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 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 4.2, sur une base Python 3.8 (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 une version plus ancienne de Python, consultez Quelle version de Python puis-je utiliser avec Django ? pour trouver la version de Django qui correspond.

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 rencontrez des problèmes dans le parcours de ce tutoriel, rendez-vous dans la section Obtenir de l'aide de la FAQ.

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
        asgi.py
        wsgi.py
```

Ces fichiers sont :

- Le premier répertoire racine `mysite/` est 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/asgi.py` : un point d'entrée pour les serveurs Web compatibles aSGI pour déployer votre projet. Voir Comment déployer avec ASGI pour plus de détails.
- `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.

juin 09, 2023 - 15:50:53
Django version 4.2, 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 « Félicitations ! » ainsi qu'une fusée qui décolle. 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 0.0.0.0:8000
```

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 petite 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 dans le même dossier 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.urls import path

from . import views

urlpatterns = [
    path("", views.index, name="index"),
]
```

L'étape suivante est de faire pointer la configuration d''"URL racine vers le module `polls.urls`. Dans `mysite/urls.py`, ajoutez une importation `django.urls.include` et insérez un appel `include()` dans la liste `urlpatterns`, ce qui donnera :

`monsite/urls.py`¶

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path("polls/", include("polls.urls")),
    path("admin/", admin.site.urls),
]
```

La fonction `include()` permet de référencer d'autres configurations d'URL. 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.

Vous avez maintenant relié une vue `index` dans la configuration d'URL. Vérifiez qu'elle fonctionne avec 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`.

Page non trouvée ?

Si vous obtenez une page d'erreur ici, vérifiez que vous accédez bien à http://localhost:8000/polls/ et non pas à http://localhost:8000/.

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

Paramètre de `path()` : `route`¶

`route` est une chaîne contenant un motif d'URL. Lorsqu'il traite une requête, Django commence par le premier motif dans `urlpatterns` puis continue de parcourir la liste en comparant l'URL reçue avec chaque motif jusqu'à ce qu'il en trouve un qui correspond.

Les motifs 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/`.

Paramètre de `path()` : `view`¶

Lorsque Django trouve un motif correspondant, il appelle la fonction de vue spécifiée, avec un objet `HttpRequest` comme premier paramètre et toutes les valeurs « capturées » par la route sous forme de paramètres nommés. Nous montrerons cela par un exemple un peu plus loin.

Paramètre de `path()` : `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 de `path()` : `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.

Où obtenir de l'aide :

Si vous rencontrez des problèmes dans le parcours de ce tutoriel, rendez-vous dans la section Obtenir de l'aide de la FAQ.

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, `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;` (MariaDB, MySQL), `.tables` (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 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 au fond 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, 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 des 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)
```

Ici, 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 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.

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 Question
    - Create model 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), il s'agit de fichiers sur un 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 Question
--
CREATE TABLE "polls_question" (
    "id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
--
-- Create model Choice
--
CREATE TABLE "polls_choice" (
    "id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL,
    "question_id" bigint NOT NULL
);
ALTER TABLE "polls_choice"
  ADD CONSTRAINT "polls_choice_question_id_c5b4b260_fk_polls_question_id"
    FOREIGN KEY ("question_id")
    REFERENCES "polls_question" ("id")
    DEFERRABLE INITIALLY DEFERRED;
CREATE INDEX "polls_choice_question_id_c5b4b260" ON "polls_choice"
("question_id");

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 indiquent 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), `` `bigint PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY `` (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`.

Une fois dans le shell, explorez l'API de base de données:

