TP3 : Python, modèles

<2019-02-27 mer.>

Table des matières

1 Les modèles **1**
- 1.1 Spécifications . . . 1
- 1.2 Ressources . . . 3

2 Remplir la base **4**
- 2.1 Migrations . . . 4
- 2.2 Stockage des données . . . 4
- 2.3 Quelques tests . . . 4

3 Servir un fichier contenant les infos des cours **4**

4 Mise en application : de la base à l'affichage **5**

5 Création de méthodes Python utiles **6**
- 5.1 Redéfinition de la méthode d'impression . . . 6
- 5.2 Arbre des groupes . . . 6
- 5.3 Plein d'autres . . . 6

6 Explorer le code de FlOpEDT **6**

1 Les modèles

1.1 Spécifications

Voici une description des modèles utilisés dans `base` et dans `people`.

Classes	Attributs	Propriétés/Description
Group	name	son nom
	parent_group	son surgroupe direct
RoomType	name	les types de salle (TD, TP, Amphi...)
Room	name	les salles, qui chacune peut avoir plusieurs
	room_type	types (la B007 est une salle de TD et de TP)
Module	name	Son nom
	abbrev	Son nom abrégé (qui doit être limité à 10 caractères)
	head	Le responsable de modules
Course	group	
	tutor	
	module	
	room_type	
	week	Un nombre entre 0 et 53
	year	
	duration	La durée du cours, en minutes
ScheduledCourse	course	Le cours à placer...
	day	models.CharField(max_length=2, default=Day.MONDAY, choices=[(d, d.value) for d in Day])
	start_time	En minutes à partir de minuit
	room	
ModuleDisplay	module	C'est du OneToOne !
	color_bg	
	color_txt	
GroupDisplay	group	
	button_height	
	button_txt	
Tutor		Hérite de la classe User de django.contrib.auth.models
FullStaff		Hérite de Tutor
SupplyStaff		Hérite de Tutor
	employer	Son affichage devrait mentionner son employeur...
	position	
Student	belong_to	Les groupes auxquels il appartient

Voici, par exemple, comment définir la classe `Group` dans le fichier `base/models.py` :

```
class Group(models.Model):
    name = models.CharField(max_length=4)
    parent_group = models.ForeignKey('self',
```

```
                                    blank=True,
                                    null=True,
                                    related_name="children_group",
                                    on_delete=models.CASCADE)

    #On va surcharger la fonction __str__ afin d'avoir un affichage plus
    #joli en cas de print
    def __str__(self):
        return self.name
```

Les arguments de la `Foreignkey` nous disent :

— `parent_group` contiendra un objet du type de `self`, autrement dit un `Group`. Ce cas est un peu particulier, parce qu'il fait référence à un objet de la classe qu'on est en train de définir ; en général, vous inclurez plutôt un autre nom de classe.

— `blank` est un argument utilisé par les formulaires, par exemple ceux de l'interface d'administration. `blank=True` indique que l'utilisateur pourra laisser le champ vide.

— `null` concerne l'attribut qu'on est en train de définir. `null=True` autorise l'attribut à être `null` (ou `None`) en python.

— `related_name` est moins important et est utile pour la liaison inverse. On est en train de définir le lien d'un `Group`, disons `g`, vers son parent. On accédera à `g.parent_group` pour obtenir son groupe parent. Étant donné un groupe `g`, `g.related_name` indique comment accéder à tous les groupes où `g` apparaît en tant que `parent_group`.

— le dernier attribut dit à Django quoi faire si un groupe qui apparaît en tant que `parent_group` est supprimé : que faire des relations qui vont être altérées. `on_delete=models.CASCADE` indique : Django, tu supprimeras toutes les relations concernées si ça arrive.

> En vous en inspirant, vous pouvez définir les autres classes. Voici les classes de models qui pourraient vous être utiles : CharField, ForeignKey, ManyToManyField, PositiveSmallIntegerField, OneToOneField,

On définira également la classe `Day` qui héritera de la classe `Enum` (pour pouvoir faire référence aux jours de la semaine) :

```
class Day(Enum):
    MONDAY = "m"
    TUESDAY = "tu"
    WEDNESDAY = "w"
    THURSDAY = "th"
    FRIDAY = "f"
    SATURDAY = "sa"
    SUNDAY = "su"
```

1.2 Ressources

> Récupérer le fichier `admin.py` et le copier dans `base`. Exécuter un `runserver` ; s'il ne se plaint pas, félicitations ! S'il se plaint, à vrai dire, c'est attendu, le contraire eût été étonnant.
>
> Lire les erreurs et corriger le `base/models.py` en fonction.

Lorsqu'il n'y a plus d'erreur, cela ne veut pas forcément dire que le `models.py` est bon. Mais en tout cas, on n'a pas de preuve évidente qu'il ne l'est pas.

