APPRENEZ

Django

eBook gratuit non affilié créé à partir des
contributeurs de Stack Overflow.

#django

Table des matières

À propos 1

Chapitre 1: Démarrer avec Django 2

Remarques 2

Versions 2

Examples 3

Lancer un projet 3

Concepts Django 5

Un exemple complet de bonjour au monde. 6

Environnement virtuel 7

Python 3.3+ 7

Python 2 7

Activer (n'importe quelle version) 8

Sinon, utilisez virtualenvwrapper 8

Alternativement: utiliser pyenv + pyenv-viritualenv 9

Définissez le chemin de votre projet 9

Exemple de fichier unique Hello World 10

Projet compatible avec le déploiement avec support Docker. 10

Structure du projet 11

Dockerfile 11

Composer 11

Nginx 12

Usage 13

Chapitre 2: Administration 14

Examples 14

Changer de liste 14

Styles CSS et scripts JS supplémentaires pour la page d'administration 15

Traiter des clés étrangères référençant de grandes tables 16

views.py 17

urls.py 17

forms.py 17

admin.py 18

Chapitre 3: Agrégations de modèles 19

Introduction 19

Examples 19

Moyenne, Minimum, Maximum, Somme de Queryset 19

Compter le nombre de relations étrangères 19

GROUB BY ... COUNT / SUM Django équivalent ORM 20

Chapitre 4: ArrayField - un champ spécifique à PostgreSQL 22

Syntaxe 22

Remarques 22

Examples 22

Un ArrayField de base 22

Spécification de la taille maximale d'un ArrayField 22

Demander l'appartenance à ArrayField avec contains 23

Nid ArrayFields 23

Interrogation pour tous les modèles qui contiennent des éléments dans une liste avec conta 23

Chapitre 5: Backends d'authentification 24

Examples 24

Backend d'authentification d'e-mail 24

Chapitre 6: Balises de modèle et filtres 25

Examples 25

Filtres personnalisés 25

Tags simples 25

Balises personnalisées avancées utilisant Node 26

Chapitre 7: Céleri en cours d'exécution avec superviseur 29

Examples 29

Configuration du céleri 29

CÉLERI 29

Superviseur en cours d'exécution 30

Céleri + RabbitMQ avec Superviseur 31

Chapitre 8: Commandes de gestion 33

Introduction 33

Remarques 33

Examples 33

Création et exécution d'une commande de gestion 33

Obtenir la liste des commandes existantes 34

Utiliser django-admin au lieu de manage.py 35

Commandes de gestion intégrée 35

Chapitre 9: Comment réinitialiser les migrations django 37

Introduction 37

Examples 37

Réinitialisation de la migration Django: suppression de la base de données existante et mi 37

Chapitre 10: Comment utiliser Django avec Cookiecutter? 38

Examples 38

Installer et configurer le projet django en utilisant Cookiecutter 38

Chapitre 11: Configuration de la base de données 40

Examples 40

MySQL / MariaDB 40

PostgreSQL 41

sqlite 42

Agencements 42

Moteur Django Cassandra 44

Chapitre 12: CRUD à Django 45

Examples 45

** Exemple CRUD le plus simple ** 45

Chapitre 13: Déploiement 50

Examples 50

Lancer l'application Django avec Gunicorn 50

Déploiement avec Heroku 50

Déploiement à distance simple fabfile.py 51

Utilisation du modèle de démarrage Heroku Django. 52

Instructions de déploiement Django. Nginx + Gunicorn + Superviseur sur Linux (Ubuntu) 52

NGINX 53

GUNICORN 54
SUPERVISEUR 55
Déployer localement sans configurer apache / nginx 55
Chapitre 14: Des modèles 57
Introduction 57
Examples 57
Créer votre premier modèle 57
Appliquer les modifications à la base de données (Migrations) 57
Créer un modèle avec des relations 59
Requêtes de base sur la base de données Django 60
Une table non gérée de base. 61
Modèles avancés 62
Clé primaire automatique 62
URL absolue 62
Représentation de chaîne 63
Terrain de limaces 63
La classe Meta 63
Valeurs calculées 63
Ajout d'une représentation sous forme de chaîne d'un modèle 64
Model mixins 65
Clé primaire UUID 66
Héritage 67
Chapitre 15: Des vues 68
Introduction 68
Examples 68
[Introduction] Simple View (Hello World Equivalent) 68
Chapitre 16: Django à partir de la ligne de commande. 69
Remarques 69
Examples 69
Django à partir de la ligne de commande. 69
Chapitre 17: Django et les réseaux sociaux 70
Paramètres 70

Examples 71

Moyen facile: python-social-auth 71

Utiliser Django Allauth 74

Chapitre 18: Django Rest Framework 77

Examples 77

API simple en lecture seule 77

Chapitre 19: django-filter 79

Examples 79

Utilisez django-filter avec CBV 79

Chapitre 20: Enregistrement 80

Examples 80

Connexion au service Syslog 80

Configuration de base de Django 81

Chapitre 21: Expressions F () 83

Introduction 83

Syntaxe 83

Examples 83

Éviter les conditions de course 83

Mise à jour du jeu de requête en vrac 83

Exécuter des opérations arithmétiques entre les champs 84

Chapitre 22: Extension ou substitution de modèle d'utilisateur 86

Examples 86

Modèle utilisateur personnalisé avec courrier électronique en tant que champ de connexion 86

Utilisez le `email` comme nom d'utilisateur et débarrassez-vous du champ` username` 89

