Table des matières

1 Prérequis

1.1 Installations

Après avoir installé Django (par exemple dans un environnement virtuel nommé `django5`), si vous disposez de virtualenvwrapper, activez l'environnement virtuel dans un terminal en ligne de commande avec :

```
workon django5 # suivant où vous avez installé Django
```

Le package **virtualenvwrapper** fournit des raccourcis pour travailler avec **virtualenv**, tels que **workon**, **mkvirtualenv** et autres. Il n'a rien à voir avec Django, mais est couramment utilisé à ses côtés.

source : https://virtualenvwrapper.readthedocs.io/en/latest/command_ref.html#workon

Ne pas utiliser d'environnement virtuel est également possible.

1.2 Commande d'administration

`django-admin` est l'utilitaire en ligne de commande de Django pour les tâches administratives. Le script `django-admin` devrait se trouver dans votre chemin système si vous avez installé Django via pip. S'il ne se trouve pas dans votre chemin, vérifiez que votre environnement virtuel est activé.

Depuis un terminal en ligne de commande, la commande `django-admin` va afficher toutes les commandes disponibles.

2 Création d'un projet

Source : https://docs.djangoproject.com/fr/5.0/intro/tutorial01/

2.1 Exemple

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
```

Nous avons décidé ici d'appeler le projet `mysite` qui est un nom autorisé pour les raisons explicitées plus bas.

2.2 Précautions de nommage

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).

Le nom `mysite` utilisé dans le tutoriel officiel Django est également utilisé ici. C'est un nom autorisé, mais ne donnez pas ce même nom à tous vos projets.

2.3 Fichiers créés

On voit que le nom choisi pour le projet (ici `mysite`) à donné lieu à la création de deux répertoires imbriqués portant ce nom.

— Le premier répertoire racine `mysite/` est un contenant pour votre projet. Son nom n'a pas d'importance pour Django ; vous pouvez, sans conséquences, le renommer comme vous le voulez par la suite.

Ce que contient le répertoire racine :

— `manage.py` : un utilitaire en ligne de commande qui vous permet d'interagir avec ce projet Django de différentes façons. Il fait la même chose que `django-admin` mais définit également la variable d'environnement `DJANGO_SETTINGS_MODULE` pour la faire pointer sur le fichier `settings.py` du projet. Généralement, en travaillant dans un seul projet Django, il est plus simple d'utiliser `manage.py` plutôt que `django-admin`. Si vous devez basculer entre différents fichiers de réglages Django, utilisez la variable `DJANGO_SETTINGS_MODULE` ou l'option de ligne de commande `--settings` de la commande `django-admin`.

— le sous-répertoire `mysite/` correspond au paquet Python effectif de votre projet (le renommer est possible, mais fortement déconseillé car cela implique d'avoir à modifier tous les fichiers du projet où `mysite` est importé, par exemple `manage.py`, `wsgi.py`, `asgi.py` et `settings.py`, et de mettre à jour les imports pour refléter le nouveau nom du répertoire). Donc choisissez bien le nom de votre projet à sa création.

Ce que contient ce sous répertoire :

— `__init__.py` : un fichier vide qui indique à Python que ce répertoire doit être considéré comme un paquet.

— `settings.py` : réglages et configuration de ce projet Django.

— `urls.py` : les déclarations des URL de ce projet Django, une sorte de "table des matières" de votre site Django.

— `asgi.py` : un point d'entrée pour les serveurs Web compatibles aSGI pour déployer votre projet (sans intérêt pour le développement).

— `wsgi.py` : un point d'entrée pour les serveurs Web compatibles WSGI pour déployer votre projet (sans intérêt pour le développement).

Remarque :

```
django-admin startproject mysite .
```

La commande précédente, du fait du point à la fin, permet de créer un projet sans répertoire racine. C'est donc à vous de créer un répertoire au préalable et de vous y rendre avant de créer le projet avec cette commande.

3 Lancer le serveur de développement

3.1 Avertissement

N'utilisez jamais le serveur de développement pour quoi que ce soit qui s'approche d'un environnement de production. Il est fait seulement pour tester votre travail pendant le développement.

3.2 Commande pour lancer le serveur de développement

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