2 Remplir la base

2.1 Migrations

Le `runserver` fait des vérifications au niveau des classes python : il vérifie que les classes existent bien dans `models.py` et qu'elles comportent bien les bons attributs. Il ne vérifie pas l'adéquation des modèles Python avec la base de données qui est censée stocker ces objets.

Tant que l'on ne dit pas explicitement à PostgreSQL de faire évoluer la structure de la base de données, celle-ci ne prend pas en compte les changements de `models.py`.

> Faire évoluer la base en construisant les fichiers de migration, puis en appliquant les migrations.

2.2 Stockage des données

> — Lancer un shell django `./manage.py shell`.
> — Exécuter le script de remplissage de la base de données `import deploy_base`.
> — Là encore, s'il se plaint, lire les erreurs et corriger le `base/models.py` en fonction.

2.3 Quelques tests

Une fois corrigé, dans le shell django vous pouvez essayer quelques requêtes pour vérifier que les données ont bien été stockées.

> — Afficher dans la console tous les cours du professeur `PSE`.
> — Afficher tous les coursplacé du module `FLOP`.
> — Ajouter un `Tutor`
> Plus dur :
> — Afficher dans la console tous les cours placés qui commencent à 8h le lundi ou à 11h le mardi.

3 Servir un fichier contenant les infos des cours

Le fichier `base/admin.py` nous a notamment servi à vérifier l'adéquation du `base/models.py` avec ses spécifications. Il va désormais surtout être utile pour faire passer les informations de cours depuis la base de données vers le javascript.

Le javascript a besoin d'un fichier CSV ; jusqu'à présent, nous utilisions les fichiers statiques `base/static/base/dataXXXX.csv`. Ces fichiers ont été créés une fois pour toute, et ne permettent pas de refléter l'état d'une base de données qui peut évoluer.

> Utiliser la `resources.ModelResource` fournie dans `base/admin.py` pour créer des fichiers CSV à la volée, dans la vue qui renvoie les données.

```
from base.admin import ScheduledCourseResource
from base.models import ScheduledCourse

def fetch_scheduled_courses(req, year=None, week=None):
    if year is None or week is None:
        year = 2018
        week = 10
```

```
try:
    year = int(year)
    week = int(week)
except ValueError:
    return HttpResponse("KO")

dataset = None

dataset = ScheduledCourseResource() \
    .export(ScheduledCourse.objects.filter(
                course__week=week,
                course__year=year)
    )

if dataset is None:
    raise Http404("What are you trying to do?")

response = HttpResponse(dataset.csv, content_type='text/csv')
response['week'] = week
response['year'] = year

return response
```

4 Mise en application : de la base à l'affichage

En s'inspirant de ce qu'on a vu ci-dessus, on va créer une page qui affiche les adresses e-mail des enseignants.

Pour cela :

- créer un objet de la classe `resources.ModelResource`, appelé `TutorResource`, que vous mettrez dans `people/admin.py` en vue de créer un .csv contenant le username et l'adresse e-mail de tous les enseignants.
- dans `people/views.py`, créer une fonction `fetch_tutors(req)` qui renvoie un fichier `dataset.csv` contenant les informations voulues (à l'aide de la fonction `HttpResponse`)
- ajouter un chemin `fetch_tutors/` dans `people/urls.py`, permettant d'appeler la fonction ci-dessus.
- ajouter un chemin `people/` dans `urls.py` permettant d'accéder aux urls de `people`.

L'accès à l'url choisie vous renverra donc un csv contenant les informations pertinentes. Reste alors à l'afficher, pour cela :

- créer un template dans `people/templates/people` qui utilisera le fichier `dataset.csv` comme une variable et l'affichera `joliment` (à vous de voir à quel point !).
- créer une fonction dans `people/views.py` qui renvoie ce template.
- créer un chemin dans `people/urls.py` qui appelle cette fonction.

5 Création de méthodes Python utiles

5.1 Redéfinition de la méthode d'impression

Regarder ce que renvoie la méthode `print` pour les objets que vous avez créés.

Modifiez la méthode `__str__` pour que ça imprime quelque chose de lisible dans la console.

5.2 Arbre des groupes

Ajouter à la classe `Group` une méthode `ancestor_groups` qui renvoie l'ensemble de tous les sur-groupes du groupe en question.

5.3 Plein d'autres

Que vous définirez...

6 Explorer le code de FlOpEDT

S'il vous reste du temps, vous pouvez cloner le dépôt du logiciel

`https://framagit.org/flopedt/FlOpEDT.git`

et l'explorer... En particulier, commencer à identifier les fichiers qu'il va vous falloir modifier pour votre projet