```
>>> from polls.models import Choice, Question  # 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.
>>> 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=datetime.timezone.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 (1)>]>
```

Une seconde. `<Question: Question object (1)>` n'est pas une représentation très utile de cet objet. 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


class Question(models.Model):
    # ...
    def __str__(self):
        return self.question_text


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.

Ajoutons aussi une méthode personnalisée à ce modèle :

`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 Choice, Question

# 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, si vous définissez `LANGUAGE_CODE`, l’écran de connexion s’affiche dans cette langue (pour autant que les traductions correspondantes existent dans Django).

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.

Plus qu’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 :

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, 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 ».

Où obtenir de l'aide :

Si vous rencontrez des problèmes dans le parcours de ce tutoriel, rendez-vous dans la section Obtenir de l'aide de la FAQ.

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 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.htm?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 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 configurations d'URL (URLconf). Une configuration d'URL associe des motifs d'URL à des vues.

Ce tutoriel fournit des instructions de base sur l'utilisation des configurations d'URL, et vous pouvez consultez Distribution des URL 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 `path()` suivants :

`polls/urls.py`¶

```
from django.urls import path

from . import views

urlpatterns = [
    # ex: /polls/
    path("", views.index, name="index"),
    # ex: /polls/5/
    path("<int:question_id>/", views.detail, name="detail"),
    # ex: /polls/5/results/
    path("<int:question_id>/results/", views.results, name="results"),
    # ex: /polls/5/vote/
    path("<int:question_id>/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.

Lorsque quelqu'un demande une page de votre site Web, par exemple « /polls/34/ », Django charge le module Python `mysite.urls` parce qu'il est mentionné dans le réglage `ROOT_URLCONF`. Il trouve la variable nommée `urlpatterns` et parcourt les motifs dans l'ordre. Après avoir trouvé la correspondance `'polls/'`, il retire le texte correspondant (`"polls/"`) et passe le texte restant – `"34/"` – à la configuration d'URL "polls.urls" pour la suite du traitement. Là, c'est `'<int:question_id>/'` qui correspond ce qui aboutit à un appel à la vue `detail()` comme ceci :

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

La partie `question_id=34` vient de `<int:question_id>`. En utilisant des chevrons, cela « capture » une partie de l'URL l'envoie en tant que paramètre nommé à la fonction de vue ; la partie `question_id` de la chaîne définit le nom qui va être utilisé pour identifier le motif trouvé, et la partie `int` est un convertisseur qui détermine ce à quoi les motifs doivent correspondre dans cette partie du chemin d'URL. Le caractère deux-points (`:`) sépare le convertisseur du nom de la partie capturée.

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 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 meilleure manière 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 %}
```

Note

Pour ne pas rallonger le tutoriel, tous les exemples de gabarit utilisent du HTML incomplet. Dans vos propres projets, vous devriez utiliser des documents HTML complets.

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 `path()` 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
path("<int:question_id>/", 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'
path("specifics/<int:question_id>/", 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.urls import path

from . import views

app_name = "polls"
urlpatterns = [
    path("", views.index, name="index"),
    path("<int:question_id>/", views.detail, name="detail"),
    path("<int:question_id>/results/", views.results, name="results"),
    path("<int:question_id>/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 les bases de la gestion de formulaires et des vues génériques.

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 et sur la réduction du code.

Où obtenir de l'aide :

Si vous rencontrez des problèmes dans le parcours de ce tutoriel, rendez-vous dans la section Obtenir de l'aide de la FAQ.

Écriture d'un formulaire minimal¶

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`¶

```
<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
<fieldset>
    <legend><h1>{{ question.question_text }}</h1></legend>
    {% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}
    {% 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 %}
</fieldset>
<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 pratique 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`¶

```
path("<int:question_id>/vote/", views.vote, name="vote"),
```

Nous avions 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.http import HttpResponse, HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls 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 F() 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 très courtes – 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 pourrons ainsi supprimer une partie de notre code. Nous avons quelques pas à faire pour effectuer 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.urls import path

from . import views

app_name = "polls"
urlpatterns = [
    path("", views.IndexView.as_view(), name="index"),
    path("<int:pk>/", views.DetailView.as_view(), name="detail"),
    path("<int:pk>/results/", views.ResultsView.as_view(), name="results"),
    path("<int:question_id>/vote/", views.vote, name="vote"),
]
```

Notez que le nom du motif correspondant dans les chaînes de chemin 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.http import HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls 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, 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.

Où obtenir de l'aide :

Si vous rencontrez des problèmes dans le parcours de ce tutoriel, rendez-vous dans la section Obtenir de l'aide de la FAQ.

Introduction aux tests automatisés¶

Que sont les tests automatisés ?¶

Les tests sont des 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 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 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 <http://fr.wikipedia.org/wiki/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 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).

Confirmez l’existence du bogue en utilisant le `shell` pour contrôler la méthode sur une question dont la date se situe dans le futur :