```
python manage.py runserver
```

Starting development server at http://127.0.0.1:8000/ Quit the server with CONTROL-C.

4 Création d'une application

4.1 Différence entre un projet et une application

- Un projet est un ensemble de réglages et d'applications pour un site Web particulier.
- 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 peut contenir plusieurs applications.
- Une application peut apparaître dans plusieurs projets.

4.2 Création d'une application

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 :
Cette structure de répertoire accueillera l'application de sondage.

4.3 Créations des vues

4.3.1 Principe

- Lorsqu'une page est demandée, Django crée un objet `HttpRequest` contenant des métadonnées au sujet de la requête.
- Puis, Django charge la vue appropriée, lui transmettant l'objet `HttpRequest` comme premier paramètre.
- Chaque vue (telle que l'entend Django) est responsable de l'**action** suivante : prendre l'objet `HttpRequest` (et les métadonnées) et renvoyer un objet `HttpResponse`. Et c'est tout !

Remarque : par la suite, nous verrons qu'en pratique, la partie visuelle proprement dite ne relève pas des "vues" ; elle relève des "templates".

4.3.2 Exemple

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

```
from django.http import HttpResponse

# Première vue

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

# Autres vues:

def detail(request, question_id):
    return HttpResponse(f"You're looking at question {question_id}")

def results(request, question_id):
    return HttpResponse(f"You're looking at the results of question {question_id}.")

def vote(request, question_id):
    return HttpResponse(f"You're voting on question {question_id}.")
```

La première vue est la vue la plus basique possible dans Django.

4.4 Lier les vues aux urls

4.4.1 Principe

Pour accéder à une vue dans un navigateur, nous devons la mapper (établir une correspondance avec) à une URL. Pour cela, nous devons définir une configuration d'URL, ou URLconf en abrégé. Ces configurations d'URL sont définies dans chaque application Django au sein de fichiers Python nommés `urls.py`.

4.4.2 Définir une URLconf pour l'application

1. Exemple

   Pour définir une URLconf pour l'application de sondages, créez un fichier `polls/urls.py` avec le contenu suivant :

   ```
   from django.urls import path

   from . import views

   urlpatterns = [
       # ex: /polls/
       path("", views.index, name="index"),
   ]
   ```

2. La fonction `path(route, view, kwargs=None, name=None)`

   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 :

   — Premier 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 le nom de domaine, ni dans les paramètres GET et POST.
   - — 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/`.

   — Deuxième 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 capturée par la route sous forme de paramètres nommés.

   — Troisième paramètre de `path()` : `kwargs`
   - — Comme nous allons le voir prochainement, les configurations d'URL ont un troisième point d'entrée facultatif qui permet de passer des paramètres supplémentaires à vos vues, via un dictionnaire Python.

   — Quatrième 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.

3. Sur l'utilité de `name` dans les gabarits utilisant `url`

   Source : https://stackoverflow.com/a/68307313/5952631

   Une petite partie d'un gabarit `index.html` pouvant être :

```
<a href="{% url 'index'                  %}">index</a>
<a href="{% url 'detail'  question_id=1 %}">detail</a>
<a href="{% url 'results' question_id=1 %}">results</a>
```

   Comme cela est évident, cela montre des liens. Mais, notez ici, à l'intérieur de la balise `<a>`, l'utilisation de la balise `url` de Django.

   Le format correct d'utilisation de la balise url de Django dans les gabarits est : `{% url 'NAME OF URL here' any_variables_here %}`

   `NAME OF URL` signifie le nom que nous donnons à une URL dans l'argument `name` de `path()`, ce qui signifie que nous devons uniquement utiliser le nom de l'URL dans l'attribut `href`, nous n'avons plus besoin d'utiliser l'URL complexe partout dans notre code, c'est une fonctionnalité géniale de Django.

4. Répertoire de l'application

   Votre répertoire d'applications devrait maintenant ressembler à :

4.4.3 Inclure l'URLconf de l'application dans l'URLconf globale du projet

1. Exemple

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

urlpatterns = [
    # permet d'inclure l'URLconf de l'application
    path("polls/", include("polls.urls")),
    path("admin/", admin.site.urls),
]
```

2. La fonction `include()`

   La fonction `include()` permet de référencer et d'inclure 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.

