ['pub_date', 'headline', 'content', 'reporter', 'slug']
        field_classes = {
            'slug': MySlugFormField,
        }
```

Finalement, si vous souhaitez contrôler complètement un champ, y compris son type, ses validateurs, son caractère obligatoire, etc., vous pouvez le faire en définissant les champs de manière déclarative comme on le ferait pour un formulaire `Form` normal.

Si vous souhaitez indiquer les validateurs d'un champ, vous pouvez le faire en définissant le champ de manière déclarative et en indiquant le paramètre `validators`:

```
from django.forms import ModelForm, CharField
from myapp.models import Article

class ArticleForm(ModelForm):
    slug = CharField(validators=[validate_slug])
```

```
class Meta:
    model = Article
    fields = ['pub_date', 'headline', 'content', 'reporter', 'slug']
```

New in Django 1.9:

L'attribut `Meta.field_classes` a été ajouté.

Note

Lorsque vous créez une instance de formulaire explicitement comme ceci, il est important de comprendre comment les formulaires de type `ModelForm` et `Form` sont liés.

`ModelForm` est comme un formulaire `Form` normal qui peut générer automatiquement certains champs. Les champs automatiquement générés dépendent du contenu de la classe `Meta` et des champs éventuellement définis de manière déclarative. Au fond, `ModelForm` va **uniquement** générer les champs qui sont **absents** du formulaire, ou en d'autres termes, les champs qui n'ont pas été définis déclarativement.

Les champs définis déclarativement sont laissés tels quels, ce qui fait que toute personnalisation effectuée au niveau des attributs de `Meta`, tels que `widgets`, `labels`, `help_texts` ou `error_messages` est ignorée ; ces derniers ne s'appliquent qu'aux champs automatiquement générés.

De même, les champs définis déclarativement ne dérivent pas leurs attributs comme `max_length` ou `required` du modèle correspondant. Si vous voulez conserver le comportement défini au niveau du modèle, vous devez définir les paramètres adéquats explicitement dans la déclaration du champ de formulaire.

Par exemple, si le modèle `Article` ressemble à ceci :

```
class Article(models.Model):
    headline = models.CharField(
        max_length=200,
        null=True,
        blank=True,
        help_text='Use puns liberally',
    )
    content = models.TextField()
```

et que vous vouliez effectuer une validation particulière pour `headline` tout en conservant les valeurs `blank` et `help_text` d'origine, voici comment vous pourriez définir `ArticleForm`:

```
class ArticleForm(ModelForm):
    headline = MyFormField(
        max_length=200,
        required=False,
        help_text='Use puns liberally',
    )

    class Meta:
```

```
model = Article
fields = ['headline', 'content']
```

Vous devez vous assurer que le type du champ de formulaire peut être utilisé pour définir le contenu du champ de modèle correspondant. S'ils ne sont pas compatibles, vous obtiendrez une erreur `ValueError` car aucune conversion implicite n'est effectuée.

Consultez la [documentation des champs de formulaire](#) pour plus d'informations sur les champs et leurs paramètres.

Activation de la régionalisation des champs

Par défaut, les champs d'un `ModelForm` ne régionalisent pas leurs données. Pour activer cette régionalisation, vous pouvez utiliser l'attribut `localized_fields` de la classe `Meta`.

```
>>> from django.forms import ModelForm
>>> from myapp.models import Author
>>> class AuthorForm(ModelForm):
...     class Meta:
...         model = Author
...         localized_fields = ('birth_date',)
```

Si `localized_fields` est défini à la valeur spéciale `'__all__'`, les valeurs de tous les champs seront régionalisées.

Héritage de formulaire

Comme pour les formulaires de base, il est possible d'étendre et de réutiliser les `ModelForms` par l'héritage. C'est utile lorsque vous devez déclarer des champs ou des méthodes supplémentaires dans une sous-classe pour les utiliser dans des formulaires dérivés de modèles. Par exemple, en utilisant la classe `ArticleForm` présentée précédemment :

```
>>> class EnhancedArticleForm(ArticleForm):
...     def clean_pub_date(self):
...         ...
```

Cela crée un formulaire qui se comporte comme `ArticleForm`, à l'exception de certaines validations et nettoyages supplémentaires pour le champ `pub_date`.

You can also subclass the parent's `Meta` inner class if you want to change the `Meta.fields` or `Meta.exclude` lists:

```
>>> class RestrictedArticleForm(EnhancedArticleForm):
...     class Meta(ArticleForm.Meta):
...         exclude = ('body',)
```

Nous conservons ici la méthode supplémentaire de `EnhancedArticleForm` et enlevons un champ de la classe `ArticleForm.Meta` originale.

Il faut cependant relever certains points.

Les champs à inclure peuvent être spécifiés en utilisant les paramètres nommés `fields` et `exclude`, ou les attributs correspondants de la classe interne `Meta` du `ModelForm`. Veuillez consulter la documentation [Sélection des champs à utiliser](#) des `ModelForm`.

... ou pour activer la régionalisation de certains champs :

```
>>> Form = modelform_factory(Author, form=AuthorForm,
localized_fields=("birth_date",))
```

Formulaires groupés de modèles

```
class models.BaseModelFormSet
```

Comme pour les [formulaires groupés normaux](#), Django fournit quelques classes de formulaires groupés améliorées qui facilitent le travail avec les modèles Django. Réutilisons le modèle `Author` défini plus haut :

```
>>> from django.forms import modelformset_factory
>>> from myapp.models import Author
>>> AuthorFormSet = modelformset_factory(Author, fields=('name', 'title'))
```

L'utilisation de `fields` indique aux formulaires groupés les champs qu'ils doivent utiliser. Il est aussi possible d'utiliser une approche par exclusion, indiquant quels sont les champs à exclure :

```
>>> AuthorFormSet = modelformset_factory(Author, exclude=('birth_date',))
```

Changed in Django 1.8:

Dans les versions précédentes, l'omission de `fields` et `exclude` aboutissait à un jeu de formulaires contenant tous les champs du modèle. Dans la version actuelle, ceci amène à une exception [ImproperlyConfigured](#).

Cela crée un jeu de formulaires groupés qui sont capables de gérer les données associées au modèle `Author`. Le fonctionnement est le même que pour des formulaires groupés normaux :

```
>>> formset = AuthorFormSet()
>>> print(formset)
<input type="hidden" name="form-TOTAL_FORMS" value="1" id="id_form-TOTAL_FORMS"
/><input type="hidden" name="form-INITIAL_FORMS" value="0" id="id_form-
INITIAL_FORMS" /><input type="hidden" name="form-MAX_NUM_FORMS" id="id_form-
MAX_NUM_FORMS" />
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-
name" type="text" name="form-0-name" maxlength="100" /></td></tr>
<tr><th><label for="id_form-0-title">Title:</label></th><td><select name="form-0-
title" id="id_form-0-title">
<option value="" selected="selected">-----</option>
<option value="MR">Mr.</option>
<option value="MRS">Mrs.</option>
<option value="MS">Ms.</option>
</select><input type="hidden" name="form-0-id" id="id_form-0-id" /></td></tr>
```


Note

[modelformset_factory\(\)](#) utilise [formset_factory\(\)](#) pour générer les formulaires groupés. Cela signifie que les formulaires groupés de modèle ne sont qu'une extension des formulaires groupés de base, mais qui savent comment interagir avec un modèle particulier.

Modification du jeu de requête

Par défaut, lorsque vous créez des formulaires groupés à partir d'un modèle, ceux-ci utilisent un jeu de requête qui contient tous les objets du modèle (par ex. `Author.objects.all()`). Vous pouvez surcharger ce comportement en utilisant le paramètre `queryset`:

```
>>> formset = AuthorFormSet(queryset=Author.objects.filter(name__startswith='O'))
```

Comme variante, vous pouvez créer une sous-classe définissant `self.queryset` dans sa méthode `__init__`:

```
from django.forms import BaseModelFormSet
from myapp.models import Author

class BaseAuthorFormSet(BaseModelFormSet):
    def __init__(self, *args, **kwargs):
        super(BaseAuthorFormSet, self).__init__(*args, **kwargs)
        self.queryset = Author.objects.filter(name__startswith='O')
```

Puis, passez votre classe `BaseAuthorFormSet` à la fonction de fabrique :

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=('name', 'title'), formset=BaseAuthorFormSet)
```

Si vous souhaitez renvoyer des formulaires groupés qui ne contiennent *aucune* instance préexistante du modèle, vous pouvez indiquer un objet `QuerySet` vide :

```
>>> AuthorFormSet(queryset=Author.objects.none())
```

Modification du formulaire

Par défaut, lorsque vous utilisez `modelformset_factory`, un formulaire de modèle est créé par [modelform_factory\(\)](#). Il peut souvent être utile d'indiquer un formulaire de modèle personnalisé. Par exemple, vous pouvez créer un formulaire de modèle personnalisé comportant une validation personnalisée :

```
class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title')

    def clean_name(self):
        # custom validation for the name field
        ...
```

Puis, indiquez ce formulaire de modèle à la fonction de fabrication :

```
AuthorFormSet = modelformset_factory(Author, form=AuthorForm)
```

Il n'est pas toujours nécessaire de définir un formulaire de modèle personnalisé. La fonction `modelformset_factory` accepte plusieurs paramètres qui sont eux-mêmes transmis à `modelform_factory` et qui sont documentés plus bas.

Indication des composants de formulaire à utiliser avec `widgets`

En exploitant le paramètre `widgets`, vous pouvez indiquer un dictionnaire de valeurs afin de personnaliser la classe de composant d'un champ particulier du `ModelForm`. C'est le même principe de fonctionnement que le dictionnaire `widgets` de la classe `Meta` interne d'un `ModelForm`:

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=('name', 'title'),
...     widgets={'name': Textarea(attrs={'cols': 80, 'rows': 20})})
```

Activation de la régionalisation des champs avec `localized_fields`

À l'aide du paramètre `localized_fields`, vous pouvez activer la régionalisation des champs du formulaire.

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=('name', 'title', 'birth_date'),
...     localized_fields=('birth_date',))
```

Si `localized_fields` est défini à la valeur spéciale `'__all__'`, les valeurs de tous les champs seront régionalisées.

Indication de valeurs initiales

Comme pour les formulaires groupés normaux, il est possible [de fournir des données initiales](#) aux formulaires groupés en indiquant le paramètre `initial` au moment de la création de l'instance de classe de formulaires groupés de modèles renvoyée par [modelformset_factory\(\)](#). Cependant, avec les formulaires groupés de modèles, les valeurs initiales ne s'appliquent qu'aux formulaires supplémentaires, ceux qui ne sont pas liés à des instances d'objets existants. Si les formulaires avec données initiales ne sont pas modifiés par l'utilisateur, ils ne seront ni validés, ni enregistrés.

Enregistrement des objets des formulaires groupés

Comme pour un `ModelForm`, vous pouvez enregistrer les données sous forme d'objet de modèle. Cela se fait par la méthode `save()` des formulaires groupés :

```
# Create a formset instance with POST data.
>>> formset = AuthorFormSet(request.POST)

# Assuming all is valid, save the data.
>>> instances = formset.save()
```

La méthode `save()` renvoie les instances qui ont été enregistrées dans la base de données. Si les données d'une instance précise n'ont pas été modifiées dans les données du formulaire, l'instance ne sera pas enregistrée dans la base de données et ne sera pas non plus incluse dans la valeur renvoyée (instances, dans l'exemple ci-dessus).

Lorsque des champs sont absents du formulaire (par exemple en raison de leur exclusion), ces champs ne sont pas modifiés par la méthode `save()`. Vous pouvez trouver plus d'informations sur cette restriction, qui est aussi valable pour les formulaires `ModelForm` normaux, dans [Sélection des champs à utiliser](#).

Passez `commit=False` pour recevoir les instances de modèles non enregistrées :

```
# don't save to the database
>>> instances = formset.save(commit=False)
>>> for instance in instances:
...     # do something with instance
...     instance.save()
```

Cela vous permet de définir des données complémentaires dans les instances avant de les enregistrer dans la base de données. Si vos formulaires groupés contiennent un champ `ManyToManyField`, vous devrez également appeler `formset.save_m2m()` pour vous assurer que les liens plusieurs-à-plusieurs soient correctement enregistrés.

Après l'appel à `save()`, les formulaires groupés du modèle posséderont trois nouveaux attributs contenant les modifications des formulaires :

`models.BaseModelFormSet.changed_objects`

`models.BaseModelFormSet.deleted_objects`

`models.BaseModelFormSet.new_objects`

Restriction du nombre d'objets modifiables

Comme pour les formulaires groupés normaux, vous pouvez utiliser les paramètres `max_num` et `extra` de [modelformset_factory\(\)](#) pour restreindre le nombre de formulaires supplémentaires affichés.

`max_num` n'empêche pas les objets existants d'être affichés :

```
>>> Author.objects.order_by('name')
[<Author: Charles Baudelaire>, <Author: Paul Verlaine>, <Author: Walt Whitman>]

>>> AuthorFormSet = modelformset_factory(Author, fields=('name',), max_num=1)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('name'))
>>> [x.name for x in formset.get_queryset()]
['Charles Baudelaire', 'Paul Verlaine', 'Walt Whitman']
```

De plus, `extra=0` n'empêche pas la création de nouvelles instances de modèles car il est possible d'[ajouter des formulaires supplémentaires par JavaScript](#) ou simplement d'envoyer des données POST supplémentaires. Les jeux de formulaires [ne fournissent pas encore la fonctionnalité](#) d'une vue uniquement en mode édition qui empêcherait la création de nouvelles instances.

Si la valeur de `max_num` est plus grande que le nombre d'objets liés existants, un maximum de `extra` formulaires vierges supplémentaires seront ajoutés aux formulaires groupés, aussi longtemps que le nombre total de formulaires n'excède pas `max_num`:

```
>>> AuthorFormSet = modelformset_factory(Author, fields=('name',), max_num=4,
extra=2)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('name'))
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-
name" type="text" name="form-0-name" value="Charles Baudelaire" maxlength="100"
/><input type="hidden" name="form-0-id" value="1" id="id_form-0-id" /></td></tr>
<tr><th><label for="id_form-1-name">Name:</label></th><td><input id="id_form-1-
name" type="text" name="form-1-name" value="Paul Verlaine" maxlength="100" /><input
type="hidden" name="form-1-id" value="3" id="id_form-1-id" /></td></tr>
<tr><th><label for="id_form-2-name">Name:</label></th><td><input id="id_form-2-
name" type="text" name="form-2-name" value="Walt Whitman" maxlength="100" /><input
type="hidden" name="form-2-id" value="2" id="id_form-2-id" /></td></tr>
<tr><th><label for="id_form-3-name">Name:</label></th><td><input id="id_form-3-
name" type="text" name="form-3-name" maxlength="100" /><input type="hidden"
name="form-3-id" id="id_form-3-id" /></td></tr>
```

Une valeur `max_num` `NONE` (par défaut) place une limite élevée du nombre de formulaires affichés (1000). En pratique, cela équivaut à aucune limite.

Utilisation de formulaires groupés de modèle dans une vue

Les formulaires groupés de modèle sont très semblables aux formulaires groupés normaux. Admettons que nous voulions afficher des formulaires groupés pour modifier les instances du modèle `Author`:

```
from django.forms import modelformset_factory
from django.shortcuts import render
from myapp.models import Author

def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author, fields=('name', 'title'))
    if request.method == 'POST':
        formset = AuthorFormSet(request.POST, request.FILES)
        if formset.is_valid():
            formset.save()
            # do something.
    else:
        formset = AuthorFormSet()
    return render(request, 'manage_authors.html', {'formset': formset})
```

Comme vous pouvez le voir, la logique de la vue de formulaires groupés de modèle n'est pas fondamentalement différente de celle de formulaires groupés normaux. La seule différence est que nous

appelons `formset.save()` pour enregistrer les données dans la base de données (comme cela a été décrit ci-dessus dans [Enregistrement des objets des formulaires groupés](#)).