```
$ python manage.py 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.test import TestCase
from django.utils import timezone

from .models import Question
```

```
class QuestionModelTests(TestCase):
    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() returns False for questions whose pub_date
        is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertIs(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'...
System check identified no issues (0 silenced).
F
======================================================================
FAIL: test_was_published_recently_with_future_question
(polls.tests.QuestionModelTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in
test_was_published_recently_with_future_question
    self.assertIs(future_question.was_published_recently(), False)
AssertionError: True is not False

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

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

Erreur différente ?

Si vous obtenez ici une erreur `NameError`, vous avez peut-être manqué une étape à la partie 2 du tutoriel où des importations de `datetime` et de `timezone` ont été ajoutées à `polls/models.py`. Recopiez ces importations et essayez de relancer les tests.

Voici ce qui s’est passé :

- La commande `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 `assertIs()`, elle a découvert que sa méthode `was_published_recently()` renvoyait `True`, alors que nous souhaitions 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'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
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 avions 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() returns False for questions whose pub_date
    is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=1, seconds=1)
    old_question = Question(pub_date=time)
    self.assertIs(old_question.was_published_recently(), False)


def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() returns True for questions whose pub_date
    is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=23, minutes=59, seconds=59)
    recent_question = Question(pub_date=time)
    self.assertIs(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 minimale, 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 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`:

```
$ python manage.py 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. Si le réglage `TIME_ZONE` dans `settings.py` n'est pas correct, il se peut que certains résultats soient faussés. Si vous n'avez pas pensé à le régler précédemment, vérifiez-le avant de continuer.

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("/")
Not Found: /
>>> # we should expect a 404 from that address; if you instead see an
>>> # "Invalid HTTP_HOST header" error and a 400 response, you probably
>>> # omitted the setup_test_environment() call described earlier.
>>> 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.urls import reverse
>>> response = client.get(reverse("polls:index"))
>>> response.status_code
200
>>> response.content
b'\n    <ul>\n    \n        <li><a href="/polls/1/">What&#x27;s up?</a></li>\n
\n    </ul>\n\n'
>>> response.context["latest_question_list"]
<QuerySet [<Question: What's up?>]>
```

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). Corrigeons 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.urls 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):
    """
    Create 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 QuestionIndexViewTests(TestCase):
    def test_no_questions(self):
        """
        If no questions exist, an appropriate message is 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_past_question(self):
        """
        Questions with a pub_date in the past are displayed on the
        index page.
        """
        question = create_question(question_text="Past question.", days=-30)
        response = self.client.get(reverse("polls:index"))
        self.assertQuerySetEqual(
            response.context["latest_question_list"],
            [question],
        )

    def test_future_question(self):
        """
        Questions with a pub_date in the future aren't 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_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        are displayed.
```

```
        """
        question = 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],
        )

    def test_two_past_questions(self):
        """
        The questions index page may display multiple questions.
        """
        question1 = create_question(question_text="Past question 1.", days=-30)
        question2 = create_question(question_text="Past question 2.", days=-5)
        response = self.client.get(reverse("polls:index"))
        self.assertQuerySetEqual(
            response.context["latest_question_list"],
            [question2, question1],
        )
```

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_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_past_question`, nous créons une question et vérifions qu'elle apparaît dans la liste.

Dans `test_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())
```

Il est ensuite nécessaire d'ajouter 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 QuestionDetailViewTests(TestCase):
    def test_future_question(self):
        """
        The detail view of a question with a pub_date in the future
        returns 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_past_question(self):
        """
        The detail view of a question with a pub_date in the past
        displays 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, 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 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.