   Exemple :

   (a) Lorsque l'utilisateur visite l'URL `http://monsite.com/polls/`, Django cherche une correspondance dans le fichier `urls.py` principal. La partie de l'URL `polls/` correspond à `path('polls/', include('polls.urls'))`. Django tronque cette partie (c'est-à-dire qu'il la supprime) et passe le reste de l'URL (`""` ici, car il n'y a rien après `polls/`) au fichier `urls.py` de l'application `polls`.

   (b) Django continue maintenant à chercher une correspondance dans `polls/urls.py`. Ici, `path(", views.index, name='index')` correspond, car l'URL restante est vide (`""`). La vue `views.index` est appelée.

   (c) Si l'utilisateur visite `http://monsite.com/polls/5/vote/`, Django tronque `polls/` et passe le reste de l'URL (`5/vote/`) à `polls/urls.py`. Ici, `path(<int:question_id>/vote/, views.vote, name='vote')` correspond, et la vue `views.vote` est appelée avec `question_id=5`.

   L'utilisation de `include()` permet donc à Django de structurer les URL de manière hiérarchique et modulaire, facilitant la maintenance et l'extension du projet. 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.

3. Quand utiliser include()

   Alors que `admin.site.urls` utilise systématiquement `path()` et qu'on pourrait ajouter d'autre urls dans l'URLconf globale grâce à `path()`, il est plutôt conseillé d'utiliser `include()` lorsque l'on veut inclure d'autres motifs d'URL. Ces motifs d'urls seront alors placés dans les URLConf des applications où l'on fera usage de `path()`.

Cela permet de structurer les URL de manière modulaire, en les séparant en plusieurs fichiers de configuration.

5 Configuration de la base de données

Source : https://docs.djangoproject.com/fr/5.0/intro/tutorial02/

5.1 Choix de la base de données

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. Pour le stockage local des données avec une faible concurrence de l'écriture et moins d'un téraoctet de contenu, SQLite est la meilleure solution. SQLite est rapide et fiable et ne nécessite aucune configuration ou maintenance. SQLite est inclus dans Python, vous n'aurez donc rien d'autre à installer pour utiliser ce type de base de données.

SQLite fonctionne très bien comme moteur de base de données pour la plupart des sites Web à faible à moyen trafic (c'est-à-dire la plupart des sites Web). La quantité de trafic Web que SQLite peut gérer dépend de l'utilisation par le site Web de sa base de données. D'une manière générale, tout site qui obtient moins de 100 000 visites par jour devrait bien fonctionner avec SQLite.

S'il existe de nombreux programmes clients qui envoient SQL à la même base de données sur un réseau, utilisez un moteur de base de données client/serveur au lieu de SQLite.

5.2 Création des tables associées

La commande `migrate` examine le réglage `INSTALLED_APPS` dans votre fichier `mysite/settings.py` et crée les tables de base de données nécessaires en fonction des réglages de base de données 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.

5.3 Examen des tables de votre base de données

Pour afficher les tables créées par Django, si cela vous intéresse, lancez le client en ligne de commande de votre base de données. Par exemple pour SQLite :

```
sqlite3 db.sqlite3
```

Puis, tapez l'une des commandes suivantes :
— `\dt` (PostgreSQL),
— `SHOW TABLES;` (MariaDB, MySQL),
— `.tables` (SQLite)
— `SELECT TABLE_NAME FROM USER_TABLES;` (Oracle)

5.4 Création des modèles

5.4.1 Philosophie des modèles

Dans la programmation orientée objet, un champ (ou attribut) est une propriété d'un objet. Cette propriété a un nom, un type de données et une valeur.

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.

— **Un modèle équivaut à une table SQL et chaque modèle est représenté par une classe Python** (qui hérite de `django.db.models.Model`).
— **Chaque champ correspond à une colonne dans une table SQL et chaque champ est représenté par un attribut de la classe Python correspondant au modèle.**

5.4.2 Modèles dans notre application de sondage

Nous allons créer deux modèles : `Question` et `Choice` (choix).