Surcharge de `clean()` d'un `ModelFormSet`

Tout comme pour `ModelForms`, la méthode `clean()` de `ModelFormSet` valide par défaut qu'aucun des éléments dans les formulaires groupés ne viole les contraintes d'unicité de votre modèle (que ce soit `unique`, `unique_together` ou `unique_for_date|month|year`). Si vous souhaitez surcharger la méthode `clean()` d'un `ModelFormSet` et conserver cette validation, vous devez appeler la méthode `clean` de la classe parente :

```
from django.forms import BaseModelFormSet

class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super(MyModelFormSet, self).clean()
        # example custom validation across forms in the formset
        for form in self.forms:
            # your custom formset validation
        ...
```

Notez également qu'une fois que vous avez atteint cette étape, les instances de modèles individuelles pour chaque formulaire ont déjà été créées. Il ne suffit donc pas de modifier une valeur dans `form.cleaned_data` pour changer la valeur enregistrée. Si vous souhaitez modifier une valeur dans `ModelFormSet.clean()`, vous devez modifier `form.instance`:

```
from django.forms import BaseModelFormSet

class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super(MyModelFormSet, self).clean()

        for form in self.forms:
            name = form.cleaned_data['name'].upper()
            form.cleaned_data['name'] = name
            # update the instance value.
            form.instance.name = name
```

Utilisation d'un jeu de requête personnalisé

Comme indiqué précédemment, il est possible de surcharger le jeu de requête utilisé par défaut par les formulaires groupés de modèle :

```
from django.forms import modelformset_factory
from django.shortcuts import render
from myapp.models import Author

def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author, fields=('name', 'title'))
    if request.method == "POST":
        formset = AuthorFormSet(
            request.POST, request.FILES,
```

```

        queryset=Author.objects.filter(name__startswith='O'),
    )
    if formset.is_valid():
        formset.save()
        # Do something.
    else:
        formset =
AuthorFormSet(queryset=Author.objects.filter(name__startswith='O'))
    return render(request, 'manage_authors.html', {'formset': formset})

```

Notez que nous transmettons le paramètre `queryset` à la fois aux requêtes POST et GET dans cet exemple.

Utilisation des formulaires groupés dans un gabarit

Il y a trois façons d’afficher des formulaires groupés dans un gabarit Django.

Premièrement, vous pouvez déléguer aux formulaires groupés l’essentiel du travail :

```

<form method="post" action="">
    {{ formset }}
</form>

```

Deuxièmement, vous pouvez afficher les formulaires groupés manuellement, mais laisser le soin à chaque formulaire de s’afficher lui-même :

```

<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form }}
    {% endfor %}
</form>

```

Lorsque vous vous occupez vous-même d’afficher les différents formulaires, prenez soin d’inclure également le formulaire de gestion comme le montre l’exemple ci-dessus. Voir la [documentation du formulaire de gestion](#).

Troisièmement, vous pouvez afficher manuellement chaque champ :

```

<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {% for field in form %}
            {{ field.label_tag }} {{ field }}
        {% endfor %}
    {% endfor %}
</form>

```

Si vous optez pour cette troisième méthode et que vous ne bouclez pas sur les champs avec un `{% for %}`, vous devez aussi inclure le champ de clé primaire. Par exemple, si vous affichez les champs `name` et `age` d’un modèle :

```

<form method="post" action="">
    {{ formset.management_form }}

```

```

{% for form in formset %}
    {{ form.id }}
    <ul>
        <li>{{ form.name }}</li>
        <li>{{ form.age }}</li>
    </ul>
{% endfor %}
</form>

```

Voyez comment nous devons explicitement afficher `{{ form.id }}`. Ceci pour s'assurer du bon fonctionnement des formulaires groupés de modèle, dans le cas de l'envoi POST (cet exemple suppose que la clé primaire s'appelle `id`. Si vous avez explicitement nommé la clé primaire du modèle autrement que `id`, prenez soin de l'afficher).

Sous-formulaires groupés

```
class models.BaseInlineFormSet
```

Les sous-formulaires groupés sont une petite couche d'abstraction au-dessus des formulaires groupés de modèle. Ils simplifient la situation où l'on manipule des objets liés au travers d'une clé étrangère. Supposons que l'on dispose de ces deux modèles :

```

from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)

```

Si vous voulez créer des formulaires groupés permettant de modifier les livres appartenant à un auteur particulier, vous pourriez faire ainsi :

```

>>> from django.forms import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book, fields=('title',))
>>> author = Author.objects.get(name='Mike Royko')
>>> formset = BookFormSet(instance=author)

```

Note

[`inlineformset_factory\(\)`](#) utilise [`modelformset_factory\(\)`](#) et définit `can_delete=True`.

Voir aussi

[can_delete et can_order affichés manuellement.](#)

Surcharge des méthodes d'un InlineFormSet

Lorsque vous surchargez des méthodes de sous-formulaires groupés (InlineFormSet), il est préférable d'hériter de [BaseInlineFormSet](#) plutôt que de [BaseModelFormSet](#).

Par exemple, si vous souhaitez surcharger `clean()`:

```
from django.forms import BaseInlineFormSet

class CustomInlineFormSet(BaseInlineFormSet):
    def clean(self):
        super(CustomInlineFormSet, self).clean()
        # example custom validation across forms in the formset
        for form in self.forms:
            # your custom formset validation
        ...
```

Voir aussi [Surcharge de clean\(\) d'un ModelFormSet](#).

Puis, au moment de créer les sous-formulaires groupés, passez le paramètre facultatif `formset`:

```
>>> from django.forms import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book, fields=('title',),
...     formset=CustomInlineFormSet)
>>> author = Author.objects.get(name='Mike Royko')
>>> formset = BookFormSet(instance=author)
```

Plus d'une clé étrangère vers le même modèle

Si votre modèle contient plus d'une clé étrangère vers le même modèle, vous devrez résoudre l'ambiguïté manuellement en utilisant `fk_name`. Par exemple, considérez le modèle suivant :

```
class Friendship(models.Model):
    from_friend = models.ForeignKey(
        Friend,
        on_delete=models.CASCADE,
        related_name='from_friends',
    )
    to_friend = models.ForeignKey(
        Friend,
        on_delete=models.CASCADE,
        related_name='friends',
    )
    length_in_months = models.IntegerField()
```

Pour résoudre cela, vous pouvez définir `fk_name` dans [inlineformset_factory\(\)](#):

```
>>> FriendshipFormSet = inlineformset_factory(Friend, Friendship,
fk_name='from_friend',
...     fields=('to_friend', 'length_in_months'))
```


Utilisation de sous-formulaires groupés dans une vue

Il peut être nécessaire de définir une vue permettant à un utilisateur de modifier les objets liés d'un modèle. Voici comment on peut le faire :

```
def manage_books(request, author_id):
    author = Author.objects.get(pk=author_id)
    BookInlineFormSet = inlineformset_factory(Author, Book, fields=('title',))
    if request.method == "POST":
        formset = BookInlineFormSet(request.POST, request.FILES, instance=author)
        if formset.is_valid():
            formset.save()
            # Do something. Should generally end with a redirect. For example:
            return HttpResponseRedirect(author.get_absolute_url())
    else:
        formset = BookInlineFormSet(instance=author)
    return render(request, 'manage_books.html', {'formset': formset})
```

Notez comment nous définissons le paramètre `instance` aussi bien dans le cas POST que le cas GET.

Indication des composants à utiliser dans le sous-formulaire

`inlineformset_factory` utilise `modelformset_factory` et transmet la plupart de ses paramètres à `modelformset_factory`. Cela signifie que vous pouvez utiliser le paramètre `widgets` tout comme vous le feriez pour `modelformset_factory`. Voir [Indication des composants de formulaire à utiliser avec widgets](#) ci-dessus.

[Formulaires groupés](#)

[Fichiers annexes de formulaire \(classe Media\)](#)

Fichiers annexes de formulaire (classe Media)

L'affichage d'un formulaire Web attractif et ergonomique exige plus que du code HTML simple, il nécessite également des feuilles de style CSS, et si vous voulez utiliser des composants « Web 2.0 » plaisants, il sera peut-être nécessaire d'inclure aussi du code JavaScript sur chaque page. La combinaison exacte de CSS et de JavaScript nécessaire pour une page donnée dépend des composants de formulaire employés sur cette page.

C'est là où la définition de fichiers annexes intervient. Django permet d'associer différents fichiers comme des feuilles de style et des scripts à des formulaires et des composants qui ont besoin de ces fichiers. Par exemple, si vous voulez utiliser un calendrier pour afficher les champs date, vous pouvez définir un composant `Calendrier` personnalisé. Ce composant peut ensuite être associé à des fichiers CSS et JavaScript requis pour afficher le calendrier. Quand le composant `Calendrier` est utilisé dans un formulaire, Django est capable d'identifier les fichiers CSS et JavaScript nécessaires et

de fournir la liste des noms de fichiers au formulaire de manière à ce qu'ils puissent être facilement inclus dans la page Web.

Fichiers annexes dans l'administration de Django

L'application d'administration de Django définit un certain nombre de composants personnalisés pour les calendriers, les sélections filtrées, etc. Ces composants définissent des exigences de fichiers annexes et l'administration de Django utilise ces composants personnalisés au lieu des composants par défaut de Django. Les gabarits de l'administration n'incluent que les fichiers nécessaires pour l'affichage des composants de la page en cours.

Si vous appréciez les composants utilisés par le site d'administration de Django, vous pouvez librement les utiliser dans votre propre application ! Ils se trouvent dans `django.contrib.admin.widgets`.

Quelle bibliothèque JavaScript ?

Il existe plusieurs bibliothèques JavaScript et beaucoup d'entre elles contiennent des composants utiles pour améliorer l'interface d'une application (comme par exemple des composants de calendrier). Django a délibérément évité de choisir parmi ces bibliothèques JavaScript. Chacune a ses forces et ses faiblesses, utilisez donc celle qui correspond à vos besoins. Django est capable d'intégrer n'importe quelle bibliothèque JavaScript.

Fichiers annexes définis statiquement

La façon la plus simple de définir des fichiers annexes est sous forme de définition statique. Avec cette méthode, la déclaration se fait dans une classe `Media` interne. Les propriétés de la classe interne définissent les exigences.

Voici un exemple simple :

```
from django import forms

class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            'all': ('pretty.css',)
        }
        js = ('animations.js', 'actions.js')
```

Ce code définit un composant `CalendarWidget`, qui est basé sur `TextInput`. Chaque fois que `CalendarWidget` sera utilisé dans un formulaire, ce dernier sera amené à inclure le fichier CSS `pretty.css` et les fichiers JavaScript `animations.js` et `actions.js`.

Cette définition statique est convertie au moment de l'exécution en une propriété de composant nommée `media`. La liste des fichiers annexes d'une instance de composant `CalendarWidget` peut être obtenue par cette propriété :

```
>>> w = CalendarWidget()
```

```
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all"
rel="stylesheet" />
<script type="text/javascript"
src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
```

Voici une liste de toutes les options possibles de Media. Il n’y a pas d’option obligatoire.

CSS

Un dictionnaire décrivant les fichiers CSS requis par les différentes formes de support d’affichage.

Les valeurs du dictionnaire doivent être composées de tuples/listes de noms de fichiers. Consultez [la section sur les chemins](#) pour plus de détails sur la manière de définir les chemins de ces fichiers.

Les clés du dictionnaire correspondent aux types de support d’affichage. Ce sont les mêmes types que les fichiers CSS acceptent dans les déclarations « media » : ‘all’, ‘aural’, ‘braille’, ‘embossed’, ‘handheld’, ‘print’, ‘projection’, ‘screen’, ‘tty’ and ‘tv’. Si vous avez besoin de feuilles de style différentes en fonction de types de support différents, indiquez des listes de fichiers CSS correspondant à chacun de ces types. L’exemple suivant fournit deux options CSS, une pour l’affichage à l’écran et l’autre pour l’impression :

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'print': ('newspaper.css',)
    }
```

Si un groupe de fichiers CSS convient à plusieurs types de support d’affichage, la clé de dictionnaire peut être une liste de types de support d’affichage séparés par des virgules. Dans l’exemple suivant, les télévisions et les projecteurs ont les mêmes exigences en terme de support :

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'tv,projector': ('lo_res.css',),
        'print': ('newspaper.css',)
    }
```

Si cette dernière définition de CSS devait être affichée, cela produirait le code HTML suivant :

```
<link href="http://static.example.com/pretty.css" type="text/css" media="screen"
rel="stylesheet" />
<link href="http://static.example.com/lo_res.css" type="text/css"
media="tv,projector" rel="stylesheet" />
<link href="http://static.example.com/newspaper.css" type="text/css" media="print"
rel="stylesheet" />
```

js

Un tuple décrivant les fichiers JavaScript nécessaires. Consultez [la section sur les chemins](#) pour plus de détails sur la manière de définir les chemins de ces fichiers.

extend

Une valeur booléenne définissant le comportement de l'héritage des déclarations `Media`.

Par défaut, tout objet utilisant une définition statique de `Media` hérite de tous les fichiers annexes associés au composant parent, et ceci quelle que soit la manière dont le parent définit ses propres exigences. Par exemple, si nous devons améliorer notre composant `Calendar` basique de l'exemple ci-dessus :

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         css = {
...             'all': ('fancy.css',)
...         }
...         js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all"
rel="stylesheet" />
<link href="http://static.example.com/fancy.css" type="text/css" media="all"
rel="stylesheet" />
<script type="text/javascript"
src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript"
src="http://static.example.com/whizbang.js"></script>
```

Le composant `FancyCalendar` hérite de tous les fichiers annexes de son composant parent. Si vous ne souhaitez pas que `Media` soit hérité de cette façon, ajoutez une déclaration `extend=False` à la déclaration de `Media`:

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         extend = False
...         css = {
...             'all': ('fancy.css',)
...         }
...         js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/fancy.css" type="text/css" media="all"
rel="stylesheet" />
<script type="text/javascript"
src="http://static.example.com/whizbang.js"></script>
```

Si vous avez besoin de pouvoir contrôler encore plus l'héritage, définissez vos fichiers annexes en utilisant une [propriété dynamique](#). Les propriétés dynamiques vous donnent un contrôle total sur les fichiers hérités et ceux qui ne le seront pas.

Media comme propriété dynamique

Si vous avez besoin d'effectuer certaines manipulations encore plus sophistiquées sur les fichiers annexes requis, vous pouvez définir directement la propriété `media`. Cela peut être réalisé en définissant une propriété du composant qui renvoie une instance de `forms.Media`. Le constructeur de `forms.Media` accepte les paramètres nommés `css` and `js` dans le même format que celui qui est utilisé pour les définitions statiques de fichiers annexes.

Par exemple, la définition statique de notre composant de calendrier pourrait tout aussi bien être réalisée de manière dynamique :

```
class CalendarWidget(forms.TextInput):
    def _media(self):
        return forms.Media(css={'all': ('pretty.css',)},
                           js=('animations.js', 'actions.js'))
    media = property(_media)
```

Consultez la section sur les [objets Media](#) pour plus de détails sur la manière de construire des valeurs de retour pour les propriétés dynamiques `media`.

Chemins dans les définitions de fichiers annexes

Les chemins utilisés pour définir les fichiers annexes peuvent être relatifs ou absolus. Si un chemin commence par `/`, `http://` ou `https://`, il sera interprété comme un chemin absolu et laissé tel quel. Tous les autres chemins seront préfixés par la valeur appropriée.

Dans le cadre de l'introduction de l'[application staticfiles](#), deux nouveaux réglages ont été ajoutés pour faire référence aux « fichiers statiques » (images, CSS, JavaScript, etc.) nécessaires à l'affichage d'une page Web complète : [STATIC_URL](#) et [STATIC_ROOT](#).

Pour trouver le bon préfixe à utiliser, Django contrôle si le réglage [STATIC_URL](#) est différent de `None` et se rabat sur [MEDIA_URL](#) en cas de besoin. Par exemple, si le réglage [MEDIA_URL](#) de votre site est `'http://uploads.example.com/'` et que [STATIC_URL](#) vaut `None`:

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             'all': ('/css/pretty.css',),
...         }
...         js = ('animations.js', 'http://othersite.com/actions.js')

>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
```

```
<script type="text/javascript"
src="http://uploads.example.com/animations.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

Mais si [STATIC_URL](#) vaut 'http://static.example.com/':

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript"
src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

Objets Media

Lorsque vous interrogez l'attribut `media` d'un composant ou d'un formulaire, la valeur renvoyée est un objet `forms.Media`. Comme nous l'avons déjà vu, la représentation textuelle d'un objet `Media` contient le code HTML nécessaire pour inclure les fichiers concernés dans le bloc `<head>` de votre page HTML.

Cependant, les objets `Media` ont certaines autres propriétés intéressantes.

