Du fait que Django a été développé dans un environnement de rédaction avec des délais très courts, il a été conçu pour rendre les tâches habituelles du développement Web rapides et simples. Voici un bref aperçu sur la manière d'écrire une application utilisant une base de données avec Django.

Le but de ce document est de vous donner assez de détails techniques pour comprendre comment fonctionne Django, mais il n'a pas pour but d'être un didacticiel ou une référence – mais nous avons cela aussi ! Quand vous êtes prêt à commencer un projet, vous pouvez démarrer avec le tutoriel ou vous plonger dans une documentation plus détaillée.

Concevez votre modèle

Bien que vous puissiez utiliser Django sans base de données, il est livré avec un mapping objet-relationnel avec lequel vous décrivez la structure de votre base de données avec du code Python.

La syntaxe de modélisation des données offre un moyen élégant de représenter vos modèles – jusqu'ici, cela a résolu bien des années de problèmes de schéma de base de données. Voici un exemple rapide :

mysite/news/models.py

```python
from django.db import models


class Reporter(models.Model):
    full_name = models.CharField(max_length=70)


    def __str__(self):              # __unicode__ on Python 2
        return self.full_name


class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)


    def __str__(self):              # __unicode__ on Python 2
        return self.headline
```

Déployez-le

Ensuite, lancez l'utilitaire en ligne de commande pour créer les tables de la base de données automatiquement :

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

La commande migrate examine tous vos modèles disponibles et crée les tables correspondantes dans votre base de données pour toutes celles qui n'existent pas encore, et fournit également de manière facultative un contrôle plus riche encore du schéma.

Profitez de l'API qui vous est offerte¶

Dès lors, vous avez gratuitement accès à une API Python riche pour manipuler vos données. L'API est générée à la volée, sans avoir besoin d'écrire du code :

```
# Import the models we created from our "news" app
>>> from news.models import Reporter, Article


# No reporters are in the system yet.
>>> Reporter.objects.all()
<QuerySet []>


# Create a new Reporter.
>>> r = Reporter(full_name='John Smith')


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


# Now it has an ID.
>>> r.id
1


# Now the new reporter is in the database.
>>> Reporter.objects.all()
<QuerySet [<Reporter: John Smith>]>


# Fields are represented as attributes on the Python object.
>>> r.full_name
'John Smith'
```

```
# Django provides a rich database lookup API.
>>> Reporter.objects.get(id=1)
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__startswith='John')
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__contains='mith')
<Reporter: John Smith>
>>> Reporter.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Reporter matching query does not exist.

# Create an article.
>>> from datetime import date
>>> a = Article(pub_date=date.today(), headline='Django is cool',
...     content='Yeah.', reporter=r)
>>> a.save()

# Now the article is in the database.
>>> Article.objects.all()
<QuerySet [<Article: Django is cool>]>

# Article objects get API access to related Reporter objects.
>>> r = a.reporter
>>> r.full_name
'John Smith'

# And vice versa: Reporter objects get API access to Article objects.
>>> r.article_set.all()
```

```
<QuerySet [<Article: Django is cool>]>


# The API follows relationships as far as you need, performing efficient
# JOINs for you behind the scenes.
# This finds all articles by a reporter whose name starts with "John".
>>> Article.objects.filter(reporter__full_name__startswith='John')
<QuerySet [<Article: Django is cool>]>


# Change an object by altering its attributes and calling save().
>>> r.full_name = 'Billy Goat'
>>> r.save()


# Delete an object with delete().
>>> r.delete()
```

Une interface d'administration dynamique : ce n'est pas juste un échafaudage, c'est une vraie maison.¶

Une fois que vos modèles sont définis, Djando peut créer automatiquement une interface d'administration professionnelle et apte à la production, un site Web qui permet aux utilisateurs authentifiés d'ajouter, modifier et supprimer des objets. C'est aussi simple que d'enregistrer votre modèle dans le site d'administration :

mysite/news/models.py

```
from django.db import models


class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)
```

mysite/news/admin.py

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

```
from . import models


admin.site.register(models.Article)
```

Ici, on part du principe que votre site peut être modifié par un membre du personnel, un client ou peut-être juste par vous, sans que vous deviez créer des interfaces d'administration uniquement pour gérer le contenu.