Où obtenir de l'aide :

Si vous rencontrez des problèmes dans le parcours de ce tutoriel, rendez-vous dans la section Obtenir de l'aide de la FAQ.

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 référez 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 meilleur moyen 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 static %}

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

La balise de gabarit `{% static %}` génère l'URL absolue des fichiers statiques.

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

Démarrez le serveur (ou redémarrez-le s'il tourne déjà) :

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

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 ce répertoire, ajoutez le fichier image que vous souhaitez utiliser comme arrière-plan. Dans le cadre de ce tutoriel, nous utiliserons un fichier nommé `background.png`, dont le chemin complet sera `polls/static/polls/images/background.png`.

Puis, ajoutez une référence à votre image dans la feuille de style (`polls/static/polls/style.css`) :

`polls/static/polls/style.css`¶

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

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

Avertissement

La balise de gabarit `{% static %}` n'est pas disponible dans les fichiers statiques qui ne sont pas générés par Django, comme une feuille de style. 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 inclues 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, 7ème partie¶

Ce tutoriel commence là où le tutoriel 6 s'achève. Nous continuons l'application de sondage Web et allons nous focaliser sur la personnalisation du site d'administration généré automatiquement par Django, que nous avons déjà exploré dans le tutoriel 2.

Où obtenir de l'aide :

Si vous rencontrez des problèmes dans le parcours de ce tutoriel, rendez-vous dans la section Obtenir de l'aide de la FAQ.

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)
```

Vous suivrez cette méthode – c'est-à-dire créer une classe d'administration de modèle, puis le transmettre en tant que deuxième paramètre de `admin.site.register()` – à chaque fois que vous aurez besoin de modifier les options d'administration pour un modèle.

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 :

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`:

`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 le lien « Ajouter une autre question » à côté de « Question ». Chaque objet avec une relation `ForeignKey` vers un autre objet gagne ceci automatiquement. Quand vous cliquez sur « Ajouter une autre question », vous obtenez une fenêtre modale avec le formulaire « Ajouter une question ». Si vous ajoutez une question dans cette fenêtre et cliquez sur « Enregistrer », Django enregistrera la question en base et l'ajoutera dynamiquement dans la sélection du formulaire « Ajouter un choix » que vous voyez.

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 :

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. Cette image montre l'emplacement ajouté :

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. Pour l'utiliser, modifiez 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 :

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 :

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"]
```

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 :

Vous pouvez cliquer sur les en-têtes de colonne pour trier selon ces valeurs – sauf dans le cas de l'en-tête `was_published_recently`, 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_recently` 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 appliquant le décorateur `display()` à cette méthode (dans `polls/models.py`), comme ceci :

`polls/models.py`¶

```
from django.contrib import admin


class Question(models.Model):
    # ...
    @admin.display(
        boolean=True,
        ordering="pub_date",
        description="Published recently?",
    )
    def was_published_recently(self):
        now = timezone.now()
        return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

Pour plus d'informations sur les propriétés configurables par ce décorateur, consultez `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` :

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, vous pouvez le changer en utilisant le système de gabarits de Django. Le site d'administration de Django est écrit lui-même avec Django, 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": [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

Tout comme pour les fichiers statiques, nous *pourrions* placer tous nos gabarits à un seul endroit et tout fonctionnerait très bien. Cependant, les gabarits qui appartiennent à une application particulière devraient se trouver dans le répertoire des gabarits de l'application (par ex. `polls/templates`) plutôt que dans celui du projet (`templates`). Nous aborderons plus en détails les raisons de ces choix dans le tutoriel sur les applications réutilisables.

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 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 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.

Lorsque vous vous serez familiarisé avec le site d'administration, lisez la partie 8 de ce tutoriel pour apprendre comment intégrer des paquets tiers.---

---

Écriture de votre première application Django, 8ème partie¶

Ce tutoriel commence là où le tutoriel 7 s'est arrêté. Nous avons construit une application de sondage web et allons maintenant examiner les paquets tiers. L'une des forces de Django est son riche écosystème de paquets tiers. Ce sont des paquets développés par la communauté qui peuvent être utilisés pour enrichir rapidement les fonctionnalités d'une application.

Ce toturiel va vous montrer comment ajouter la barre d'outils de débogage Django Debug Toolbar, un paquet tiers couramment utilisé. Django Debug Toolbar figure dans les trois paquets tiers les plus souvent installés dans les résultats du sondage Django Developers Survey de ces dernières années.

Où obtenir de l'aide :

Si vous rencontrez des problèmes dans le parcours de ce tutoriel, rendez-vous dans la section Obtenir de l'aide de la FAQ.

Installation de Django Debug Toolbar¶

Django Debug Toolbar est un outil bien utile pour déboguer les applications web Django. Il s'agit d'un paquet tiers maintenu par l'organisation Jazzband. Cette barre d'outils aide à comprendre comment une application fonctionne et sert à identifier des problèmes. Elle le fait en fournissant des panneaux présentant des informations de débogage sur la requête et la réponse de la page en cours.

Pour installer une application tierce telle que cette barre d'outils, il est nécessaire d'installer le paquet en exécutant la commande ci-dessous dans un environnement virtuel activé. C'est le même procédé que pour l'installation de Django présentée dans une étape précédente.