Sous-ensembles de fichiers annexes

Si vous ne souhaitez obtenir que les fichiers d'un type particulier, vous pouvez utiliser l'opérateur d'indice pour filtrer le support qui vous intéresse. Par exemple :

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all"
rel="stylesheet" />
<script type="text/javascript"
src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>

>>> print(w.media['css'])
<link href="http://static.example.com/pretty.css" type="text/css" media="all"
rel="stylesheet" />
```

Lorsque vous employez l'opérateur d'indice, la valeur renvoyée est un nouvel objet `Media`, mais qui ne contient que les fichiers qui vous intéressent.

Combinaison d'objets Media

Les objets `Media` peuvent aussi être fusionnés. Lorsque l'on additionne deux objets `Media`, l'objet `Media` résultant contient l'union des fichiers annexes contenus dans les deux objets :

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             'all': ('pretty.css',)
```

```

...         }
...         js = ('animations.js', 'actions.js')

>>> class OtherWidget(forms.TextInput):
...     class Media:
...         js = ('whizbang.js',)

>>> w1 = CalendarWidget()
>>> w2 = OtherWidget()
>>> print(w1.media + w2.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all"
rel="stylesheet" />
<script type="text/javascript"
src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript"
src="http://static.example.com/whizbang.js"></script>

```

Media pour les formulaires

Les composants ne sont pas les seuls objets qui peuvent posséder des définitions `media`, les formulaires peuvent aussi en définir. Les règles concernant les définitions `media` des formulaires sont les mêmes que pour les composants : les déclarations peuvent être statiques ou dynamiques ; les règles des chemins et de l'héritage de ces déclarations sont également identiques.

Que vous définissiez une déclaration `media` ou non, *tous* les objets `Form` possèdent une propriété `media`. La valeur par défaut de cette propriété est le résultat de la fusion de toutes les définitions `media` de tous les composants du formulaire :

```

>>> from django import forms
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)

>>> f = ContactForm()
>>> f.media
<link href="http://static.example.com/pretty.css" type="text/css" media="all"
rel="stylesheet" />
<script type="text/javascript"
src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript"
src="http://static.example.com/whizbang.js"></script>

```

Si vous souhaitez associer des fichiers annexes supplémentaires à un formulaire, par exemple des règles CSS pour la mise en forme du formulaire, ajoutez simplement une déclaration `Media` au formulaire :

```

>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)
...
...     class Media:

```

```

...         css = {
...             'all': ('layout.css',)
...         }

>>> f = ContactForm()
>>> f.media
<link href="http://static.example.com/pretty.css" type="text/css" media="all"
rel="stylesheet" />
<link href="http://static.example.com/layout.css" type="text/css" media="all"
rel="stylesheet" />
<script type="text/javascript"
src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript"
src="http://static.example.com/whizbang.js"></script>

```

[Création de formulaires à partir de modèles](#)
[Gabarits](#)

Gabarits

Par sa nature liée au Web, Django a besoin d'un procédé agile de génération dynamique de HTML. L'approche la plus couramment utilisée est de se baser sur des gabarits. Un gabarit contient la partie statique du résultat HTML souhaité ainsi qu'une certaine syntaxe particulière définissant comment insérer le contenu dynamique. Pour un exemple pratique de création de pages HTML avec des gabarits, consultez la [partie 3 du tutoriel](#).

Un projet Django peut être configuré avec un ou plusieurs moteurs de gabarit (ou même aucun si vous n'utilisez pas de gabarit). Django est livré avec des moteurs intégrés pour son propre système de gabarits, appelé de manière originale le langage de gabarits de Django (DTL), ainsi que pour l'alternative répandue [Jinja2](#). Des moteurs pour d'autres langages de gabarits peuvent être mis à disposition par des applications tierces.

Django définit une API standard pour le chargement et la production de gabarits de manière indépendante du moteur utilisé. Le chargement consiste à trouver le gabarit correspondant à un identifiant donné et à le pré-traiter, ce qui revient généralement à le compiler dans une représentation en mémoire. La production consiste à interpoler le gabarit en fonction de données de contexte et à renvoyer le texte résultant.

Le [langage de gabarits de Django](#) est le système de gabarits propre à Django. Jusqu'à Django 1.8, il s'agissait de la seule option disponible. C'est une bonne bibliothèque de gabarit, même si son approche est parfois un peu rigide et présente quelques particularités. SI vous n'avez pas de motivation précise pour choisir un moteur différent, nous vous recommandons de travailler avec ce langage, à plus forte raison si vous écrivez une application réutilisable et que vous pensez distribuer des gabarits. Les

applications contribuéées de Django qui comprennent des gabarits, telle que [django.contrib.admin](#), utilisent le moteur de Django.

Pour des raisons historiques, la prise en charge générique des moteurs de gabarit et l'implémentation du langage de gabarit de Django se trouvent tous deux dans l'espace de noms `django.template`.

Prise en charge des moteurs de gabarit

New in Django 1.8:

La prise en charge de plusieurs moteurs de gabarit et le réglage [TEMPLATES](#) ont été ajoutés dans Django 1.8.

Configuration

Les moteurs de gabarit sont configurés dans le réglage [TEMPLATES](#). Il s'agit d'une liste de configurations, une par moteur. La valeur par défaut est vide. Le fichier `settings.py` généré par la commande [startproject](#) définit une valeur plus utile :

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            # ... some options here ...
        },
    },
]
```

[BACKEND](#) est un chemin Python pointé vers une classe de moteur de gabarit implémentant l'API de moteur de gabarit de Django. Les moteurs intégrés sont [django.template.backends.django.DjangoTemplates](#) et [django.template.backends.jinja2.Jinja2](#).

Comme la plupart des moteurs chargent des gabarits à partir de fichiers, la configuration de chaque moteur contient au premier niveau deux réglages courants :

- [DIRS](#) définit une liste de répertoires dans lesquels le moteur recherche des fichiers sources de gabarits, dans l'ordre de leur recherche.
- [APP_DIRS](#) indique si le moteur doit rechercher les gabarits dans les applications installées. Chaque moteur définit un nom conventionnel à attribuer au sous-répertoire des applications où ses gabarits devraient se trouver.

Même si ce n'est pas fréquent, il est possible de configurer plusieurs instances d'un même moteur avec des options différentes. Dans ce cas, il est nécessaire de définir un nom [NAME](#) unique pour chaque moteur.

[OPTIONS](#) contient des réglages spécifiques au moteur.

Utilisation

Le module `django.template.loader` définit deux fonctions pour charger des gabarits.

`get_template(template_name, dirs=_dirs_undefined, using=None)`[\[source\]](#)

Cette fonction charge le gabarit ayant le nom donné et renvoie un objet `Template`.

Le type exact de la valeur renvoyée dépend du moteur qui a chargé le gabarit. Chaque moteur possède sa propre classe `Template`.

`get_template()` essaie avec chaque moteur de gabarit dans l'ordre jusqu'à ce que l'un d'eux réussisse. Si le gabarit n'est pas trouvé, une exception [TemplateDoesNotExist](#) est générée. Si le gabarit est trouvé mais contient une syntaxe non valable, une exception [TemplateSyntaxError](#) est générée.

La façon dont les gabarits sont recherchés et chargés dépend de chaque moteur et de sa configuration.

Si vous souhaitez restreindre la recherche à un moteur de gabarit particulier, passez le nom [NAME](#) du moteur dans le paramètre `using`.

Obsolète depuis la version 1.8: Le paramètre `dirs` a été rendu obsolète.

Changed in Django 1.8:

Le paramètre `using` a été ajouté.

Changed in Django 1.8:

`get_template()` renvoie un objet `Template` dépendant du moteur au lieu de [django.template.Template](#).

`select_template(template_name_list, dirs=_dirs_undefined, using=None)`[\[source\]](#)

`select_template()` est semblable à `get_template()`, sauf qu'il accepte une liste de noms de gabarits. Il essaie chaque nom dans l'ordre et renvoie le premier gabarit existant.

Obsolète depuis la version 1.8: Le paramètre `dirs` a été rendu obsolète.

Changed in Django 1.8:

Le paramètre `using` a été ajouté.

Changed in Django 1.8:

`select_template()` renvoie un objet `Template` dépendant du moteur au lieu de [`django.template.Template`](#).

Si le chargement d'un gabarit échoue, les deux exceptions suivantes définies dans `django.template` peuvent être générées :

exception `TemplateDoesNotExist(msg, tried=None, backend=None, chain=None)`[\[source\]](#)

Cette exception est générée lorsqu'aucun gabarit n'a pu être trouvé. Elle accepte les paramètres facultatifs suivants pour remplir le [gabarit postmortem](#) sur la page de débogage :

`backend`

L'instance de moteur de gabarit dans lequel l'exception s'est produite.

`tried`

Une liste de sources qui ont été parcourues pour la recherche du gabarit. Elle se trouve sous la forme d'une liste de tuples contenant (`origine`, `statut`), où `origine` est un objet de [type origine](#) et `statut` est une chaîne contenant la raison expliquant l'absence du gabarit.

`chain`

Une liste d'exceptions intermédiaires [TemplateDoesNotExist](#) générées durant le chargement du gabarit. Ceci est exploité par des fonctions comme [get_template\(\)](#), qui essaient de charger un gabarit donné depuis plusieurs moteurs.

New in Django 1.9:

Les paramètres `backend`, `tried` et `chain` ont été ajoutés.

exception `TemplateSyntaxError(msg)`[\[source\]](#)

Cette exception est générée lorsqu'un gabarit a été trouvé, mais qu'il contient des erreurs.

Les objets `Template` renvoyés par `get_template()` et `select_template()` doivent fournir une méthode `render()` ayant la signature suivante :

`Template.render(context=None, request=None)`

Produit ce gabarit en fonction du contexte donné.

Si `context` est fourni, il doit s'agir d'un [dict](#). S'il n'est pas fourni, le moteur va produire le gabarit avec un contexte vide.

Si `request` est fourni, il doit s'agir d'un objet [HttpRequest](#). Le moteur doit se charger ensuite de le rendre disponible, ainsi que le jeton CSRF, dans le gabarit. La manière de le faire dépend de chaque moteur.

Voici un exemple de l'algorithme de recherche. Pour cet exemple, le réglage [TEMPLATES](#) est :

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            '/home/html/example.com',
            '/home/html/default',
        ],
    },
    {
        'BACKEND': 'django.template.backends.jinja2.Jinja2',
        'DIRS': [
            '/home/html/jinja2',
        ],
    },
]
```

Si vous appelez `get_template('story_detail.html')`, voici les fichiers que Django va rechercher, dans l'ordre :

- `/home/html/example.com/story_detail.html` (moteur 'django')
- `/home/html/default/story_detail.html` (moteur 'django')
- `/home/html/jinja2/story_detail.html` (moteur 'jinja2')

Si vous appelez `select_template(['story_253_detail.html', 'story_detail.html'])`, voici ce que Django va rechercher :

- `/home/html/example.com/story_253_detail.html` (moteur 'django')
- `/home/html/default/story_253_detail.html` (moteur 'django')
- `/home/html/jinja2/story_253_detail.html` (moteur 'jinja2')
- `/home/html/example.com/story_detail.html` (moteur 'django')
- `/home/html/default/story_detail.html` (moteur 'django')
- `/home/html/jinja2/story_detail.html` (moteur 'jinja2')

Lorsque Django trouve un gabarit existant, il stoppe sa recherche.

Astuce

Vous pouvez utiliser [select_template\(\)](#) pour une sélection agile des gabarits. Par exemple, si vous avez rédigé un article et que vous voulez pouvoir utiliser des gabarits spécifiques pour certains articles, utilisez quelque chose comme `select_template(['story_%s_detail.html' %`

`story.id, 'story_detail.html']`). Cela vous permet d'utiliser un gabarit adapté à un article individuel, tout en se rabattant sur un gabarit standard pour les articles sans gabarit dédié.

Il est possible – et préférable – d'organiser les gabarits dans des sous-répertoires de chaque répertoire contenant des gabarits. La convention est de créer un sous-répertoire par application Django, en y ajoutant d'autres sous-répertoires au besoin.

Ce conseil est tout à votre avantage. Le placement de tous les gabarits au niveau racine d'un seul répertoire devient rapidement ingérable.

Pour charger un gabarit se trouvant dans un sous-répertoire, il suffit d'utiliser la barre oblique, comme ceci :

```
get_template('news/story_detail.html')
```

Toujours avec le même réglage [TEMPLATES](#) que ci-dessus, cet exemple va tenter de charger les gabarits suivants :

- `/home/html/example.com/news/story_detail.html` (moteur 'django')
- `/home/html/default/news/story_detail.html` (moteur 'django')
- `/home/html/jinja2/news/story_detail.html` (moteur 'jinja2')

De plus, pour rationaliser l'aspect répétitif du chargement et de la production des gabarits, Django propose une fonction raccourci qui automatise le processus.

```
render_to_string(template_name, context=None,  
context_instance=_context_instance_undefined, request=None, using=None)\[source\]
```

`render_to_string()` charge un gabarit comme [get_template\(\)](#) et appelle sa méthode `render()` immédiatement. Elle accepte les paramètres ci-après.

`template_name`

Le nom d'un gabarit à charger et à produire. S'il s'agit d'une liste de noms de gabarits, Django utilise [select_template\(\)](#) au lieu de [get_template\(\)](#) pour chercher le gabarit.

`context`

Un [dict](#) à utiliser comme contexte de gabarit lors de la production.

Changed in Django 1.8:

Le paramètre `context` était appelé `dictionary`. Ce nom est obsolète dans Django 1.8 et sera supprimé dans Django 1.10.

`context` est dorénavant facultatif. Un contexte vide sera utilisé s'il n'est pas défini.

context_instance

Une instance de [Context](#) ou une sous-classe (par ex. une instance de [RequestContext](#)) à utiliser comme contexte de gabarit.

Obsolète depuis la version 1.8: Le paramètre `context_instance` est obsolète. Utilisez `context` et, si nécessaire, `request`.

request

Un objet [HttpRequest](#) facultatif qui sera disponible durant le processus de rendu du gabarit.

New in Django 1.8:

Le paramètre `request` a été ajouté.

Usage example:

```
from django.template.loader import render_to_string
rendered = render_to_string('my_template.html', {'foo': 'bar'})
```

Voir aussi le raccourci [render\(\)](#) qui appelle [render_to_string\(\)](#) et fournit le résultat à un objet [HttpResponse](#) prêt à être renvoyé depuis une vue.

Finalement, vous pouvez utiliser directement des moteurs configurés :

engines

Les moteurs de gabarit sont disponibles dans `django.template.engines`:

```
from django.template import engines

django_engine = engines['django']
template = django_engine.from_string("Hello {{ name }}!")
```

La clé de recherche, `'django'` dans cet exemple, correspond au réglage [NAME](#) du moteur.

Moteurs intégrés

`class DjangoTemplates`[\[source\]](#)

Définissez [BACKEND](#) à `'django.template.backends.django.DjangoTemplates'` pour configurer un moteur de gabarit de Django.

Lorsque [APP_DIRS](#) vaut `True`, les moteurs `DjangoTemplates` cherchent les gabarits dans le sous-répertoire `templates` des applications installées. Ce nom générique a été conservé par rétrocompatibilité.

Les moteurs DjangoTemplates acceptent les [OPTIONS](#) suivantes :

- `'allowed_include_roots'` : une liste de chaînes représentant les préfixes autorisés pour la balise de gabarit `{% ssi %}`. C'est une mesure de sécurité pour que les auteurs de gabarits ne puissent pas accéder à des fichiers auxquels ils ne devraient pas avoir accès.

Par exemple, si `'allowed_include_roots'` vaut `['/home/html', '/var/www']`, `{% ssi /home/html/foo.txt %}` fonctionne, mais pas `{% ssi /etc/passwd %}`.

La valeur par défaut est une liste vide.

Obsolète depuis la version 1.8: `allowed_include_roots` est obsolète car la balise `{% ssi %}` l'est également.

- `'context_processors'` : une liste de chemins Python pointés vers des objets exécutables utilisés pour remplir le contexte lorsqu'un gabarit est produit avec une requête. Ces exécutables acceptent un objet requête comme paramètre et renvoient un dictionnaire d'éléments à fusionner dans le contexte.

La valeur par défaut est une liste vide.

Voir [RequestContext](#) pour plus d'informations.

- `'debug'` : une valeur booléenne qui active ou désactive le mode débogage des gabarits. Quand elle vaut `True`, une page d'erreur élaborée affiche un rapport détaillé lors de toute exception générée durant le rendu des gabarits. Ce rapport contient les extraits concernés du gabarit avec les bonnes lignes mises en évidence.