— Une Question possède deux champs : un énoncé de type `CharField` et une date de mise en ligne de type `DateTimeField`.
— 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 dont les attributs correspondent aux champs des modèles.

Éditez le fichier `polls/models.py` de façon à ce qu'il ressemble à ceci :

```
from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)    # énoncé de la question
    pub_date = models.DateTimeField("date published")    # date de publication


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    # chaque vote (Choice) n'est relié qu'à une seule Question
    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 attributs (variables de classe). Chaque attribut représentant un champ de la base de données pour ce modèle.

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.

Chaque autre attribut est un champ 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.

Liste des champs proposés pas Django : https://docs.djangoproject.com/fr/5.0/ref/models/fields/#field-types

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.

6 Inclure l'application au projet

6.1 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.

6.2 Étape 1 : créer ou modifier les modèles

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 :

```
INSTALLED_APPS = [
    "polls.apps.PollsConfig", # ajout d'une référence
    "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.

6.3 Étape 2 : créer des migrations correspondant à ces changements

Exécutons une autre commande :

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

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

Vous pouvez consulter la migration pour vos nouveaux modèles si vous le voulez ; il s'agit du fichier `polls/migrations/0001_initial.py`. Vous n'êtes pas censé les lire chaque fois que Django en crée, mais ils sont conçus pour être humainement lisibles.

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.

Si cela vous intéresse, vous pouvez 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.

6.4 Étape 3 : appliquer ces modifications 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
```

— Des clés primaires (ID) sont ajoutées automatiquement.
— Django ajoute `_id` au nom de champ des clés étrangères.

7 L'interface de programmation (API)

Maintenant, utilisons un **shell interactif Python** pour bénéficier de l'API que Django met gratuitement à notre disposition.

Pour lancer un shell Python, utilisez cette commande :

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

7.1 Importer les modèles d'une application

```
# Importe les modèles de l'application polls
>>> from polls.models import Choice, Question
```

7.2 Lister tous les objets (instances d'une classe de modèle)

Pour lister tous les enregistrements :

```
>>> Question.objects.all()
```

```
>>> Choice.objects.all()
```

7.3 Créer de nouveaux objets

7.3.1 Créer une nouvelle question

Un objet `question` a deux attributs : `question_text` et `pub_date`

La prise en charge des fuseaux horaires (timezone) est activée dans le fichier de paramètres par défaut, donc Django attend :

```
>>> from django.utils import timezone
>>> q = Question(question_text="Quoi de neuf ?",
                 pub_date=timezone.now())
```

7.3.2 Sauvegarder une question dans la base de données

Vous devez appeler `save()` explicitement.

```
>>> q.save()
```

Maintenant la question à un ID.

```
>>> q.id
```

7.3.3 Ajouter des méthodes `__str__()` aux modèles

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.

```
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
```

Dorénavant, lister les objets transmettra une liste des textes qui avaient été saisis.

```
>>> Question.objects.all()
```

7.3.4 Ajouter une méthode à un modèle

```
import datetime

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


class Question(models.Model):
    # ...
    def was_published_recently(self): # publié il y a moins d'un jour
        return timezone.now() - datetime.timedelta(days=1) <= self.pub_date
```

7.3.5 Afficher les choix associés à une question

Sélectionnez une question :

```
>>> q = Question.objects.get(pk=1)
```

Afficher tous les choix associés à cette question donc aucun jusqu'à présent :

```
>>> q.choice_set.all()
```

7.3.6 Créer de nouveaux choix associés à une question

Donnons quelques choix à cette question.

L'appel `create` construit un nouvel objet `Choice`, exécute l'instruction `INSERT`, ajoute le choix à l'ensemble de choix disponibles et renvoie le nouvel objet de type `Choice`.

Django crée un ensemble (défini comme `choice_set`) pour contenir "l'autre côté" d'une relation clé étrangère (par exemple le choix d'une question) accessible via l'API.

Créez trois choix :

```
>>> q.choice_set.create(choice_text="Not much", votes=0)

>>> q.choice_set.create(choice_text="The sky", votes=0)

>>> c = q.choice_set.create(choice_text="Just hacking again", votes=0)
```

7.3.7 Relations

Les objets de type Choice ont un accès API à l'objet de type Question qui leur est associé.