Un schéma habituel lors de la création d'une application Django est de créer les modèles et de mettre en place les sites d'administration aussi vite que possible, pour que votre équipe (ou clients) puisse commencer à saisir les données. Ensuite, le développement de la couche de présentation publique des données peut avancer à son rythme.

Conception des URL

Un schéma d'URL propre et élégant est un aspect important dans une application Web de qualité. Django encourage la conception de belles URL et ne place aucune extension dans celles-ci, comme **.php** ou **.asp**.

Pour concevoir les URL d'une application, vous créez un module Python appelé URLconf. C'est un sommaire pour votre application, il contient les liaisons entre vos motifs d'URL et les fonctions Python associées. Ces URLconfs servent aussi à séparer les URL de votre code Python.

Voici à quoi peut ressembler un URLconf pour l'exemple de **Reporter**/**Article** précédent :

mysite/news/urls.py

```
from django.conf.urls import url


from . import views


urlpatterns = [
    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),
]
```

Le code ci-dessus relie, avec de simples expressions régulières, les URL aux fonctions Python réceptrices (« views »). Les expressions régulières utilisent les parenthèses pour « capturer » les valeurs à partir des URL. Quand un utilisateur demande une page, Django parcourt tous les motifs, dans l'ordre, et s'arrête dès qu'un de ces motifs correspond à l'URL demandée (si aucun d'eux ne correspond, Django appelle une vue 404 spéciale). Ceci est extrêmement rapide, car les expressions régulières sont compilées au chargement.

Une fois qu'une des expressions régulières correspond, Django appelle la vue associée, qui est une fonction Python. Chaque vue reçoit un objet requête (« request ») qui contient les métadonnées de la requête ainsi que les valeurs capturées dans l'expression régulière.

Par exemple, si un utilisateur demande l'URL « /articles/2005/05/39323/ », Django appellera la fonction **news.views.article_detail(request, '2005', '05', '39323')**.

Écriture des vues

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.

Généralement, une vue récupère des données d'après les paramètres, charge un template et affiche le template avec les données récupérées. Voici un exemple de vue pour l'exemple précédent **year_archive** :

mysite/news/views.py

```
from django.shortcuts import render


from .models import Article


def year_archive(request, year):
    a_list = Article.objects.filter(pub_date__year=year)
    context = {'year': year, 'article_list': a_list}
    return render(request, 'news/year_archive.html', context)
```

Cet exemple utilise le système de gabarits de Django qui a plusieurs fonctions puissantes mais s'efforce de rester assez simple à l'utilisation pour les non-programmeurs.

Élaboration de vos gabarits

Le code ci-dessus charge le gabarit **news/year_archive.html**.

L'algorithme de recherche des gabarits de Django vous permet de minimiser la redondance parmi les gabarits. Dans vos réglages de Django, vous indiquez une liste des dossiers contenant potentiellement des gabarits avec **DIRS**. Si un gabarit n'existe pas dans le premier dossier, Django vérifie le deuxième, etc.

Admettons que le gabarit **news/article_detail.html** a été trouvé. Voilà à quoi il pourrait ressembler :

mysite/news/templates/news/year_archive.html

```
{% extends "base.html" %}


{% block title %}Articles for {{ year }}{% endblock %}


{% block content %}
```

```
<h1>Articles for {{ year }}</h1>


{% for article in article_list %}
    <p>{{ article.headline }}</p>
    <p>By {{ article.reporter.full_name }}</p>
    <p>Published {{ article.pub_date|date:"F j, Y" }}</p>
{% endfor %}
{% endblock %}
```

Les variables sont entourées par deux accolades. **{{ article.headline }}** veut dire « Afficher la valeur de l'attribut titre de l'article ». Mais les points ne sont pas utilisés uniquement pour l'utilisation des attributs. Ils peuvent aussi être utilisés pour la recherche des clés de dictionnaires, d'index, ou pour appeler des fonctions.