La valeur par défaut correspond à la valeur du réglage [DEBUG](#).

- `'loaders'` : une liste de chemins Python pointés vers des classes de chargeurs de gabarits. Chaque classe `Loader` sait comment importer les gabarits d'une source particulière. Il est possible d'indiquer des tuples au lieu de chaînes. Le premier élément du tuple correspond au nom de classe de `Loader` alors que les éléments suivants seront transmis à la classe `Loader` en vue de son initialisation.

La valeur par défaut dépend des valeurs de [DIRS](#) et de [APP_DIRS](#).

Voir [Types de chargeurs](#) pour les détails.

- `'string_if_invalid'` : résultat, sous forme de chaîne de caractères, que le système des gabarits utilise pour remplacer le contenu de variables non valides (par ex. mal orthographiées).

La valeur par défaut est une chaîne vide.

Voir [Traitement des variables non valides](#) pour les détails.

- `'file_charset'` : le jeu de caractères utilisé pour lire les fichiers de gabarits depuis le disque.

La valeur par défaut est équivalente au réglage [FILE_CHARSET](#).

- `'libraries'`: un dictionnaire d'étiquettes et de chemins Python pointés de modules de balises de gabarit à inscrire auprès du moteur de gabarit. Ceci peut être utilisé pour ajouter de nouvelles bibliothèques ou pour fournir des étiquettes alternatives à celles qui existent. Par exemple :

```
OPTIONS={
    'libraries': {
        'myapp_tags': 'path.to.myapp.tags',
        'admin.urls': 'django.contrib.admin.templatetags.admin_urls',
    },
}
```

Les bibliothèques peuvent être chargées en passant la clé de dictionnaire correspondante à la balise [{% load %}](#).

- `'builtins'`: une liste de chemins Python pointés de modules de balises de gabarit à ajouter aux [modules intégrés](#). Par exemple :

```
OPTIONS={
    'builtins': ['myapp.builtins'],
}
```

Les balises et les filtres des bibliothèques intégrées peuvent être utilisés sans devoir d'abord faire appel à la balise [{% load %}](#).

New in Django 1.9:

Les paramètres `libraries` et `builtins` ont été ajoutés.

`class Jinja2`[\[source\]](#)

Nécessite que [Jinja2](#) soit installé :

```
$ pip install Jinja2
```

Définissez [BACKEND](#) à `'django.template.backends.jinja2.Jinja2'` pour configurer un moteur de gabarit [Jinja2](#).

Lorsque [APP_DIRS](#) vaut `True`, les moteurs `Jinja2` cherchent les gabarits dans le sous-répertoire `jinja2` des applications installées.

L'élément le plus important dans [OPTIONS](#) est `'environment'`. Il s'agit d'un chemin Python pointé vers un objet exécutable renvoyant un environnement `Jinja2`. La valeur par défaut est `'jinja2.Environment'`. Django appelle cet objet et transmet d'autres options en tant que paramètres nommés. De plus, ajoute des valeurs par défaut qui diffèrent de celles de `Jinja2` pour quelques-unes :

- `'autoescape': True`

- 'loader': un chargeur configuré pour [DIRS](#) et [APP_DIRS](#)
- 'auto_reload': settings.DEBUG
- 'undefined': DebugUndefined si settings.DEBUG sinon Undefined

The default configuration is purposefully kept to a minimum. If a template is rendered with a request (e.g. when using [render\(\)](#)), the Jinja2 backend adds the globals request, csrf_input, and csrf_token to the context. Apart from that, this backend doesn't create a Django-flavored environment. It doesn't know about Django context processors, filters, and tags. In order to use Django-specific APIs, you must configure them into the environment.

Par exemple, vous pouvez créer monproject/jinja2.py avec ce contenu :

```
from __future__ import absolute_import # Python 2 only

from django.contrib.staticfiles.storage import staticfiles_storage
from django.core.urlresolvers import reverse

from jinja2 import Environment

def environment(**options):
    env = Environment(**options)
    env.globals.update({
        'static': staticfiles_storage.url,
        'url': reverse,
    })
    return env
```

et définir l'option 'environment' à 'monproject.jinja2.environment'.

Dès lors, vous pouvez utiliser les structures suivantes dans les gabarits Jinja2 :

```

<a href="{{ url('admin:index') }}">Administration</a>
```

Les concepts de balises et filtres existent aussi bien dans le langage de gabarit de Django que dans Jinja2, mais ils sont utilisés différemment. Comme Jinja2 prend en charge le passage de paramètres à des objets exécutables dans les gabarits, beaucoup de fonctionnalités qui nécessitent une balise de gabarit ou un filtre dans les gabarits Django peuvent être implémentées simplement en appelant une fonction dans les gabarits Jinja2, comme l'exemple ci-dessus le montre. L'espace de noms global de Jinja2 élimine le besoin de processeurs de contexte de gabarit. Le langage de gabarit de Django ne possède pas d'équivalent aux tests Jinja2.

Moteurs personnalisés

Voici comment implémenter un moteur de gabarit personnalisé afin d'utiliser un autre système de gabarits. Un moteur de gabarit est une classe qui hérite de

`django.template.backends.base.BaseEngine`. Elle doit implémenter `get_template()` et, facultativement, `from_string()`. Voici un exemple d'une bibliothèque de gabarit fictive `foobar`:

```
from django.template import TemplateDoesNotExist, TemplateSyntaxError
from django.template.backends.base import BaseEngine
from django.template.backends.utils import csrf_input_lazy, csrf_token_lazy

import foobar
```

```
class FooBar(BaseEngine):

    # Name of the subdirectory containing the templates for this engine
    # inside an installed application.
    app_dirname = 'foobar'

    def __init__(self, params):
        params = params.copy()
        options = params.pop('OPTIONS').copy()
        super(FooBar, self).__init__(params)

        self.engine = foobar.Engine(**options)

    def from_string(self, template_code):
        try:
            return Template(self.engine.from_string(template_code))
        except foobar.TemplateCompilationFailed as exc:
            raise TemplateSyntaxError(exc.args)

    def get_template(self, template_name):
        try:
            return Template(self.engine.get_template(template_name))
        except foobar.TemplateNotFound as exc:
            raise TemplateDoesNotExist(exc.args, backend=self)
        except foobar.TemplateCompilationFailed as exc:
            raise TemplateSyntaxError(exc.args)
```

```
class Template(object):

    def __init__(self, template):
        self.template = template

    def render(self, context=None, request=None):
        if context is None:
            context = {}
        if request is not None:
            context['request'] = request
            context['csrf_input'] = csrf_input_lazy(request)
            context['csrf_token'] = csrf_token_lazy(request)
        return self.template.render(context)
```

Voir [DEP 182](#) pour plus d'informations.

Intégration du débogage pour les moteurs personnalisés

New in Django 1.9:

L'intégration de la page de débogage pour les moteurs de gabarit non Django a été ajoutée.

La page de débogage de Django présente des points d'entrée pour fournir des informations détaillées lorsqu'une erreur de gabarit se produit. Les moteurs de gabarit personnalisés peuvent utiliser ces points d'entrée pour améliorer les informations d'erreur qui sont présentées aux utilisateurs. Les points d'entrée suivants sont disponibles :

Gabarit postmortem

Le gabarit postmortem apparaît lorsque [TemplateDoesNotExist](#) est générée. Il présente la liste des moteurs et chargeurs de gabarit utilisés lors de la recherche du gabarit concerné. Par exemple, si deux moteurs Django sont configurés, le gabarit postmortem ressemble à ceci :

Template-loader postmortem

Django tried loading these templates, in this order:

Using engine django:

- `django.template.loaders.filesystem.Loader: /path/to/templates/xnotexists.html (Source does not exist)`
- `django.template.loaders.filesystem.Loader: /path/to/templates2/xnotexists.html (Source does not exist)`
- `django.template.loaders.app_directories.Loader: /path/to/app1/templates/xnotexists.html (Source does not exist)`

Using engine django2:

- `django.template.loaders.filesystem.Loader: /path/to/django2/xnotexists.html (Source does not exist)`

Les moteurs personnalisés peuvent remplir le gabarit postmortem en passant les paramètres backend et `tried` lors de la génération de [TemplateDoesNotExist](#). Les moteurs qui utilisent le gabarit postmortem [doivent indiquer une origine](#) sur l'objet de gabarit.

Information de ligne contextuelle

Si une erreur se produit pendant l'analyse et le rendu d'un gabarit, Django peut afficher la ligne à laquelle s'est produite l'erreur. Par exemple :

Error during template rendering

In template `/path/to/template.html`, error at line 4

Invalid block tag: 'syntax'

	1	some
	2	lines
	3	before
	4	Hello {% syntax error %} {{ world }}
	5	some
	6	lines
	7	after
	8	

Les moteurs personnalisés peuvent fournir cette information en définissant un attribut `template_debug` sur les exceptions générées pendant l'analyse et le rendu. Cet attribut est un [dict](#) possédant les valeurs suivantes :

- `'name'` : le nom du gabarit dans lequel l'exception s'est produite.
- `'message'` : le message de l'exception.
- `'source_lines'` : les lignes précédentes, suivantes ainsi que la ligne elle-même où s'est produite l'exception. C'est pour fournir du contexte, il ne faut donc pas inclure plus d'une vingtaine de lignes.
- `'line'` : le numéro de ligne à laquelle s'est produite l'exception.
- `'before'` : le contenu de la ligne ayant provoqué l'erreur, avant le symbole qui a produit l'erreur.
- `'during'` : le symbole qui a généré l'erreur.
- `'after'` : le contenu de la ligne ayant provoqué l'erreur, après le symbole qui a produit l'erreur.
- `'total'` : le nombre de lignes dans `source_lines`.
- `'top'` : le numéro de ligne où `source_lines` commence.
- `'bottom'` : le numéro de ligne où `source_lines` se termine.

Étant donné l'erreur de gabarit ci-dessus, `template_debug` ressemblerait à ceci :

```
{
  'name': '/path/to/template.html',
  'message': "Invalid block tag: 'syntax'",
  'source_lines': [
    (1, 'some\n'),
    (2, 'lines\n'),
    (3, 'before\n'),
    (4, 'Hello {% syntax error %} {{ world }}\n'),
    (5, 'some\n'),
    (6, 'lines\n'),
    (7, 'after\n'),
    (8, ''),
  ],
  'line': 4,
  'before': 'Hello ',
  'during': '{% syntax error %}',
  'after': ' {{ world }}\n',
  'total': 9,
  'bottom': 9,
  'top': 1,
}
```

API d'origine et intégration tierce

Les gabarits Django possèdent un objet [Origin](#) accessible par leur attribut `template.origin`. Ceci permet aux informations de débogage d'apparaître dans le [gabarit postmortem](#), de même que dans des bibliothèques tierces, telle que [Django Debug Toolbar](#).

Les moteurs personnalisés peuvent fournir leurs propres informations `template.origin` en créant un objet qui définit les attributs suivants :

- `'name'` : le chemin complet vers le gabarit.
- `'template_name'` : le chemin relatif vers le gabarit tel que transmis aux méthodes de chargement de gabarits.
- `'loader_name'` : une chaîne facultative identifiant la fonction ou la classe utilisée pour charger le gabarit, par exemple `django.template.loaders.filesystem.Loader`.

Le langage de gabarit de Django

Syntaxe

À propos de cette section

Il s'agit ici d'un aperçu de la syntaxe du langage de gabarit de Django. Pour plus de détails, voir la [référence de la syntaxe du langage](#).

Un gabarit Django est un simple document texte ou un chaîne Python, balisés à l'aide du langage de gabarit de Django. Certaines structures sont reconnues et interprétées par le moteur de gabarit. Les principales sont les variables et les balises.

Un gabarit est produit avec un contexte. Le processus de production remplace les variables par leurs valeurs qui sont cherchées dans le contexte, et il exécute les balises. Tout le reste est affiché tel quel.

La syntaxe du langage de gabarit de Django implique quatre structures.

Variables

Une variable affiche une valeur à partir du contexte, qui est un objet de type dictionnaire faisant correspondre des clés à des valeurs.

Les variables sont entourées par `{{ et }}` comme ceci :

My first name is `{{ first_name }}`. My last name is `{{ last_name }}`.

Avec un contexte `{'first_name': 'John', 'last_name': 'Doe'}`, ce gabarit produit :

My first name is John. My last name is Doe.

La consultation de dictionnaire, d'attribut et d'indice de liste est implémentée par une notation pointée :

`{{ my_dict.key }}`

```
{{ my_object.attribute }}  
{{ my_list.0 }}
```

Si le contenu d'une variable s'avère être un objet exécutable, le système de gabarit l'appelle sans paramètre et utilise son résultat à la place de l'objet exécutable.

Balises

Les balises permettent d'appliquer une logique arbitraire dans le processus de rendu.

Cette définition est volontairement vague. Par exemple, une balise peut produire du contenu, servir de structure de contrôle telle qu'une instruction « if » ou une boucle « for », extraire du contenu d'une base de données ou même de donner accès à d'autres balises de gabarit.

Les balises sont entourées par {% et %}, comme ceci :

```
{% csrf_token %}
```

La plupart des balises acceptent des paramètres :

```
{% cycle 'odd' 'even' %}
```

Certaines balises exigent des balises d'introduction et de terminaison :

```
{% if user.is_authenticated %}Hello, {{ user.username }}.{% endif %}
```

Une [référence des balises intégrés](#) est disponible tout comme des [instructions pour écrire des balises personnalisées](#).

Filtres

Les filtres transforment les valeurs de variables et les paramètres de balises.

Ils ressemblent à ceci :

```
{{ django|title }}
```

Avec un contexte {'django': 'the web framework for perfectionists with deadlines'}, ce gabarit produit le résultat suivant :

```
The Web Framework For Perfectionists With Deadlines
```

Certains filtres acceptent un paramètre :

```
{{ my_date|date:"Y-m-d" }}
```

Une [référence des filtres intégrés](#) est disponible tout comme des [instructions pour écrire des filtres personnalisés](#).

Commentaires

Les commentaires ressemblent à ceci :

```
{# this won't be rendered #}
```

Une balise `{% comment %}` autorise des commentaires sur plusieurs lignes.

Composants

À propos de cette section

Il s'agit ici d'un aperçu des API du langage de gabarit de Django. Pour plus de détails, voir la [référence des API](#).

Moteur

[django.template.Engine](#) encapsule une instance du système de gabarit de Django. La raison principale de créer directement une telle instance est d'utiliser le langage de gabarit de Django en dehors d'un projet Django.

[django.template.backends.django.DjangoTemplates](#) est un adaptateur léger autour de [django.template.Engine](#) pour l'adapter à l'API de moteur de gabarit de Django.

Gabarit

[django.template.Template](#) représente un gabarit compilé. Les gabarits sont obtenus par [Engine.get_template\(\)](#) ou [Engine.from_string\(\)](#)

De même, [django.template.backends.django.Template](#) est un adaptateur léger autour de [django.template.Template](#) pour l'adapter à l'API de gabarit commune.

Contexte

[django.template.Context](#) contient des métadonnées en plus des données de contexte. Il est transmis à [Template.render\(\)](#) en vue de la production d'un gabarit.

[django.template.RequestContext](#) est une sous-classe de [Context](#) qui stocke la requête [HttpRequest](#) en cours et exécute les processeurs de contexte de gabarit.

L'API commune ne possède pas de concept équivalent. Les données de contexte sont transmises dans un simple [dict](#) et la requête [HttpRequest](#) en cours est transmise séparément si nécessaire.

Chargeurs

Les chargeurs de gabarits sont responsables de la découverte des gabarits, de leur chargement et du renvoi d'objets [Template](#).

Django fournit plusieurs [chargeurs de gabarits intégrés](#) et prend en charge des [chargeurs de gabarits personnalisés](#).

Processeurs de contexte

Les processeurs de contexte sont des fonctions qui reçoivent la requête [HttpRequest](#) en cours comme paramètre et renvoient un [dict](#) de données à ajouter au contexte de production.

Leur utilisation principale est d'ajouter dans le contexte des données fréquemment utilisées partagées par tous les gabarits, sans devoir répéter le code correspondant dans chaque vue.

Django fournit un bon nombre de [processeurs de contexte intégrés](#). L'implémentation d'un processeur de contexte personnalisé est aussi simple que de définir une fonction.