```
>>> c.question
```

Et vice versa, les objets de type `Question` ont accès aux objets de type `Choice` :

```
>>> q.choice_set.all()

>>> q.choice_set.count()
```

7.4 Filtrer

7.4.1 Filtrer les enregistrements par valeur de champ

```
>>> Question.objects.filter(id=1)

>>> Question.objects.filter(question_text__startswith="Quoi")
```

La méthode `filter()` renvoie un objet queryset.

Si vous utilisez `filter()`, vous le faites généralement chaque fois que vous attendez plus d'un objet correspondant à vos critères.

Si aucun élément ne correspond à vos critères, `filter()` renvoie un ensemble de requêtes vide sans générer d'erreur.

7.4.2 Filtrer les choix

L'API suit automatiquement les relations autant que vous en avez besoin. Utilisez des traits de soulignement doubles pour séparer les relations. Cela fonctionne à autant de niveaux que vous le souhaitez ; il n'y a pas de limite.

Trouvez tous les choix pour toute question dont la date_{pub} est cette année (en réutilisant la variable `current_year` que nous avons créée ci-dessus) :

```
>>> Choice.objects.filter(question__pub_date__year=current_year)
```

7.4.3 Filtrer puis supprimer

Supprimons l'un des choix. Utilisez `delete()` pour cela :

```
>>> c = q.choice_set.filter(choice_text__startswith="Just hacking")
>>> c.delete()
```

7.5 Obtenir un objet

Si vous utilisez `get()`, vous attendez un (et un seul) élément correspondant à vos critères.

```
# Obtenez la question qui a été publiée cette année.
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
```

La méthode `get()` renvoie une erreur si l'élément n'existe pas ou s'il existe plusieurs éléments correspondant à vos critères.

```
>>> Question.objects.get(id=2)
```

Vous devez donc toujours utiliser `if` dans un bloc `try.. except ..` ou avec une fonction de raccourci comme `get_object_or_404` afin de gérer correctement les exceptions.

7.6 Raccourci pour obtenir un objet par clé primaire

La recherche par clé primaire est le cas le plus courant, donc Django fournit un raccourci pour les recherches exactes par clé primaire.

Ce qui suit est identique à `Question.objects.get(id=1)`.

```
>>> Question.objects.get(pk=1)
```

Assurez-vous que notre méthode personnalisée a fonctionné :

```
>>> q = Question.objects.get(pk=1)
>>> q.was_published_recently()
```

8 Introduction au site d'administration de Django

8.1 Philosophie

L'interface d'administration n'est pas destinée à être utilisée par les visiteurs du site ; elle est conçue pour les administrateurs c'est-à-dire les gestionnaires du site qui éditent le contenu pour ajouter des nouvelles, des histoires, des événements, des résultats sportifs, etc.

Django procure une interface uniforme pour les administrateurs du site.

Django a été écrit dans un environnement éditorial, avec une très nette séparation entre les **éditeurs de contenu** et le site **public** publié à destination des visiteurs.

8.2 Création d'un utilisateur administrateur

Pour créer un compte administrateur, utilisez la commande suivante :

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

8.3 Démarrage du serveur de développement

Le site d'administration de Django est activé par défaut.

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 ń *admin* ż 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` dans `settings.py`, l'écran de connexion s'affiche dans cette langue (pour autant que les traductions correspondantes existent dans Django).

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

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 :

```
from django.contrib import admin
from .models import Question
admin.site.register(Question)
```

Maintenant les sondages apparaissent sur la Page d'accueil du site d'administration de Django.

Cliquez sur la question ń Quoi de neuf ? ż pour la modifier à travers le formulaire d'édition de l'objet question lequel 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.

Si la valeur de `Date de publication` ne correspond pas à l'heure à laquelle vous avez créé cette question 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.

Si vous 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.

9 Création de l'interface publique (les vues)

9.1 Aperçu

— 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 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.

Un modèle (motif) 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.

9.2 Écriture de vues supplémentaires

```
from django.http import HttpResponse

# Première vue
```

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

# Autres vues:

def detail(request, question_id):
    return HttpResponse(f"You're looking at question {question_id}")

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

def vote(request, question_id):
    return HttpResponse(f"You're voting on question {question_id}.")
```

Nous avions parlé de la première vue qui est la vue la plus basique possible dans Django. Les autres vues acceptent un paramètre.