```
$ python -m pip install django-debug-toolbar
```

Les paquets tiers qui s'intègrent à Django ont besoin d'une phase de post-installation pour qu'ils s'intègrent dans votre projet. Il s'agit fréquemment d'ajouter l'application du paquet au réglage `INSTALLED_APPS`. Certains paquets ont besoin d'autres modifications, comme des ajouts à la configuration d'URL (`urls.py`).

Django Debug Toolbar nécessite plusieurs étapes de configuration. Suivez-les dans son guide d'installation. Ces étapes ne sont pas reproduites dans ce tutoriel, car faisant partie d'un paquet tiers, elles peuvent être modifiées indépendamment du cycle de Django.

Une fois installée, vous devriez voir apparaître la « poignée » DjDT sur la partie droite de la fenêtre du navigateur dès que vous rafraîchissez la page de l'application de sondage. Cliquez dessus pour ouvrir la barre de débogage et utiliser les outils de chaque panneau. Consultez la page de documentation des panneaux pour plus de détails sur le contenu de chacun des panneaux.

Obtenir de l'aide¶

À un moment ou à un autre, vous rencontrerez des problèmes. Par exemple, la barre pourrait ne pas s'afficher. Lorsque cela arrive et que vous n'êtes pas capable de le résoudre vous-même, vous avez plusieurs possibilités.

1. Si le problème est lié à un paquet en particulier, vérifiez s'il existe un guide de dépannage ou une FAQ dans la documentation du paquet. Par exemple, pour Django Debug Toolbar, il existe une section Tips qui présente des options de dépannage.
2. Recherchez des problèmes semblables dans le système de suivi des tickets du paquet. Pour Django Debug Toolbar, cela se trouve sur GitHub.
3. Consultez le forum de Django.
4. Rejoignez le serveur Discord de Django.
5. Rejoignez le canal IRC #Django sur Libera.chat.

Installation d'autres paquets tiers¶

Il existe bien d'autres paquets tiers que vous pouvez découvrir en utilisant l'excellente ressource Django, Django Packages.

Il peut être difficile de savoir quel paquet tiers utiliser. Cela dépend des vos besoins et de vos objectifs. Dans certains cas, il peut être acceptable d'utiliser un paquet dans un stade alpha. D'autres fois, vous avez besoin d'être sûr qu'il est prêt pour la production. Adam Johnson a écrit un article qui détaille un ensemble de caractéristiques qui permettent de qualifier un paquet comme «bien maintenu». Le site Django Packages affiche des données sur certaines de ces caractéristiques, telles que la date de dernière mise à jour du paquet.

Comme Adam le signale dans son article, lorsque la réponse à l'une des questions est «non», c'est une opportunité de contribuer.

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.