[Fichiers annexes de formulaire \(classe `Media`\)](#)

[Vues fondées sur les classes](#)

Vues fondées sur les classes

Une vue est un exécutable acceptant une requête et renvoyant une réponse. Ce n'est pas forcément une simple fonction et Django fournit des exemples de classes qui peuvent être utilisées comme des vues. Celles-ci vous permettent de structurer vos vues et de réutiliser du code en exploitant l'héritage et les « mixins ». Il existe également des vues génériques pour des tâches simples (nous y reviendrons plus tard), mais vous pouvez tout à fait concevoir votre propre structure de vues réutilisables en fonction de votre cas d'utilisation. Pour des détails plus complets, consultez la [documentation de référence des vues fondées sur les classes](#).

- [Introduction aux vues fondées sur les classes](#)
- [Vues génériques fondées sur les classes, fournies par Django](#)
- [Gestion de formulaires avec les vues fondées sur les classes](#)
- [Utilisation de mixins avec les vues fondées sur les classes](#)

Exemples de base

Django fournit des classes de vues basiques qui conviennent à une large palette d'applications. Toutes ces vues héritent de la classe [View](#), qui gère la liaison de la vue aux URL, la distribution en fonction de la méthode HTTP et d'autres fonctionnalités de base. [RedirectView](#) est utile pour de simples redirections HTTP alors que [TemplateView](#) étend la classe de base afin d'être capable d'afficher un gabarit.

Utilisation simple dans une configuration d'URL

La manière la plus simple d'utiliser des vues génériques est de les créer directement dans votre configuration d'URL. Si vous ne devez changer qu'un nombre restreint d'attributs simples d'une vue fondée sur une classe, il suffit de les transmettre directement dans l'appel de méthode [as_view\(\)](#):

```
from django.conf.urls import url
from django.views.generic import TemplateView

urlpatterns = [
    url(r'^about/$', TemplateView.as_view(template_name="about.html")),
]
```

Tout paramètre transmis à [as_view\(\)](#) surcharge l'attribut de même nom de la classe. Dans cet exemple, nous définissons `template_name` de la vue `TemplateView`. Le même système de surcharge peut être utilisé pour l'attribut `url` de [RedirectView](#).

Héritage des vues génériques

L'autre façon, plus puissante, d'utiliser les vues génériques est d'hériter d'une vue existante et de surcharger ses attributs (comme `template_name`) ou ses méthodes (comme `get_context_data`) dans votre sous-classe pour fournir d'autres valeurs ou méthodes. Considérez par exemple une vue qui ne fait qu'afficher un gabarit, `about.html`. Django possède une vue générique pour faire cela, [TemplateView](#), il nous suffit donc d'en hériter et de surcharger la variable du nom du gabarit :

```
# some_app/views.py
from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "about.html"
```

Puis, il suffit d'ajouter cette nouvelle vue dans notre configuration d'URL. Comme [TemplateView](#) est une classe, pas une fonction, il est nécessaire de faire correspondre l'URL avec la méthode de classe [as_view\(\)](#) qui constitue un point d'entrée de type fonction pour les vues fondées sur les classes :

```
# urls.py
from django.conf.urls import url
from some_app.views import AboutView

urlpatterns = [
    url(r'^about/$', AboutView.as_view()),
]
```

Pour plus d'informations sur la façon d'utiliser les vues génériques intégrées dans Django, consultez le prochain thème sur les [vues génériques fondées sur les classes](#).

Prise en charge de méthodes HTTP alternatives

Supposons que quelqu'un veuille accéder à notre bibliothèque de livres par HTTP en utilisant les vues comme API. Le client de cette API se connecte de temps à autre et télécharge les données des livres publiés depuis sa dernière visite. Mais si aucune publication n'a eu lieu dans l'intervalle, ce serait une perte de temps processeur et de bande passante que de récupérer les livres de la base de données, de produire une réponse complète et de l'envoyer au client. Il pourrait être préférable d'interroger l'API au sujet de la date de publication la plus récente.

Nous faisons correspondre l'URL à la vue de la liste des livres dans la configuration d'URL :

```
from django.conf.urls import url
from books.views import BookListView

urlpatterns = [
    url(r'^books/$', BookListView.as_view()),
]
```

Et la vue :

```
from django.http import HttpResponse
from django.views.generic import ListView
from books.models import Book

class BookListView(ListView):
    model = Book

    def head(self, *args, **kwargs):
        last_book = self.get_queryset().latest('publication_date')
        response = HttpResponse('')
        # RFC 1123 date format
        response['Last-Modified'] = last_book.publication_date.strftime('%a, %d %b
%Y %H:%M:%S GMT')
        return response
```

Si on accède à la vue par une requête GET, une simple liste d'objets est renvoyée dans la réponse (en utilisant le gabarit `book_list.html`). Mais si le client lance une requête HEAD, le corps de la réponse sera vide et l'en-tête `Last-Modified` indiquera la date de publication de livre la plus récente. Sur la base de cette information, le client peut décider de charger ou non la liste complète des objets.

[Gabarits](#)

[Introduction aux vues fondées sur les classes](#)

Introduction aux vues fondées sur les classes

Les vues fondées sur les classes sont une autre manière d'implémenter les vues par des objets Python au lieu de fonctions. Elles ne remplacent pas les vues fondées sur les fonction, mais présentent certaines différences et avantages comparées aux vues fondées sur les fonctions :

- L'organisation du code en lien avec les méthodes HTTP spécifiques (GET, POST, etc.) peut se faire par des méthodes séparées au lieu d'utiliser un branchement conditionnel.
- Des techniques orientées objet comme les « mixins » (héritage multiple) peuvent être utilisées pour factoriser le code en composants réutilisables.

Liaisons et historique des vues génériques, des vues fondées sur les classes et de celles fondées sur les fonctions

Au départ, il n'y avait que le contrat de la vue fonction à laquelle Django transmet une requête [HttpRequest](#) et de laquelle il s'attend à recevoir une réponse [HttpResponse](#). C'était ce que Django lui-même pratiquait.

Assez rapidement, on a reconnu des idiomes et des modèles courants dans le développement des vues. Les vues génériques fondées sur les fonctions ont été introduites pour abstraire ces pratiques et faciliter le développement de vues pour ces cas courants.

Le problème avec les vues génériques fondées sur les fonctions, c'est qu'elles recouvrent bien les cas simples, mais qu'il n'est pas possible de les étendre ou de les personnaliser au-delà de simples options de configurations, limitant ainsi leur utilité dans bien des applications du monde réel.

Les vues génériques fondées sur les classes ont été créées avec les mêmes objectifs que celles fondées sur les fonctions, pour faciliter le développement de vues. Cependant, la manière d'implémenter cette solution par l'utilisation de mixins fournit un ensemble d'outils aboutissant à des vues génériques fondées sur les classes bien plus souples et extensibles que leur contrepartie fondées sur les fonctions.

Si vous avez essayé d'utiliser les vues génériques fondées sur les fonctions par le passé et que vous les avez trouvées trop limitées, vous ne devriez pas simplement considérer les vues génériques fondées sur les classes comme des équivalents, mais plutôt comme une nouvelle approche pour résoudre les problèmes d'origine que les vues génériques étaient censées résoudre.

L'ensemble des classes de base et des mixins que Django utilise pour construire les vues génériques fondées sur les classes sont conçues pour une souplesse maximale, et présentent ainsi de nombreux points d'accrochage sous les formes d'implémentation de méthodes par défaut et d'attributs qui ne vous concerneront probablement pas dans les cas d'utilisation les plus simples. Par exemple, au lieu de se limiter à un attribut de classe pour `form_class`, l'implémentation utilise une méthode `get_form` appelant à son tour la méthode `get_form_class` qui dans son implémentation par défaut ne fait que renvoyer l'attribut `form_class` de la classe. Cela vous donne plusieurs options pour indiquer le

formulaire à utiliser, que ce soit par un simple attribut ou par l'appel dynamique à une méthode que l'on peut surcharger. Ces options paraissent ajouter une complexité un peu absurde dans des situations simples, mais sans elles, les scénarios plus compliqués seraient limités.

Utilisation des vues fondées sur les classes

À la base, une vue fondée sur les classes permet de répondre à différentes méthodes de requête HTTP par différentes méthodes d'une classe plutôt que de faire appel à des branchements conditionnels dans une unique vue de type fonction.

Alors que le code pour gérer la méthode HTTP GET dans une fonction de vue ressemble à quelque chose comme cela :

```
from django.http import HttpResponse

def my_view(request):
    if request.method == 'GET':
        # <view logic>
        return HttpResponse('result')
```

Dans une vue fondée sur les classes, cela devient :

```
from django.http import HttpResponse
from django.views.generic import View

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

Comme le résolveur d'URL de Django est conçu pour envoyer la requête et ses paramètres associés à une fonction exécutable et non pas à une classe, les vues fondées sur les classes possèdent une méthode de classe [`as_view\(\)`](#) servant de point d'entrée exécutable vers la classe. Le point d'entrée `as_view` crée une instance de la classe et appelle sa méthode [`dispatch\(\)`](#). `dispatch` examine la requête pour déterminer s'il s'agit d'un GET, d'un POST, etc. et relaie la requête à une méthode correspondante si elle existe, ou génère l'erreur [`HttpResponseNotAllowed`](#) dans le cas contraire :

```
# urls.py
from django.conf.urls import url
from myapp.views import MyView

urlpatterns = [
    url(r'^about/$', MyView.as_view()),
]
```

Il vaut la peine de relever que ce qui est renvoyé par la méthode est identique à ce qui est renvoyé par une vue de type fonction, en l'occurrence une instance de [`HttpResponse`](#). Cela signifie que les [raccourcis `http`](#) ou les objets [`TemplateResponse`](#) peuvent tout à fait être utilisés dans une vue fondée sur les classes.

Même si une vue fondée sur les classes minimale ne nécessite aucun attribut de classe pour faire son travail, les attributs de classe sont utiles dans beaucoup d'architectures basées sur les classes et il existe deux façons de configurer ou de définir des attributs de classe.

La première est la manière standard de Python d'hériter ou de surcharger des attributs et des méthodes dans les sous-classes. Ainsi si la classe parente avait un attribut `greeting` comme ceci :

```
from django.http import HttpResponse
from django.views.generic import View

class GreetingView(View):
    greeting = "Good Day"

    def get(self, request):
        return HttpResponse(self.greeting)
```

Vous pouvez surcharger cela dans une sous-classe :

```
class MorningGreetingView(GreetingView):
    greeting = "Morning to ya"
```

Une autre option est de configurer des attributs de classe comme paramètres nommés dans l'appel à [`as_view\(\)`](#) dans la configuration d'URL :

```
urlpatterns = [
    url(r'^about/$', GreetingView.as_view(greeting="G'day")),
]
```

Note

Alors qu'une instance de classe est créée pour chaque requête qu'elle doit traiter, les attributs de classe définis par le point d'entrée [`as_view\(\)`](#) ne sont configurés qu'une seule fois au moment de l'importation de la configuration d'URL.

Utilisation de mixins

Les mixins sont une forme d'héritage multiple où les comportements et les attributs de plusieurs classes parentes peuvent être combinés.

Par exemple, dans les vues génériques fondées sur les classes, il existe un mixin nommé [`TemplateResponseMixin`](#) dont le rôle principal est de définir la méthode [`render_to_response\(\)`](#). Lorsqu'il est combiné avec le comportement de la classe de base [`View`](#), le résultat est une classe [`TemplateView`](#) qui distribue les requêtes vers les méthodes correspondantes (un comportement défini dans la classe de base `View`), et qui possède une méthode [`render_to_response\(\)`](#) utilisant un attribut [`template_name`](#) pour renvoyer un objet [`TemplateResponse`](#) (comportement défini dans `TemplateResponseMixin`).

Les mixins sont une très bonne façon de réutiliser du code entre plusieurs classes, mais il y a aussi un revers de médaille. Plus votre code est réparti entre des mixins, plus il sera difficile de lire une classe

enfant et de comprendre exactement ce qu'elle fait, et plus il sera difficile de savoir quelle méthode de quel mixin surcharger si vous héritez d'une structure avec une profonde arborescence d'héritage.

Notez également que vous ne pouvez hériter que d'une vue générique, c'est-à-dire qu'une seule classe parente peut hériter de [View](#) et toutes les autres classes (le cas échéant) doivent être des mixins. Si vous essayez d'hériter de plus d'une classe héritant de `View`, par exemple pour utiliser un formulaire au sommet d'une liste et combiner [ProcessFormView](#) avec [ListView](#), cela ne marchera pas comme espéré.

Gestion de formulaires avec les vues fondées sur les classes

Une vue de type fonction qui gère des formulaires peut ressembler à ceci :

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

from .forms import MyForm

def myview(request):
    if request.method == "POST":
        form = MyForm(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect('/success/')
    else:
        form = MyForm(initial={'key': 'value'})

    return render(request, 'form_template.html', {'form': form})
```

Une vue équivalente fondée sur les classes pourrait ressembler à ceci :

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.views.generic import View

from .forms import MyForm

class MyFormView(View):
    form_class = MyForm
    initial = {'key': 'value'}
    template_name = 'form_template.html'

    def get(self, request, *args, **kwargs):
        form = self.form_class(initial=self.initial)
        return render(request, self.template_name, {'form': form})

    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect('/success/')

        return render(request, self.template_name, {'form': form})
```

Il s'agit d'un cas très simple, mais vous pouvez voir que vous aurez ensuite l'option de personnaliser cette vue en surchargeant l'un des ses attributs de classe, par ex. `form_class`, à travers la configuration d'URL ou par l'héritage et la surcharge de l'une ou plusieurs de ses méthodes (les deux pouvant être combinés).

Décoration des vues fondées sur les classes

L'extension des vues fondées sur les classes n'est pas limitée aux mixins. Il est aussi possible d'utiliser des décorateurs. Comme les vues fondées sur les classes ne sont pas des fonctions, leur décoration fonctionne différemment selon que vous utilisez `as_view()` ou que vous créez une sous-classe.

Décoration dans la configuration d'URL

La façon la plus simple de décorer les vues fondées sur les classes est de décorer le résultat de la méthode `as_view()`. L'endroit le plus commode pour le faire est la configuration d'URL dans laquelle la vue est déployée :

```
from django.contrib.auth.decorators import login_required, permission_required
from django.views.generic import TemplateView

from .views import VoteView

urlpatterns = [
    url(r'^about/$',
        login_required(TemplateView.as_view(template_name="secret.html"))),
    url(r'^vote/$', permission_required('polls.can_vote')(VoteView.as_view())),
]
```

Cette approche applique le décorateur au niveau de l'instance. Si vous souhaitez que chaque instance d'une vue soit systématiquement décorée, vous devez suivre une autre approche.

Décoration de la classe

Pour décorer toutes les instances d'une vue fondée sur les classes, vous devez décorer la définition de classe elle-même. À cet effet, appliquez le décorateur à la méthode `dispatch()` de la classe.

Une méthode de classe n'est pas totalement identique à une fonction autonome, il n'est donc pas possible de simplement appliquer un décorateur de fonction à la méthode, il faut préalablement le transformer en un décorateur de méthode. Le décorateur `method_decorator` transforme un décorateur de fonction en un décorateur de méthode afin qu'il puisse être utilisé sur une méthode d'instance. Par exemple :

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django.views.generic import TemplateView

class ProtectedView(TemplateView):
    template_name = 'secret.html'

    @method_decorator(login_required)
```

```
def dispatch(self, *args, **kwargs):
    return super(ProtectedView, self).dispatch(*args, **kwargs)
```

Ou plus succinctement, vous pouvez plutôt décorer la classe et passer le nom de la méthode à décorer dans le paramètre nommé `name`:

```
@method_decorator(login_required, name='dispatch')
class ProtectedView(TemplateView):
    template_name = 'secret.html'
```

Si un ensemble de décorateurs sont utilisés à plusieurs endroits, vous pouvez définir une liste ou un tuple de décorateurs et l'utiliser au lieu d'appeler plusieurs fois `method_decorator()`. Ces deux classes sont équivalentes :

```
decorators = [never_cache, login_required]

@method_decorator(decorators, name='dispatch')
class ProtectedView(TemplateView):
    template_name = 'secret.html'

@method_decorator(never_cache, name='dispatch')
@method_decorator(login_required, name='dispatch')
class ProtectedView(TemplateView):
    template_name = 'secret.html'
```

Les décorateurs traitent une requête dans l'ordre où ils sont placés comme décorateurs. Dans l'exemple, `never_cache()` traitera la requête avant `login_required()`.