Notez que **{{ article.pub_date|date:"F j, Y" }}** utilise des « tuyaux » de type Unix (le caractère «|» (pipe)). On appelle cela un filtre de gabarit, c'est un moyen de filtrer la valeur d'une variable. Dans le cas présent, le filtre « date » formate un objet Python de type « datetime » dans le format fourni (comme dans la fonction « date » de PHP).

Vous pouvez enchaîner autant de filtres que vous le voulez. Vous pouvez écrire vos propres filtres de gabarits. Vous pouvez écrire des balises de gabarits personnalisées qui utilisent du code Python en arrière-plan.

Enfin, Django utilise le concept d'"« héritage de gabarits ». C'est ce que fait **{% extends "base.html" %}**. Cela veut dire « Charge premièrement le gabarit nommé "base", qui a défini certains blocs, et remplit ces blocs avec le contenu qui suit. ». Pour résumer, cela permet de diminuer significativement la redondance dans les gabarits : chaque gabarit ne doit définir que ce qui lui est propre.

Voici à quoi le gabarit « base.html » pourrait ressembler, y compris l'utilisation de fichiers statiques :

mysite/templates/base.html

```
{% load static %}
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <img src="{% static "images/sitelogo.png" %}" alt="Logo" />
    {% block content %}{% endblock %}
</body>
```

```
</html>
```

De façon simplifiée, il définit l'aspect général de votre site (avec le logo) et positionne des « trous » que les gabarits enfants peuvent remplir. De cette façon, le changement de style d'un site devient un jeu d'enfant : il suffit de changer un seul fichier, le gabarit de base.

Cela vous permet aussi de créer plusieurs versions d'un site, avec des gabarits de base différents, tout en réutilisant les gabarits enfants. Les créateurs de Django ont utilisé cette technique pour créer des versions « mobiles » très différentes de leurs sites, en créant simplement un nouveau gabarit de base.

Notez que vous n'avez pas à utiliser le système de gabarits de Django si vous en préférez un autre. Bien que le système de gabarits de Django soit particulièrement bien intégré avec la couche modèle de Django, rien ne vous force à l'utiliser. Vous n'êtes pas non plus obligé d'utiliser l'API de base de données de Django. Il est possible d'utiliser une autre couche d'abstraction de base de données, de lire des fichiers XML, des fichiers sur le disque ou toute autre solution. Chaque partie de Django — modèles, vues, gabarits — est découplée des autres.

Ceci n'est que la base

Ceci n'était qu'un rapide coup d'œil sur les fonctions de Django. En voici quelques autres utiles :

- Un système de cache qui s'intègre avec « memcached » ou d'autres moteurs.
- Un système de syndication qui rend la génération de flux RSS et Atom aussi simple que d'écrire des petites classes Python.
- D'autres sympatiques fonctionnalités automatiquement générées pour l'interface d'administration, cet aperçu a tout juste effleuré la surface.

Pour vous, les étapes suivantes sont probablement de télécharger Django, de lire le didacticiel et de rejoindre la communauté. Merci de votre intérêt !

Guide d'installation rapide¶

Avant de pouvoir utiliser Django, il vous faut l'installer. Le guide complet d'installation couvre toutes les possibilités, mais cette page vous guidera pour une installation simple et minimale qui sera suffisante pour que vous puissiez suivre toute l'introduction.

Installation de Python¶

Django est un framework Python, il a donc besoin de Python. Consultez Quelle version de Python puis-je utiliser avec Django ? pour plus de détails. Django fonctionne avec Python 2.6, 2.7, 3.2 ou 3.3. Python inclut une base de données légère appelée SQLite, il n'est donc pas nécessaire de configurer une base de données pour le moment.

Obtenez la dernière version de Python à l'adresse https://www.python.org/downloads/ 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.

Vous pouvez vérifier que Python est installé en saisissant **python** dans votre shell ; vous devriez voir quelque chose qui ressemble à :

```
Python 3.4.x

[GCC 4.x] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>>
```

Configuration d'une base de données

Cette étape n'est nécessaire que si vous souhaitez travailler avec un moteur de base de données plus « solide », comme PostgreSQL, MySQL ou Oracle. Pour installer une telle base de données, consultez Informations sur l'installation de bases de données.