9.3 Transmission de paramètres aux vues

```python
from django.urls import path

from . import views

urlpatterns = [
    # ex: /polls/
    path("", views.index, name="index"), # ou views.index appelle la fonction index(request)
                                         # définie dans le fichier views.py de l'application
    # 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 fonction `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.

9.4 Écriture de vues qui font réellement des choses

Source : https://docs.djangoproject.com/fr/5.0/intro/tutorial03/

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.

9.4.1 Codage en dur sans gabarit

L'allure de la page est codée en dur dans la vue.

```
def latestQuestions(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)
```

9.4.2 Codage avec gabarits

Le système de gabarits de Django permet de 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`.

Insérez le code suivant dans ce gabarit :

```
{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        Codé de la mauvaise manière:
        <li><a href="/polls/{{ question.id }}/">
            {{ question.question_text }}</a></li>
        Codé de la bonne manière en utilisant app_name et le
        paramètre name dans les fonctions path():
        <li><a href="{% url 'polls:detail' question.id %}">
            {{ question.question_text }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

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

```
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 des liens pointant vers la page de détail de la question sélectionnée.

9.4.3 Codage propre avec gabarits

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 pour cela : `render()`

Voici les vues `index()` et `detail()` réécrites avec `render()` :

```
from django.http import HttpResponse
from django.template import loader
from django.shortcuts import get_object_or_404, render
```

```python
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)

def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    context = {"question": question}
    return render(request, "polls/detail.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.
- 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.
- 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.

9.5 Django et les espaces de noms (namespaces)

9.5.1 Philosophie

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.

Django permet de structurer les URLs de manière hiérarchique à l'aide d'espaces de noms (`namespaces`). Ceci est particulièrement utile lorsqu'une application Django inclut plusieurs modules (ou applications) qui peuvent avoir des vues ayant le même nom, mais qui doivent être distinguées les unes des autres.

9.5.2 Exemple

Supposons que vous ayez deux applications dans votre projet Django, appelées `polls` et `blog`, et que chacune d'elles ait une vue appelée `vote`. Sans espaces de noms, Django ne saurait pas quelle vue `vote` appeler. En ajoutant un espace de noms, vous pouvez les distinguer :

- `polls:vote` fait référence à la vue `vote` dans l'application `polls`.
- `blog:vote` fait référence à la vue `vote` dans l'application `blog`.

Ceci est utilisé dans l'écriture `{% url 'polls:vote' question.id %}` qui permet à Django de comprendre qu'il doit rechercher une vue nommée `vote` dans l'application ou l'espace de noms `polls`, et de construire l'URL correspondante en utilisant l'identifiant de la question (`question.id`) passé en paramètre.

9.5.3 Conclusion

Le : est donc essentiel pour spécifier l'espace de noms dans Django et pour différencier les vues lorsque vous avez des noms de vues similaires dans différentes applications.

10 Écriture de formulaires

Source : https://docs.djangoproject.com/fr/5.0/intro/tutorial04/

10.1 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> :

```
<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>
```

10.2 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 %}`.

11 Affichage des résultats

Reprenons les principes précédents pour afficher les résultats.

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

11.1 Réécrivons la vue `results()` dans `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})
```

Cette vue requiert le gabarit `polls/results.html`.

11.2 Écrivons maintenant le gabarit `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>
```

12 Utilisation des vues génériques

Les vues `index()`, `detail()` et `results()` sont très courtes et 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 ajoutent une couche d'abstraction pour les procédés courants au point où vous n'avez même plus besoin d'écrire du code Python pour écrire une application.