Changed in Django 1.9:

La possibilité d'utiliser `method_decorator()` sur une classe et la possibilité d'accepter une liste ou un tuple de décorateurs ont été ajoutées.

Dans cet exemple, chaque instance de `ProtectedView` possédera la protection de `login`.

Note

`method_decorator` transmet `*args` et `**kwargs` comme paramètres à la méthode décorée de la classe. Si votre méthode n'accepte pas un jeu de paramètres compatible, cela générera une exception `TypeError`.

[Vues fondées sur les classes](#)

[Vues génériques fondées sur les classes, fournies par Django](#)

Vues génériques fondées sur les classes, fournies par Django

L'écriture d'applications Web peut être une tâche monotone, car certains motifs se répètent encore et encore. Django essaie d'enlever une partie de cette monotonie au niveau des modèles et des gabarits, mais les développeurs Web font également face à ces répétitions au niveau des vues.

Les *vues génériques* de Django ont été développées pour résoudre ce problème. Elles s'attachent à trouver des motifs ou idiomes courants dans le développement des vues et à les abstraire de façon à ce qu'on puisse rapidement écrire des vues courantes sans devoir écrire trop de code.

Il est possible d'identifier certaines tâches courantes, comme l'affichage d'une liste d'objets, et d'écrire du code qui affiche une liste de *n'importe quel* objet. Puis, le modèle concerné peut être transmis comme paramètre supplémentaire dans la configuration d'URL.

Django est livré avec des vues génériques pour accomplir les choses suivantes :

- Afficher des pages de liste et de détail pour un seul type d'objet. Si nous devons créer une application pour gérer des conférences, une vue `TalkListView` et une vue `RegisteredUserListView` seraient des exemples de vues de listes. Une page d'une seule conférence pourrait être un exemple de ce que nous appelons une vue de « détail ».
- Présenter des objets datés dans des pages d'archive par année/mois/jour, avec les détails associés et des pages « éléments récents ».
- Permettre à des utilisateurs de créer, mettre à jour et supprimer des objets, avec ou sans gestion des autorisations.

Considérées globalement, ces vues offrent des interfaces simples pour effectuer les tâches les plus courantes que rencontrent les développeurs.

Extension des vues génériques

Il est indiscutable que l'utilisation des vues génériques peut considérablement accélérer le développement. Dans la plupart des projets, cependant, on atteint souvent un stade où les vues génériques ne suffisent plus. En fait, la question la plus fréquemment posée par les nouveaux développeurs Django est de savoir comment il est possible de gérer plus de variétés de situations avec les vues génériques.

C'est l'une des raisons pour lesquelles les vues génériques ont été revues pour la version 1.3 ; précédemment, ce n'étaient que des fonctions de vue avec une très large diversité d'options. Maintenant, au lieu de transmettre une grosse partie de configuration dans la configuration d'URL, la

manière recommandée d'étendre les vues génériques est d'en faire des sous-classes et de surcharger leurs attributs et méthodes.

Ceci dit, les vues génériques ont leurs limites. Si vous vous retrouvez à vous battre pour implémenter vos vues comme des sous-classes de vues génériques, il se pourrait qu'il soit plus efficace d'écrire simplement le code dont vous avez besoin dans vos vues, que ce soit par des fonctions ou des classes.

Vous trouverez davantage d'exemples de vues génériques dans certaines applications tierces, mais n'hésitez pas à écrire vos propres vues selon vos besoins.

Vues génériques d'objets

[TemplateView](#) est certainement utile, mais les vues génériques de Django exposent tout leur potentiel lorsqu'il s'agit de présenter des vues de contenu de la base de données. Dans la mesure où c'est une tâche très courante, Django contient un ensemble de vues génériques intégrées qui facilitent énormément les vues en liste et les vues de détail pour les objets.

Commençons par examiner quelques exemples d'affichage d'une liste d'objets ou d'un objet individuel.

Nous allons utiliser ces modèles :

```
# models.py
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    class Meta:
        ordering = ["-name"]

    def __str__(self):
        # __unicode__ on Python 2
        return self.name

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')

    def __str__(self):
        # __unicode__ on Python 2
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)
    publication_date = models.DateField()
```

Nous devons maintenant définir une vue :

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
```

Et finalement connecter cette vue à une URL :

```
# urls.py
from django.conf.urls import url
from books.views import PublisherList

urlpatterns = [
    url(r'^publishers/$', PublisherList.as_view()),
]
```

Voilà tout le code Python qu'il suffit d'écrire. Il faut cependant encore écrire un gabarit. Nous pourrions indiquer explicitement à la vue le gabarit à utiliser en lui ajoutant un attribut `template_name`, mais en l'absence d'un gabarit explicite, Django déduit un nom de gabarit à partir du nom de l'objet. Dans ce cas, le gabarit « automatique » sera `"books/publisher_list.html"`, la partie « books » provient du nom de l'application contenant ce modèle alors que « publisher » provient du nom du modèle en minuscules.

Note

Ainsi, lorsque (par exemple) l'option `APP_DIRS` d'un moteur `DjangoTemplates` est définie à `True` dans [TEMPLATES](#), un emplacement de gabarit pourrait être :
`/chemin/vers/projet/books/templates/books/publisher_list.html`.

Ce gabarit sera rendu avec un contexte contenant une variable appelée `object_list` contenant elle-même les objets `Publisher`. Un gabarit très simple pourrait ressembler à ceci :

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

{% block content %}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list %}
            <li>{{ publisher.name }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

Et voilà, tout est là. Toutes les fonctionnalités sympathiques des vues génériques proviennent des modifications d'attributs définis dans la vue générique. La [référence des vues génériques](#) documente en détails toutes les vues génériques et leurs options. La suite de ce document s'attelle à décrire certaines manières fréquentes de personnaliser et étendre les vues génériques.

Construction de contextes de gabarits « astucieux »

Vous avez peut-être constaté que notre exemple de gabarit de liste d'éditeurs stocke tous les éditeurs dans une variable nommée `object_list`. Même si cela fonctionne très bien, ce n'est pas très « sympa » pour les auteurs de gabarits : ils doivent simplement savoir qu'ils manipulent ici des éditeurs.

Si vous avez affaire à un objet de modèle, c'est déjà fait pour vous. Lorsque vous avez affaire à un objet ou un jeu de requête, Django est capable de renseigner le contexte en utilisant la version en minuscules du nom de la classe de modèle. Ceci s'ajoute à l'élément `object_list` par défaut, mais contient exactement les mêmes données, c'est-à-dire `publisher_list`.

Si ce n'est toujours pas un bon nom, vous pouvez indiquer explicitement le nom de la variable de contexte. L'attribut `context_object_name` d'une vue générique indique le nom de variable de contexte à utiliser :

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
    context_object_name = 'my_favorite_publishers'
```

La mise à disposition d'un nom `context_object_name` signifiant est toujours une bonne idée. Vos collaborateurs rédacteurs de gabarits vous remercieront.

Ajout de contexte supplémentaire

Il est souvent nécessaire de compléter par quelques informations supplémentaires au-delà de ce que la vue générique fournit. Par exemple, on peut penser à afficher une liste de livres sur la page de détail des éditeurs. La vue générique [DetailView](#) place l'éditeur dans le contexte, mais comment ajouter d'autres informations dans le gabarit ?

La réponse consiste à créer une sous-classe de [DetailView](#) et d'y placer votre propre implémentation de la méthode `get_context_data`. L'implémentation par défaut n'y ajoute que l'objet à afficher dans le gabarit, mais en la surchargeant, vous pouvez l'enrichir :

```
from django.views.generic import DetailView
from books.models import Publisher, Book

class PublisherDetail(DetailView):

    model = Publisher

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super(PublisherDetail, self).get_context_data(**kwargs)
        # Add in a QuerySet of all the books
        context['book_list'] = Book.objects.all()
        return context
```

Note

Généralement, `get_context_data` fusionne les données de contexte de toutes les classes parentes avec celles de la classe actuelle. Pour préserver ce comportement dans vos propres classes où vous souhaitez modifier le contexte, il faut prendre soin d'appeler `get_context_data` de la classe parente (`super`). Pour autant qu'il n'y ait pas deux classes essayant de modifier la même clé, le résultat sera correct. Cependant, si l'une des classes tente de surcharger une clé après qu'une classe parente l'a définie (après l'appel à `super`), toute classe enfant de cette classe devra aussi définir explicitement cette clé après `super` pour être certain de surcharger tous les parents. Si cela vous cause des ennuis, analysez l'ordre de résolution des méthodes de votre vue.

Un autre élément à signaler est que les données de contexte provenant des vues génériques basées sur les classes vont écraser les données fournies par les processeurs de contexte ; voir [`get_context_data\(\)`](#) comme exemple.

Affichage de sous-ensembles d'objets

Examinons maintenant un peu plus attentivement le paramètre `model` que nous avons toujours utilisé. Ce paramètre, qui définit le modèle de base de données sur lequel la vue va porter, est disponible pour toutes les vues génériques qui opèrent sur un seul objet ou une liste d'objets. Cependant, le paramètre `model` n'est pas le seul moyen de définir le type d'objet d'une vue, il est aussi possible de définir la liste des objets par le paramètre `queryset`:

```
from django.views.generic import DetailView
from books.models import Publisher

class PublisherDetail(DetailView):
    context_object_name = 'publisher'
    queryset = Publisher.objects.all()
```

L'indication de `model = Publisher` n'est en fait qu'un raccourci pour dire `queryset = Publisher.objects.all()`. Cependant, en utilisant `queryset` pour définir une liste d'objets filtrée, vous pouvez spécifier plus finement les objets qui seront visibles dans la vue (voir [Création de requêtes](#) pour plus d'informations à propos des objets [QuerySet](#), et consultez la [référence des vues fondées sur les classes](#) pour obtenir tous les détails).

Comme exemple simple, essayons de trier une liste de livres par date de publication, les plus récents en premier :

```
from django.views.generic import ListView
from books.models import Book

class BookList(ListView):
    queryset = Book.objects.order_by('-publication_date')
    context_object_name = 'book_list'
```

Il s'agit d'un exemple très minimal, mais qui montre bien le principe de fonctionnement. Naturellement, le but est généralement de faire plus que de simplement trier les objets. Si vous souhaitez présenter une liste de livres provenant d'un éditeur particulier, la même technique peut être utilisée :

```
from django.views.generic import ListView
from books.models import Book

class AcmeBookList(ListView):

    context_object_name = 'book_list'
    queryset = Book.objects.filter(publisher__name='ACME Publishing')
    template_name = 'books/acme_list.html'
```

Remarquez qu'en plus d'un paramètre `queryset` filtré, nous indiquons également un nom de gabarit personnalisé. Sans cela, la vue générique utiliserait le même gabarit que la liste d'objets « standard », ce qui ne correspondrait pas au but recherché.

Remarquez également que ce n'est pas une façon très élégante d'afficher les livres d'un éditeur particulier. Si nous voulions ensuite ajouter une page pour un autre éditeur, nous devrions ajouter des lignes supplémentaires dans la configuration d'URL et nous serions pratiquement limités à quelques éditeurs. Nous aborderons ce problème dans la section suivante.

Note

Si vous obtenez une erreur 404 en accédant à `/books/acme/`, vérifiez que vous possédez bien un objet `Publisher` dont le nom est « ACME Publishing ». Les vues génériques ont un paramètre `allow_empty` pour cette situation. Consultez la [référence des vues fondées sur les classes](#) pour plus de détails.

Filtrage dynamique

Un autre besoin fréquent est de filtrer les objets d'une liste selon un critère de l'URL. Plus haut, nous avons figé le nom de l'éditeur dans la configuration d'URL, mais comment faire si nous voulions écrire une vue affichant tous les livres d'un éditeur librement choisi ?

Heureusement, la classe `ListView` possède une méthode [get_queryset\(\)](#) que nous pouvons surcharger. Précédemment, elle renvoyait simplement la valeur du paramètre `queryset`, mais maintenant nous pouvons ajouter de la logique supplémentaire.

L'élément clé pour que cela fonctionne est qu'au moment où les vues fondées sur les classes sont appelées, divers attributs utiles sont créés pour `self`. En plus de la requête elle-même (`self.request`), il y a aussi les paramètres positionnels (`self.args`) et nommés (`self.kwargs`) capturés en fonction de la configuration d'URL.

Ici, nous avons une configuration d'URL comportant un seul groupe capturé :

```
# urls.py
from django.conf.urls import url
```

```
from books.views import PublisherBookList

urlpatterns = [
    url(r'^books/([\w-]+)/$', PublisherBookList.as_view()),
]
```

Puis, nous allons écrire la vue `PublisherBookList`:

```
# views.py
from django.shortcuts import get_object_or_404
from django.views.generic import ListView
from books.models import Book, Publisher

class PublisherBookList(ListView):

    template_name = 'books/books_by_publisher.html'

    def get_queryset(self):
        self.publisher = get_object_or_404(Publisher, name=self.args[0])
        return Book.objects.filter(publisher=self.publisher)
```

Comme vous pouvez le voir, il est relativement facile d'ajouter de la logique supplémentaire dans la sélection des objets de base de données. Nous pourrions par exemple filtrer en se basant sur l'utilisateur connecté (`self.request.user`) ou toute autre logique plus complexe.

Nous pouvons aussi ajouter en même temps l'éditeur dans le contexte, afin de pouvoir l'utiliser dans le gabarit :

```
# ...

def get_context_data(self, **kwargs):
    # Call the base implementation first to get a context
    context = super(PublisherBookList, self).get_context_data(**kwargs)
    # Add in the publisher
    context['publisher'] = self.publisher
    return context
```

Autres opérations supplémentaires

Le dernier usage que nous examinerons concerne les opérations supplémentaires effectuées avant ou après l'appel à la vue générique.

Imaginez que nous ayons un champ `last_accessed` dans le modèle `Author` qui nous sert à garder la trace du moment le plus récent où quelqu'un a consulté la page de cet auteur :

```
# models.py
from django.db import models

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')
    last_accessed = models.DateTimeField()
```

La classe générique `DetailView` n'a évidemment aucune conscience de la présence de ce champ, mais nous pouvons une nouvelle fois écrire facilement une vue personnalisée pour maintenir ce champ à jour.

Tout d'abord, nous devons ajouter une ligne de détail d'auteur dans la configuration d'URL pour faire le lien avec une vue personnalisée :

```
from django.conf.urls import url
from books.views import AuthorDetailView

urlpatterns = [
    #...
    url(r'^authors/(?P<pk>[0-9]+)/$', AuthorDetailView.as_view(), name='author-
detail'),
]
```

Puis il s'agit d'écrire la nouvelle vue. `get_object` étant la méthode qui récupère l'objet, nous la surchargeons en « enveloppant » l'appel par défaut :

```
from django.views.generic import DetailView
from django.utils import timezone
from books.models import Author

class AuthorDetailView(DetailView):
    queryset = Author.objects.all()

    def get_object(self):
        # Call the superclass
        object = super(AuthorDetailView, self).get_object()
        # Record the last accessed date
        object.last_accessed = timezone.now()
        object.save()
        # Return the object
        return object
```

Note

Cette configuration d'URL emploie le group nommé `pk`, il s'agit là du nom par défaut utilisé par `DetailView` pour trouver la valeur de clé primaire à utiliser pour filtrer le jeu de requête.

Si vous souhaitez nommer différemment le groupe, vous pouvez définir `pk_url_kwarg` dans la vue. Vous trouverez davantage de détails dans la référence de [DetailView](#).

[Introduction aux vues fondées sur les classes](#)

[Gestion de formulaires avec les vues fondées sur les classes](#)

Gestion de formulaires avec les vues fondées sur les classes

Le traitement de formulaires utilise généralement trois parcours :

- Affichage initial GET (vierge ou contenu pré-rempli)
- Envoi POST avec données non valides (réaffiche normalement le formulaire avec indication des erreurs)
- Envoi POST avec données valides (traitement des données normalement suivi par une redirection)

Lorsqu'on implémente soi-même ces étapes, il en résulte souvent beaucoup de code répétitif (voir [Utilisation d'un formulaire dans une vue](#)). Pour l'éviter, Django fournit un ensemble de vues génériques fondées sur les classes dédiées au traitement des vues.