Désinstallation de toute version plus ancienne de Django

Si vous mettez à jour votre installation de Django depuis une version précédente, vous devez désinstaller l'ancienne version de Django avant d'installer la nouvelle.

Installation de Django

Vous avez trois possibilités pour installer facilement Django :

- Installer une version officielle. C'est la meilleure approche pour la plupart des utilisateurs.
- Installer une version de Django fournie dans les paquets de votre système.
- Installer la dernière version de développement. Cette option est réservée aux enthousiastes qui souhaitent les fonctions les plus récentes et qui ne craignent pas d'utiliser un code récent. Il est possible que de nouvelles anomalies apparaissent dans cette version de développement, mais en les signalant, vous aiderez au développement de Django. De même, il y a plus de risques que les paquets tiers ne soient pas compatible avec cette version, comparé à la dernière version stable.

Consultez toujours la documentation correspondant à la version de Django que vous utilisez !

Si vous choisissez une des deux premières options, prêtez attention aux parties de la documentation marquées par **nouveau dans la version de développement**. Cette phrase indique des fonctionnalités qui ne sont disponibles que dans les versions de développement de Django ; si vous essayez de les utiliser avec une version officielle, elles ne fonctionneront pas.

Vérification

Pour contrôler que Django est accessible par Python, saisissez **python** dans votre shell. Puis, à l'invite de commande Python, essayez d'importer Django :

```
>>> import django

>>> print(django.get_version())
```

```
1.11
```

Il se peut que vous ayez installé une autre version de Django.

Écriture de votre première application Django

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 supposerons que Django est déjà installé. Vous pouvez savoir si Django est installé et sa version en exécutant la commande suivante dans un terminal (indiqué par le préfixe `$`) :

```
$ 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 1.11 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.

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 : un fichier vide qui indique à Python que ce répertoire doit être considéré comme un paquet. Si vous êtes débutant en Python, lisez les informations sur les paquets (en) dans la documentation officielle de Python.
- 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.



décembre 02, 2017 - 15:50:53

Django version 1.11, 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. Par exemple, pour écouter toutes les IP publiques disponibles (ce qui est utile si vous exécutez Vagrant ou que souhaitez montrer votre travail à des personnes sur d'autres ordinateurs de votre réseau), faites :

```
$ python manage.py runserver 0:8000
```

0 est un raccourci pour **0.0.0.0**. 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.

Vos applications peuvent se trouver à n'importe quel endroit de votre chemin de recherche Python. Dans ce didacticiel, nous allons créer une application de sondage au même niveau que le fichier **`manage.py`** pour qu'elle puisse être importée comme module de premier niveau plutôt que comme sous-module de **`monsite`**.

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),
]
```

La fonction **include()** permet de référencer d'autres configurations d'URL. Remarquez que les expressions régulières dans la fonction **include()** n'ont pas de **$** (caractère de fin de correspondance) mais plutôt une barre oblique finale. Quand Django rencontre un **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.

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 interessant 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¶

Le nommage des URL permet de les référencer de manière non ambiguë depuis d'autres portions de code Django, en particulier depuis les gabarits. Cette fonctionnalité puissante permet d'effectuer des changements globaux dans les modèles d'URL de votre projet en ne modifiant qu'un seul fichier.

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.

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 résistante à la charge 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 empaqueter 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 FROMUSER_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 règlage **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'introspection 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 o 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.

Pour inclure l'application dans notre projet, nous avons besoin d'ajouter une référence à sa classe de configuration dans le réglage **INSTALLED_APPS**. La classe **PollsConfig** se trouve dans le fichier **polls/apps.py**, ce qui signifie que son chemin pointé est **'polls.apps.PollsConfig'**. Modifiez le fichier **mysite/settings.py** et ajoutez ce chemin pointé au réglage **INSTALLED_APPS**. Il doit 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':
```

```
  polls/migrations/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 consultez 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, auth, contenttypes, polls, 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()

<QuerySet []>


# 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()
<QuerySet [<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 **pythonmanage.py shell**:

```
>>> from polls.models import Question, Choice


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

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

# 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()
<QuerySet [<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)

<QuerySet [<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 :

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 :

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 :

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 :

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

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 :