Les deux vues suivantes sont **les vues génériques d'affichage** ; elles sont conçues pour afficher des données. Pour beaucoup de projets, il s'agit habituellement des vues les plus fréquemment utilisées :

— la vue générique `ListView` implémente le concept d'**afficher une liste d'objets** ;

— la vue générique `DetailView` implémente celui d'**afficher une page détaillée pour un type particulier d'objet**.

Elles sont fondées sur des classes pas sur des fonctions.

Nous allons convertir notre application de sondage pour qu'elle utilise le système de vues génériques.

12.1 Écriture de l'URLconf

La manière la plus directe d'utiliser des vues génériques est de les créer directement dans votre configuration d'URL. Si vous ne devez changer qu'un nombre restreint d'attributs d'une vue fondée sur une classe, vous pouvez les transmettre directement dans l'appel de méthode `as_view()`. Tout paramètre transmis à `as_view()` surcharge l'attribut de même nom de la classe.

Ici, nous aurions pu simplement transmettre le paramètre `template_name = "polls/detail.html"` dans l'appel `as_view()` en écrivant `path("<int:pk>/", views.DetailView.as_view(template_name = "polls/detail.html"), name="detail")` et nous aurions pu faire de même pour `template_name = "polls/results.html"` mais nous n'allons pas le faire car nous allons plutôt utiliser l'héritage des vues génériques dans l'écriture des vues, juste après.

Ouvrez la configuration d'URL `polls/urls.py` et modifiez-la ainsi :

```
from django.urls import path

from . import views

app_name = "polls"
urlpatterns = [
    # recours aux vues génériques
    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"),
    # recours à une vue classique
    path("<int:question_id>/vote/", views.vote, name="vote"),
]
```

Notez que le nom des motifs de correspondance dans les chaînes de chemin des deuxième et troisième motifs ont changé de `<question_id>` à `<pk>`. Ceci est nécessaire car la vue générique `DetailView` sera utilisée pour remplacer les vues `detail()` et `results()`, et que cette vue s'attend à ce que la valeur de clé primaire capturée dans l'URL soit nommée "pk".

Pourquoi utiliser `as_view()` dans ce cas puisqu'à ce stade nous ne modifions aucun attribut ?

Dans les vues basées sur les classes, **vous devez appeler la fonction `as_view()` afin de renvoyer une vue qui prend une requête et renvoie une réponse**. C'est le principal point d'entrée dans le cycle requête-réponse en cas de vues génériques. `as_view` est la fonction (méthode de classe) qui connectera votre classe `MyView` (ici `IndexView`, `DetailView` et `ResultsView`) à son URL.

12.2 Écriture des vues

Plutôt que de transmettre les paramètres directement dans l'appel de méthode `as_view()` , l'autre façon de faire, plus puissante, d'utiliser les vues génériques est d'hériter d'une vue existante et de surcharger ses attributs (comme `template_name`) ou ses méthodes (comme `get_queryset`) dans votre sous-classe pour fournir d'autres valeurs ou méthodes.

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 :

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

Chaque vue générique doit connaître le modèle sur lequel elle agira.
Pour cela, on utilise :
— soit l'attribut `model` (dans cet exemple, `model = Question` pour `DetailView` et `ResultsView`),
— soit on définit la méthode `get_queryset()` (tel qu'illustré pour la vue `IndexView`).

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.

13 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. Avant de l'utiliser dans `tests.py` commencerons par voir son emploi dans 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()
```

Ensuite, il est nécessaire d'importer la classe Client de test :

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

Nous pouvons maintenant simuler l'interaction d'un utilisateur avec le code au niveau des vues.

```
>>> # 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
>>> response.context["latest_question_list"]
<QuerySet [<Question: What's up?>]>
```

14 Introduction aux tests automatisés

Source : https://docs.djangoproject.com/fr/5.0/intro/tutorial05/

14.1 Que sont les tests automatisés ?

Les tests sont des routines qui vérifient le fonctionnement de votre code. Dans les tests automatisés, le travail de test est fait pour vous par le système.

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 ?).

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.

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.

14.2 Création d'un test de modèle

Nous avions ajouté la méthode `was_published_recently()` au modèle `Question`. Elle est censée retourner `True` si la question a été publiée depuis moins d'un jour et `False` dans le cas contraire.