Formulaires élémentaires

Étant donné un formulaire simple de contact :

forms.py

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # send email using the self.cleaned_data dictionary
        pass
```

La vue peut être construite en utilisant une classe `FormView`:

views.py

```
from myapp.forms import ContactForm
from django.views.generic.edit import FormView

class ContactView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = '/thanks/'

    def form_valid(self, form):
        # This method is called when valid form data has been POSTed.
        # It should return an HttpResponseRedirect.
        form.send_email()
        return super(ContactView, self).form_valid(form)
```

Notes :

- `FormView` hérite de [TemplateResponseMixin](#), ce qui fait qu'on peut utiliser [template_name](#) à cet endroit.
- L'implémentation par défaut de [form_valid\(\)](#) redirige simplement vers [success_url](#).

Formulaires de modèles

Les vues génériques brillent particulièrement lorsqu'elles interagissent avec des modèles. Ces vues génériques vont automatiquement créer un formulaire [ModelForm](#), pour autant qu'elles puissent déterminer la classe de modèle à utiliser :

- Si l'attribut [model](#) est présent, c'est cette classe de modèle qui sera utilisée.
- Si [get_object\(\)](#) renvoie un objet, c'est la classe de cet objet qui sera utilisée.
- Si un attribut [queryset](#) est présent, c'est le modèle correspondant à ce jeu de requête qui sera utilisé.

Les vues de formulaire de modèle contiennent une implémentation de [form_valid\(\)](#) qui enregistre automatiquement le modèle. Vous pouvez surcharger cela si vous avez des exigences particulières ; voir ci-dessous pour des exemples.

Vous n'avez même pas besoin d'indiquer un attribut `success_url` pour [CreateView](#) ou [UpdateView](#), elles vont utiliser [get_absolute_url\(\)](#) sur l'objet de modèle le cas échéant.

Si vous souhaitez utiliser un formulaire [ModelForm](#) personnalisé (par exemple pour ajouter de la validation en plus), il suffit de définir [form_class](#) dans la vue.

Note

Lorsqu'une classe de formulaire personnalisée est présente, il est toujours nécessaire de définir le modèle, même si la classe [form_class](#) est une classe [ModelForm](#).

Nous devons d'abord ajouter une méthode [get_absolute_url\(\)](#) à notre classe `Author`:

`models.py`

```
from django.core.urlresolvers import reverse
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)

    def get_absolute_url(self):
        return reverse('author-detail', kwargs={'pk': self.pk})
```

Puis nous pouvons utiliser [CreateView](#) et compagnie pour faire le travail. Remarquez que nous avons ici juste à configurer les vues génériques fondées sur les classes ; nous n'avons pas à écrire nous-même de logique de vue :

views.py

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.core.urlresolvers import reverse_lazy
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['name']

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('author-list')
```

Note

Nous devons utiliser [reverse_lazy\(\)](#) ici, pas simplement `reverse` car la configuration d'URL n'est pas chargée au moment où le fichier est importé.

L'attribut `fields` fonctionne de la même manière que l'attribut `fields` de la classe interne `Meta` de [ModelForm](#). Sauf dans les cas où la classe de formulaire est définie d'une autre manière, cet attribut est obligatoire et la vue génère une exception [ImproperlyConfigured](#) s'il est absent.

Si vous définissez à la fois les attributs [fields](#) et [form_class](#), une exception [ImproperlyConfigured](#) est générée.

Changed in Django 1.8:

L'omission de l'attribut `fields` était précédemment admise et aboutissait à un formulaire contenant tous les champs du modèle.

Changed in Django 1.8:

Précédemment, si `fields` et `form_class` étaient tous deux définis, `fields` était silencieusement ignoré.

Finalement, nous connectons ces nouvelles vues dans la configuration d'URL :

urls.py

```
from django.conf.urls import url
from myapp.views import AuthorCreate, AuthorUpdate, AuthorDelete

urlpatterns = [
    # ...
    url(r'author/add/$', AuthorCreate.as_view(), name='author-add'),
    url(r'author/(?P<pk>[0-9]+)/$', AuthorUpdate.as_view(), name='author-update'),
    url(r'author/(?P<pk>[0-9]+)/delete/$', AuthorDelete.as_view(), name='author-
delete'),
]
```

Note

Ces vues héritent de [SingleObjectTemplateResponseMixin](#) qui utilise [template_name_suffix](#) pour construire [template_name](#) en fonction du modèle.

Dans cet exemple :

- [CreateView](#) et [UpdateView](#) utilisent `myapp/author_form.html`
- [DeleteView](#) utilise `myapp/author_confirm_delete.html`

Si vous aimeriez avoir des gabarits séparés pour [CreateView](#) et [UpdateView](#), vous pouvez définir [template_name](#) ou [template_name_suffix](#) dans votre classe de vue.

Modèles et `request.user`

Pour garder trace de l'utilisateur ayant créé un objet en utilisant [CreateView](#), vous pouvez utiliser un formulaire [ModelForm](#) personnalisé. Premièrement, ajoutez la relation de clé étrangère dans le modèle :

`models.py`

```
from django.contrib.auth.models import User
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)
    created_by = models.ForeignKey(User, on_delete=models.CASCADE)

    # ...
```

Dans la vue, prenez soin de ne pas inclure `created_by` dans la liste des champs modifiables et surchargez [form_valid\(\)](#) afin d'ajouter l'utilisateur :

`views.py`

```
from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

    def form_valid(self, form):
        form.instance.created_by = self.request.user
        return super(AuthorCreate, self).form_valid(form)
```

Notez que vous devrez [décorer cette vue](#) par [login_required\(\)](#) ou trouver un autre moyen de gérer les accès d'utilisateurs non autorisés dans [form_valid\(\)](#).

Exemple AJAX

Voici un exemple simple montrant une possible implémentation d'un formulaire fonctionnant aussi bien pour des requêtes AJAX que pour des envois de formulaire POST « normaux » :

```
from django.http import JsonResponse
from django.views.generic.edit import CreateView
from myapp.models import Author

class AjaxableResponseMixin(object):
    """
    Mixin to add AJAX support to a form.
    Must be used with an object-based FormView (e.g. CreateView)
    """
    def form_invalid(self, form):
        response = super(AjaxableResponseMixin, self).form_invalid(form)
        if self.request.is_ajax():
            return JsonResponse(form.errors, status=400)
        else:
            return response

    def form_valid(self, form):
        # We make sure to call the parent's form_valid() method because
        # it might do some processing (in the case of CreateView, it will
        # call form.save() for example).
        response = super(AjaxableResponseMixin, self).form_valid(form)
        if self.request.is_ajax():
            data = {
                'pk': self.object.pk,
            }
            return JsonResponse(data)
        else:
            return response

class AuthorCreate(AjaxableResponseMixin, CreateView):
    model = Author
    fields = ['name']
```

[Vues génériques fondées sur les classes, fournies par Django](#)
[Utilisation de mixins avec les vues fondées sur les classes](#)

Utilisation de mixins avec les vues fondées sur les classes

Prudence

Ceci est un sujet avancé. Nous suggérons d'avoir une bonne connaissance préalable des [vues Django fondées sur les classes](#) avant d'explorer ces techniques.

Les vues fondées sur les classes intégrées dans Django fournissent de nombreuses fonctionnalités, mais certaines d'entre elles pourraient être utiles de manière indépendante. Par exemple, si l'on écrit une vue qui utilise un gabarit pour produire une réponse HTTP, mais sans utiliser [TemplateView](#); ou encore si l'on veut utiliser un gabarit seulement pour une requête POST et faire quelque chose de totalement différent pour les requêtes GET. Même si on pourrait utiliser directement [TemplateResponse](#), on se retrouverait avec pas mal de code dupliqué.

C'est pour cette raison que Django fournit aussi un certain nombre de mixins englobant des fonctionnalités de manière distincte. Le rendu de gabarits, par exemple, est isolé dans la classe [TemplateResponseMixin](#). La documentation de référence de Django contient une [documentation complète de toutes les classes mixins](#).

Contexte et réponses par gabarit

Deux classes mixins centrales sont à disposition pour aider à présenter une interface cohérente pour travailler avec les gabarits dans les vues fondées sur les classes.

[TemplateResponseMixin](#)

Chaque vue intégrée renvoyant une réponse [TemplateResponse](#) appelle la méthode [render_to_response\(\)](#) fournie par [TemplateResponseMixin](#). Dans la plupart des cas, cet appel se fait automatiquement (par exemple, elle est appelée par la méthode `get()` aussi bien de [TemplateView](#) que de [DetailView](#)). De même, il est peu probable que vous ayez besoin de la surcharger, bien que cela peut se révéler utile si vous voulez que la réponse renvoie du contenu non rendu par un gabarit Django. Vous pouvez trouver un exemple d'un tel usage dans [l'exemple JsonResponseMixin](#).

`render_to_response()` appelle elle-même [get_template_names\(\)](#) qui ne fait que consulter l'attribut `template_name` de la vue, par défaut. Deux autres mixins ([SingleObjectTemplateResponseMixin](#) et [MultipleObjectTemplateResponseMixin](#)) surchargent cette méthode pour offrir des valeurs par défaut plus souples lorsqu'il s'agit de manipuler des objets réels.

[ContextMixin](#)

Chaque vue intégrée nécessitant des données de contexte, comme pour effectuer le rendu d'un gabarit (y compris [TemplateResponseMixin](#) ci-dessus), doit appeler [get_context_data\(\)](#) en transmettant sous forme de paramètres nommés les données qu'elle souhaite y placer. `get_context_data()` renvoie un dictionnaire ; dans [ContextMixin](#), elle renvoie simplement ses paramètres nommés, mais il est fréquent de surcharger cela pour ajouter davantage de contenu dans ce dictionnaire.

Construction des vues fondées sur les classes de Django

Voyons comment deux des vues fondées sur les classes de Django sont construites à partir de mixins fournissant des fonctionnalités isolées. Nous examinerons [DetailView](#) qui produit une vue de détail d'un objet et [ListView](#) qui produit une liste d'objets, généralement à partir d'un jeu de requête, avec pagination facultative. Ceci nous amènera à étudier quatre mixins qui, combinées entre elles, fournissent des fonctionnalités utiles lors du traitement d'un objet Django unique ou d'un ensemble d'objets.

Il existe également des classes mixins dans les vues génériques d'édition ([FormView](#) et les vues spécifiques aux modèles [CreateView](#), [UpdateView](#) et [DeleteView](#)), ainsi que dans les vues génériques centrées sur les dates. Ces classes sont décrites dans la [documentation de référence des mixins](#).

DetailView: traitement d'un seul objet Django

Pour afficher le détail d'un objet, il y a fondamentalement deux choses à faire : récupérer l'objet, puis produire une réponse [TemplateResponse](#) à l'aide du gabarit adéquat, en transmettant l'objet dans le contexte.

Pour obtenir l'objet, [DetailView](#) se base sur [SingleObjectMixin](#) qui fournit une méthode [get_object\(\)](#) qui va rechercher l'objet en se basant sur l'URL de la requête (elle cherche les paramètres nommés pk et slug tels que déclarés dans la configuration d'URL et sélectionne l'objet en utilisant soit l'attribut [model](#) de la vue, soit l'attribut [queryset](#) si celui-ci est présent).

[SingleObjectMixin](#) surcharge aussi [get_context_data\(\)](#), qui est utilisée dans toutes les vues fondées sur les classes de Django pour fournir les données de contexte servant au rendu des gabarits.

Pour produire la réponse [TemplateResponse](#), [DetailView](#) utilise [SingleObjectTemplateResponseMixin](#), qui étend [TemplateResponseMixin](#), surchargeant [get_template_names\(\)](#) comme expliqué plus haut. Il fournit en réalité un bon nombre d'options élaborées, mais le nom que la plupart des gens vont utiliser est `<étiquette_application>/<nom_modèle>_detail.html`. La partie `_detail` peut être modifiée en définissant un attribut [template_name_suffix](#) différent dans une sous-classe (par exemple, les [vues génériques d'édition](#) emploient `_form` pour les vues de création et de mise à jour et `_confirm_delete` pour les vues de suppression).

ListView: traitement de plusieurs objets Django

Les listes d'objets suivent grosso modo le même modèle : récupérer d'abord une liste d'objets (potentiellement paginés), typiquement un objet [QuerySet](#), puis produire une réponse [TemplateResponse](#) avec le gabarit adéquat pour exploiter cette liste d'objets.

Pour obtenir les objets, [ListView](#) utilise [MultipleObjectMixin](#) qui offre à la fois [get_queryset\(\)](#) et [paginate_queryset\(\)](#). Au contraire de [SingleObjectMixin](#), il n'est pas nécessaire d'identifier des éléments d'URL pour déterminer le jeu de requête à traiter ; le comportement par défaut est de simplement utiliser l'attribut [queryset](#) ou [model](#) de la classe de vue. Une raison fréquente de vouloir surcharger ici [get_queryset\(\)](#) est d'adapter la liste d'objets de manière dynamique, par exemple en fonction de l'utilisateur actuel ou pour exclure des articles dont la date est dans le futur dans un blog.

[MultipleObjectMixin](#) surcharge aussi [get_context_data\(\)](#) pour inclure les variables de contexte appropriées à la pagination (indiquant des valeurs vides si la pagination est désactivée). Elle compte sur la présence de `object_list` dans les paramètres nommés transmis, ce dont se charge [ListView](#).

Pour produire une réponse [TemplateResponse](#), [ListView](#) utilise ensuite [MultipleObjectTemplateResponseMixin](#); comme pour [SingleObjectTemplateResponseMixin](#) ci-dessus, `get_template_names()` est surchargée pour fournir une [série d'options](#) dont la plus utilisée est le nom de gabarit `<étiquette_application>/<nom_modèle>_list.html` où la partie `_list` provient aussi de l'attribut [template_name_suffix](#) (les vues génériques centrées sur les dates utilisent des suffixes tels que `_archive`, `_archive_year`, etc. pour utiliser différents gabarits correspondant aux différentes vues de listes spécialisées pour les dates).

Utilisation des mixins de vues fondées sur les classes de Django

Maintenant que nous avons vu comment les classes génériques fondées sur les classes de Django utilisent les mixins à disposition, examinons d'autres façons de les combiner. Nous allons naturellement toujours les combiner avec d'autres vues intégrées ou génériques fondées sur les classes, mais les cas d'utilisation auxquels Django répond ne recouvrent pas toutes les situations plus rares auxquelles on peut être confronté.

Avertissement

On ne peut pas combiner tous les mixins, ni combiner n'importe quel mixin avec toutes les vues génériques fondées sur les classes. Nous présentons ici quelques exemples qui fonctionnent ; si vous souhaitez construire d'autres fonctionnalités, vous devrez étudier les interactions entre les attributs et les méthodes qui se chevauchent entre les différentes classes utilisées, ainsi que la façon dont [l'ordre de résolution des méthodes](#) affecte la version des méthodes appelée et l'ordre de ces appels.

La documentation de référence des [vues fondées sur les classes](#) et des [mixins de vues fondées sur les classes](#) de Django vous aidera à comprendre quels attributs et méthodes peuvent être la source de conflits entre les différentes classes et mixins.

Dans le doute, il est souvent préférable de simplifier les choses en se basant sur [View](#) ou [TemplateView](#), peut-être avec [SingleObjectMixin](#) ou [MultipleObjectMixin](#). Même si vous devez finalement écrire un peu plus de code, il sera probablement plus compréhensible pour quelqu'un devant s'y atteler plus tard, et en réduisant les interactions à surveiller, vous vous économiserez un peu de temps de réflexion (il est bien sûr toujours possible de se plonger dans l'implémentation de Django des vues génériques fondées sur les classes pour y chercher de l'inspiration sur la façon de traiter les problèmes).

Utilisation de `SingleObjectMixin` avec `View`

Si nous voulons écrire une classe de vue simple ne répondant qu'aux requêtes POST, nous héritons de [View](#) et écrivons une méthode `post()` dans la sous-classe. Cependant, si le traitement doit porter sur un objet particulier identifié par l'URL, il est souhaitable de profiter de la fonctionnalité offerte par [SingleObjectMixin](#).

Nous allons démontrer cela avec le modèle `Author` que nous avons utilisé dans [l'introduction aux vues génériques basées sur les classes](#).

`views.py`

```
from django.http import HttpResponseRedirect, HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.views.generic import View
from django.views.generic.detail import SingleObjectMixin
from books.models import Author

class RecordInterest(SingleObjectMixin, View):
    """Records the current user's interest in an author."""
    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseRedirect()

        # Look up the author we're interested in.
        self.object = self.get_object()
        # Actually record interest somehow here!

        return HttpResponseRedirect(reverse('author-detail', kwargs={'pk':
self.object.pk})))
```

En pratique, l'intérêt devrait probablement être enregistré dans un stockage clé-valeur plutôt que dans une base de données relationnelle, nous avons donc laissé cet aspect de côté. La seule partie de la vue qui peut profiter de l'utilisation de [SingleObjectMixin](#) est l'endroit où il s'agit de récupérer l'auteur concerné par l'intérêt, ce qui est fait en appelant simplement `self.get_object()`. Tout le reste est pris en charge pour nous par le mixin.