```
class Question(models.Model):
    # ...
    def was_published_recently(self): # publié il y a moins d'un jour
        return timezone.now() - datetime.timedelta(days=1) <= self.pub_date
```

Le problème est que `was_published_recently()` renvoie `True` pour les questions dont `pub_date` est dans le futur.

```
time = timezone.now() + datetime.timedelta(days=30)
future_question = Question(pub_date=time)
```

Créons un test qui permet de révéler ce bogue.

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

```
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)
```

14.3 Lancement des tests

Dans le terminal, nous pouvons lancer notre test :

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

Ce qui va se passer :

- La commande `manage.py test polls` va chercher des tests dans l'application `polls` (les méthodes de test sont celles dont le nom commence par test) ;
- Une base de données est spécialement crée pour les tests ;
- Dans `test_was_published_recently_with_future_question`, une instance `Question` dont le champ `pub_date` est 30 jours dans le futur est crée ;
- À l'aide de la méthode `assertIs()`, le test va révéler que la méthode `was_published_recently()` renvoie `True`, alors que nous souhaitons qu'elle renvoie `False`.
- Le test nous indique le nom du test qui a échoué ainsi que la ligne de l'assertion qui a échoué.

```
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_qu
    self.assertIs(future_question.was_published_recently(), False)
AssertionError: True is not False
```

14.4 Amélioration du modèle

Corrigez la méthode dans `models.py`

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

Relancer les test ne révélera plus aucun problème.

```
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'...
```

14.5 Des tests de modèle plus exhaustifs

Pendant que nous y sommes, ajoutons deux tests ainsi nous disposerons de trois tests qui permettent de confirmer que `Question.was_published_recently()` renvoie des valeurs correctes pour des questions :

— passées (publiées depuis plus d'un jour),
— récentes (publiées depuis moins d'un jour)
— et futures (publiées après maintenant).

```
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)
```

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.

```
Found 3 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...
----------------------------------------------------------------------
```

```
Ran 3 tests in 0.001s

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

14.6 Amélioration de la vue

L'application de sondage publiera toute les questions, y compris celles dont le champ `pub_date` est situé dans le futur. Cela 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.

Nous avions introduit la vue IndexView basée sur la classe ListView :

```
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 :

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

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

```
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`.

14.7 Test de la nouvelle vue

Vous pourriez maintenant vérifier par vous-même que tout fonctionne comme prévu en lançant `runserver` et en accédant au site depuis votre navigateur. Il faudrait créer des questions avec des dates dans le passé et dans le futur et vérifier que seules 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 automatisé pour tester la vue.

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

```
from django.urls import reverse
```

Ce nom de fonction (`reverse`) semble particulièrement ni évocateur ni intuitif, ce que l'on peut considérer comme une faute grave.

Nous avons vu, qu'étant donné un modèle d'URL, Django utilise la fonction `path()` pour choisir la bonne vue et générer une page. Autrement dit, *path vue et nom de la vue*. Mais parfois, comme lors d'une redirection, vous devez aller dans le sens inverse. C'est-à-dire, donner à Django le nom d'une vue, et attendre de Django qu'il génère l'URL appropriée. En d'autres termes, *nom de la vue path*.

Autrement dit `reverse()` fait l'inverse de la fonction `path()`. Il s'agit d'un exemple typique de dénomination qui met l'accent sur un aspect d'une entité (par exemple une fonction) qui était au premier plan dans l'esprit du programmeur à l'époque, compte tenu de son contexte, mais qui n'est pas la dénomination la plus utile dans le contexte plus large de tout autre développeur. Nous tombons souvent dans ce piège en tant que programmeurs : la dénomination des entités est si importante pour leur découvrabilité qu'elle vaut la peine de s'y arrêter et de réfléchir aux différents contextes et de choisir celui qui est le plus approprié pour que les autres programmeurs le trouvent transparent.

Il aurait sans doute été plus transparent ici de nommer cette fonction `pathFromViewName` ou `urlFromViewName` plutôt que `reverse` ; mais cela n'a pas été le cas.

Ajoutez ce qui suit à `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],
        )
```