Nous pouvons assez facilement brancher cette vue dans nos URL :

`urls.py`

```

from django.conf.urls import url
from books.views import RecordInterest

urlpatterns = [
    #...
    url(r'^author/(?P<pk>[0-9]+)/interest/$', RecordInterest.as_view(),
        name='author-interest'),
]

```

Remarquez le groupe nommé pk qui sera utilisé par [get_object\(\)](#) pour rechercher l'instance Author. Il est aussi possible d'utiliser un « slug » ou toute autre fonctionnalité de [SingleObjectMixin](#).

Utilisation de SingleObjectMixin avec ListView

[ListView](#) intègre la pagination, mais il peut être par exemple souhaitable de paginer une liste d'objets qui sont tous liés (par clé étrangère) à un autre objet. Dans notre exemple de publication, on pourrait vouloir paginer tous les livres d'un éditeur particulier.

Une façon de faire cela serait de combiner [ListView](#) avec [SingleObjectMixin](#), afin que le jeu de requête de la liste de livres paginée puisse dépendre de l'éditeur trouvé comme objet unique. Afin de faire cela, il est nécessaire d'avoir deux jeux de requête différents :

Le jeu de requête Book utilisé par [ListView](#)

Comme nous avons accès à l'éditeur Publisher des livres que nous souhaitons afficher, nous surchargeons simplement `get_queryset()` et utilisons le [gestionnaire inverse de clé étrangère](#) de l'objet Publisher.

Le jeu de requête Publisher utilisé par [get_object\(\)](#)

Nous comptons sur l'implémentation par défaut de `get_object()` pour récupérer le bon objet Publisher. Cependant, nous devons explicitement transmettre un paramètre `queryset` car sinon, l'implémentation par défaut de `get_object()` appellerait `get_queryset()` que nous avons surchargée pour renvoyer des objets BOOK au lieu d'objets Publisher.

Note

Il faut être prudent avec `get_context_data()`. Comme les deux classes [SingleObjectMixin](#) et [ListView](#) vont placer des éléments dans les données de contexte sous la valeur `context_object_name`, si elle est définie, nous allons plutôt nous assurer de manière explicite que l'objet Publisher se trouve bien dans les données de contexte. [ListView](#) se chargera d'ajouter les contenus `page_obj` et `paginator` adéquats pour autant que nous n'oublions pas d'appeler `super()`.

Nous pouvons maintenant écrire une nouvelle vue `PublisherDetail`:

```

from django.views.generic import ListView
from django.views.generic.detail import SingleObjectMixin
from books.models import Publisher

class PublisherDetail(SingleObjectMixin, ListView):
    paginate_by = 2
    template_name = "books/publisher_detail.html"

    def get(self, request, *args, **kwargs):
        self.object = self.get_object(queryset=Publisher.objects.all())
        return super(PublisherDetail, self).get(request, *args, **kwargs)

    def get_context_data(self, **kwargs):
        context = super(PublisherDetail, self).get_context_data(**kwargs)
        context['publisher'] = self.object
        return context

    def get_queryset(self):
        return self.object.book_set.all()

```

Remarquez notre manière de définir `self.object` dans `get()` afin que nous puissions ensuite l'utiliser dans `get_context_data()` et `get_queryset()`. Si vous ne définissez pas `template_name`, le nom du gabarit prendra la valeur par défaut choisie par [ListView](#), ce qui donnera dans ce cas `"books/book_list.html"` car il s'agit d'une liste de livres. [ListView](#) n'a pas conscience de la présence de [SingleObjectMixin](#), elle ne sait donc pas du tout que cette vue a quelque chose à voir avec un objet `Publisher`.

Nous avons délibérément défini `paginate_by` à un petit nombre dans l'exemple afin que vous n'ayez pas à créer beaucoup de livres pour voir la pagination à l'œuvre ! Voici le gabarit que vous pourriez utiliser :

```

{% extends "base.html" %}

{% block content %}
    <h2>Publisher {{ publisher.name }}</h2>

    <ol>
        {% for book in page_obj %}
            <li>{{ book.title }}</li>
        {% endfor %}
    </ol>

    <div class="pagination">
        <span class="step-links">
            {% if page_obj.has_previous %}
                <a href="?page={{ page_obj.previous_page_number }}">previous</a>
            {% endif %}

            <span class="current">
                Page {{ page_obj.number }} of {{ paginator.num_pages }}.
            </span>

            {% if page_obj.has_next %}
                <a href="?page={{ page_obj.next_page_number }}">next</a>
            {% endif %}
        </span>
    </div>

```

```
        </span>
    </div>
{% endblock %}
```

Risque de trop grande complexité

Généralement, il est possible d'utiliser [TemplateResponseMixin](#) et [SingleObjectMixin](#) lorsque leurs fonctionnalités sont nécessaires. Comme montré plus haut, en prenant certaines précautions, on peut même combiner [SingleObjectMixin](#) avec [ListView](#). Cependant, les choses deviennent de plus en plus complexes avec ces combinaisons ; voici une bonne règle générale :

Indice

Chacune de vos vues ne devrait utiliser que des mixins ou des vues en provenance de l'un des groupes de vues génériques fondées sur les classes : [détail](#), [liste](#), [édition](#) et [date](#). Par exemple, il est convenable de combiner [TemplateView](#) (vue intégrée) avec [MultipleObjectMixin](#) (liste générique), mais vous rencontrerez sûrement des problèmes si vous combinez [SingleObjectMixin](#) (détail générique) avec [MultipleObjectMixin](#) (liste générique).

To show what happens when you try to get more sophisticated, we show an example that sacrifices readability and maintainability when there is a simpler solution. First, let's look at a naive attempt to combine [DetailView](#) with [FormMixin](#) to enable us to POST a Django [Form](#) to the same URL as we're displaying an object using [DetailView](#).

Utilisation de [FormMixin](#) avec [DetailView](#)

Rappelez-vous notre exemple précédent qui combinait [View](#) et [SingleObjectMixin](#). Nous enregistrons l'intérêt d'un utilisateur pour un auteur particulier. Admettons que nous voulions maintenant donner la possibilité d'écrire un message motivant leur intérêt. Nous partons encore une fois du principe que nous ne stockerons pas cela dans une base de données relationnelle, mais plutôt dans quelque chose de plus exotique dont nous ne nous soucierons pas dans cet exemple.

À ce stade, il est naturel de faire appel à un formulaire [Form](#) pour englober les informations envoyées par le navigateur de l'utilisateur vers Django. En admettant que nous adhérons aussi aux principes de [REST](#), nous aimerions utiliser la même URL pour l'affichage de l'auteur que pour la capture du message en provenance de l'utilisateur. Réécrivons notre vue `AuthorDetailView` dans cette optique.

Nous conserverons le traitement GET de la classe [DetailView](#), même s'il faudra rajouter un formulaire dans les données de contexte pour pouvoir l'afficher dans le gabarit. Nous allons aussi profiter du traitement de formulaire de [FormMixin](#) et écrire un peu de code afin qu'en cas d'envoi POST, le formulaire soit instancié de manière appropriée.

Note

Nous utilisons [FormMixin](#) et implémentons nous-même la méthode `post()` plutôt que d'essayer de mélanger [DetailView](#) avec [FormView](#) (qui fournit déjà une méthode `post()` exploitable), parce que ces deux vues implémentent `get()` et que cela risquerait d'ajouter de la confusion.

Voici à quoi ressemble notre nouvelle vue `AuthorDetail`:

```
# CAUTION: you almost certainly do not want to do this.
# It is provided as part of a discussion of problems you can
# run into when combining different generic class-based view
# functionality that is not designed to be used together.

from django import forms
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.views.generic import DetailView
from django.views.generic.edit import FormMixin
from books.models import Author

class AuthorInterestForm(forms.Form):
    message = forms.CharField()

class AuthorDetail(FormMixin, DetailView):
    model = Author
    form_class = AuthorInterestForm

    def get_success_url(self):
        return reverse('author-detail', kwargs={'pk': self.object.pk})

    def get_context_data(self, **kwargs):
        context = super(AuthorDetail, self).get_context_data(**kwargs)
        context['form'] = self.get_form()
        return context

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseRedirect()
        self.object = self.get_object()
        form = self.get_form()
        if form.is_valid():
            return self.form_valid(form)
        else:
            return self.form_invalid(form)

    def form_valid(self, form):
        # Here, we would record the user's interest using the message
        # passed in form.cleaned_data['message']
        return super(AuthorDetail, self).form_valid(form)
```

`get_success_url()` ne sert qu'à indiquer une destination de redirection, ce qui est utilisé dans l'implémentation par défaut de `form_valid()`. Nous devons fournir notre propre méthode `post()`, comme expliqué précédemment, et surcharger `get_context_data()` pour que le formulaire [Form](#) soit disponible dans les données de contexte.

Une meilleure solution

Il est assez évident que le nombre d'interactions subtiles entre [FormMixin](#) et [DetailView](#) éprouve déjà nos capacités conceptuelles. Il est improbable que vous ayez envie d'écrire vous-même ce genre de classe.

Dans ce cas, il serait relativement facile d'écrire simplement la méthode `post()` vous-même, en conservant [DetailView](#) comme seule fonctionnalité générique, même si l'écriture du code de gestion des formulaires implique beaucoup de duplication.

Sinon, une approche certainement plus simple que ci-dessus serait d'avoir une vue séparée pour le traitement du formulaire, ce qui permettrait d'utiliser sans problème [FormView](#) séparément de [DetailView](#).

Une meilleure solution alternative

En réalité, ce que nous essayons de faire ici est d'utiliser deux vues classes différentes pour la même URL. Pourquoi donc ne pas simplement faire cela ? La division est ici très claire : les requêtes GET devraient aboutir à la vue [DetailView](#) (avec en plus le formulaire dans les données de contexte) et les requêtes POST à la vue [FormView](#). Mettons d'abord en place ces deux vues.

La vue `AuthorDisplay` est presque la même que [celle que nous avons présentée en introduction](#). Nous devons écrire notre propre méthode `get_context_data()` pour mettre à disposition `AuthorInterestForm` dans le gabarit. Nous omettons la surcharge de `get_object()` de la version précédente par souci de clarté :

```
from django.views.generic import DetailView
from django import forms
from books.models import Author

class AuthorInterestForm(forms.Form):
    message = forms.CharField()

class AuthorDisplay(DetailView):
    model = Author

    def get_context_data(self, **kwargs):
        context = super(AuthorDisplay, self).get_context_data(**kwargs)
        context['form'] = AuthorInterestForm()
        return context
```

Puis pour `AuthorInterest`, il s'agit d'une simple sous-classe de [FormView](#), mais nous devons y adjoindre [SingleObjectMixin](#) pour pouvoir obtenir l'auteur concerné par l'action en cours et ne pas oublier de définir `template_name` pour s'assurer que les erreurs de formulaires soient affichées dans le même gabarit que `AuthorDisplay` utilise lors de la requête GET:

```
from django.core.urlresolvers import reverse
from django.http import HttpResponseRedirect
from django.views.generic import FormView
from django.views.generic.detail import SingleObjectMixin
```

```

class AuthorInterest(SingleObjectMixin, FormView):
    template_name = 'books/author_detail.html'
    form_class = AuthorInterestForm
    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseRedirect()
        self.object = self.get_object()
        return super(AuthorInterest, self).post(request, *args, **kwargs)

    def get_success_url(self):
        return reverse('author-detail', kwargs={'pk': self.object.pk})

```

Pour terminer, nous lions le tout en créant une nouvelle vue `AuthorDetail`. Nous savons déjà que l'appel à [`as_view\(\)`](#) sur une vue fondée sur les classes nous renvoie un objet se comportant exactement comme une vue de type fonction. Nous pouvons donc effectuer cet appel au moment où il faut choisir entre les deux « sous-vues ».

Vous pouvez bien sûr transmettre des paramètres nommés à [`as_view\(\)`](#) comme vous le feriez dans la configuration d'URL, par exemple si vous vouliez que le comportement de `AuthorInterest` puisse aussi être utilisé avec une autre URL mais en affichant un gabarit différent :

```

from django.views.generic import View

class AuthorDetail(View):

    def get(self, request, *args, **kwargs):
        view = AuthorDisplay.as_view()
        return view(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        view = AuthorInterest.as_view()
        return view(request, *args, **kwargs)

```

Cette approche peut également être employée avec toute autre vue fondée sur les classes, générique ou créée par vos soins, héritant directement de [`View`](#) ou de [`TemplateView`](#), car elle conserve les différentes vues aussi distinctes que possible.

Plus que du simple code HTML

Les vues fondées sur les classes se révèlent particulièrement utiles au moment où l'on a besoin de répéter plusieurs fois un processus semblable. Imaginez qu'il faille écrire une API et que chaque vue doive renvoyer du JSON au lieu de produire une page HTML.

Il est alors possible de créer une classe mixin afin de l'utiliser dans chaque vue, gérant ainsi à un seul endroit la conversion vers le format JSON.

Par exemple, une simple classe mixin pour JSON pourrait ressembler à ceci :

```

from django.http import JsonResponse

```

```

class JsonResponseMixin(object):
    """
    A mixin that can be used to render a JSON response.
    """
    def render_to_json_response(self, context, **response_kwargs):
        """
        Returns a JSON response, transforming 'context' to make the payload.
        """
        return JsonResponse(
            self.get_data(context),
            **response_kwargs
        )

    def get_data(self, context):
        """
        Returns an object that will be serialized as JSON by json.dumps().
        """
        # Note: This is *EXTREMELY* naive; in reality, you'll need
        # to do much more complex handling to ensure that arbitrary
        # objects -- such as Django model instances or querysets
        # -- can be serialized as JSON.
        return context

```

Note

Consultez la documentation [Sérialisation d'objets Django](#) pour plus d'informations sur la façon de transformer correctement des modèles Django et des jeux de requête au format JSON.

Ce mixin fournit une méthode `render_to_json_response()` ayant la même signature que [render_to_response\(\)](#). Pour l'utiliser, nous devons simplement la combiner par exemple à une classe `TemplateView` et surcharger `render_to_response()` afin d'appeler `render_to_json_response()` à la place :

```

from django.views.generic import TemplateView

class JSONView(JsonResponseMixin, TemplateView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)

```

Il serait aussi possible d'utiliser cette classe mixin avec l'une des vues génériques. On peut composer sa propre version de [DetailView](#) en combinant `JsonResponseMixin` avec `django.views.generic.detail.BaseDetailView` (celle-ci contenant le comportement de [DetailView](#) avant le rendu du gabarit) :

```

from django.views.generic.detail import BaseDetailView

class JSONDetailView(JsonResponseMixin, BaseDetailView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)

```

Cette vue peut ensuite être déployée de la même façon que pour n'importe quelle vue [DetailView](#), reproduisant le même comportement à l'exception du format de la réponse.

Si vous avez de réels goûts d'aventure, vous pourriez même tenter de combiner avec une sous-classe de [DetailView](#) capable de renvoyer *à la fois* du contenu HTML et JSON en fonction d'une propriété de la requête HTTP, telle qu'un paramètre de requête ou un en-tête HTTP. Il suffit de combiner `JSONResponseMixin` avec [SingleObjectTemplateResponseMixin](#) et de surcharger l'implémentation de [render_to_response\(\)](#) pour déléguer le rendu à la méthode appropriée en fonction du type de réponse demandée par l'utilisateur :

```
from django.views.generic.detail import SingleObjectTemplateResponseMixin

class HybridDetailView(JSONResponseMixin, SingleObjectTemplateResponseMixin,
BaseDetailView):
    def render_to_response(self, context):
        # Look for a 'format=json' GET argument
        if self.request.GET.get('format') == 'json':
            return self.render_to_json_response(context)
        else:
            return super(HybridDetailView, self).render_to_response(context)
```

En raison de la façon dont Python résout la surcharge des méthodes, l'appel à `super(HybridDetailView, self).render_to_response(context)` fait en réalité appel à l'implémentation [render_to_response\(\)](#) de [TemplateResponseMixin](#).

[Gestion de formulaires avec les vues fondées sur les classes](#)
[Migrations](#)