Étendre facilement le modèle d'utilisateur de Django 91

Spécification d'un modèle d'utilisateur personnalisé 93

Référencement du modèle d'utilisateur 95

Chapitre 23: Formes 96

Examples 96

Exemple ModelForm 96

Définir un formulaire Django à partir de zéro (avec des widgets) 96

Suppression du champ modelForm en fonction de la condition de views.py 96

Téléchargement de fichiers avec les formulaires Django 98
Validation des champs et Validation du modèle (Modification de l'e-mail de l'utilisateur) 100
Chapitre 24: Formsets 102
Syntaxe 102
Examples 102
Formsets avec des données initialisées et unifiées 102
Chapitre 25: Fuseaux horaires 104
Introduction 104
Examples 104
Activer le support de fuseau horaire 104
Définition des fuseaux horaires de session 104
Chapitre 26: Gestionnaires et ensembles de requêtes personnalisés 107
Examples 107
Définition d'un gestionnaire de base à l'aide de la méthode Querysets et de la méthode `as 107
select_related pour toutes les requêtes 108
Définir des gestionnaires personnalisés 109
Chapitre 27: Intégration continue avec Jenkins 111
Examples 111
Script de pipeline Jenkins 2.0+ 111
Script de pipeline Jenkins 2.0+, conteneurs Docker 111
Chapitre 28: Internationalisation 113
Syntaxe 113
Examples 113
Introduction à l'internationalisation 113
Mise en place 113
settings.py 113
Marquage des chaînes comme traduisible 113
Traduire les chaînes 114
Traduction paresseuse vs non paresseuse 114
Traduction en gabarits 115
Traduire les chaînes 116

Cas d'utilisation Noop 118
Pièges communs 118
traductions floues 118
Cordes multilignes 119
Chapitre 29: JSONField - un champ spécifique à PostgreSQL 120
Syntaxe 120
Remarques 120
Enchaînement des requêtes 120
Examples 120
Créer un JSONField 120
Disponible dans Django 1.9+ 120
Création d'un objet avec des données dans un JSONField 121
Interrogation des données de niveau supérieur 121
Interrogation de données imbriquées dans des dictionnaires 121
Interrogation des données présentes dans les tableaux 121
Classement par valeurs JSONField 121
Chapitre 30: Le débogage 123
Remarques 123
Examples 123
Utilisation du débogueur Python (Pdb) 123
Utiliser la barre d'outils de débogage de Django 124
Utiliser "assert False" 126
Envisagez d'écrire plus de documentation, de tests, de journalisation et d'assertions au l 126
Chapitre 31: Les signaux 127
Paramètres 127
Remarques 127
Examples 128
Extension de l'exemple de profil utilisateur 128
Syntaxe différente pour poster / pré-signaler 128
Comment trouver s'il s'agit d'une insertion ou d'une mise à jour dans le signal pre_save 129
Héritage des signaux sur les modèles étendus 129
Chapitre 32: Mapper des chaînes à des chaînes avec HStoreField - un champ spécifique à Pos

131

Syntaxe 131

Examples 131

Configuration de HStoreField 131

Ajout de HStoreField à votre modèle 131

Créer une nouvelle instance de modèle 131

Effectuer des recherches de clés 132

Utiliser contient 132

Chapitre 33: Meta: Guide de documentation 133

Remarques 133

Examples 133

Les versions non prises en charge ne nécessitent pas de mention spéciale 133

Chapitre 34: Middleware 134

Introduction 134

Remarques 134

Examples 134

Ajouter des données aux requêtes 134

Middleware pour filtrer par adresse IP 135

Exception de traitement global 136

Comprendre le nouveau style du middleware Django 1.10 137

Chapitre 35: Migrations 138

Paramètres 138

Examples 138

Travailler avec des migrations 138

Migrations manuelles 139

Fausses migrations 141

Noms personnalisés pour les fichiers de migration 141

Résoudre les conflits de migration 141

introduction 141

Fusion de migrations 142

Changer un CharField en un ForeignKey 142

Chapitre 36: Paramètres 144

Examples....................144
Définition du fuseau horaire....................144
Accéder aux paramètres....................144
Utiliser BASE_DIR pour assurer la portabilité des applications....................145
Utilisation de variables d'environnement pour gérer les paramètres sur les serveurs....................145
settings.py....................146
Utiliser plusieurs paramètres....................146
Alternative n ° 1....................147
Alternative n ° 2....................147
Utiliser plusieurs fichiers de besoins....................147
Structure....................148
Masquage de données secrètes à l'aide d'un fichier JSON....................148
Utiliser un DATABASE_URL de l'environnement....................149
Chapitre 37: Processeurs de contexte....................151
Remarques....................151
Examples....................151
Utiliser un processeur de contexte pour accéder aux paramètres.DEBUG dans les modèles....................151
Utiliser un processeur de contexte pour accéder à vos entrées de blog les plus récentes da....................152
Extension de vos modèles....................153
Chapitre 38: Querysets....................154
Introduction....................154
Examples....................154
Requêtes simples sur un modèle autonome....................154
Requêtes avancées avec des objets Q....................155
Réduire le nombre de requêtes sur ManyToManyField (problème n + 1)....................155
Problème....................155
Solution....................156
Réduire le nombre de requêtes sur le champ ForeignKey (problème n + 1)....................157
Problème....................157
Solution....................158
Obtenir le jeu de requête SQL pour Django....................159

Obtenir le premier et le dernier enregistrement de QuerySet 159
Requêtes avancées avec des objets F 159
Chapitre 39: RangeFields - un groupe de champs spécifiques à PostgreSQL 161
Syntaxe 161
Examples 161
Inclure des champs de plage numériques dans votre modèle 161
Mise en place pour RangeField 161
Création de modèles avec des champs de plage numériques 161
Utiliser contient 161
Utiliser contain_by 162
Utiliser le chevauchement 162
Utiliser None pour signifier aucune limite supérieure 162
Opérations de gammes 162
Chapitre 40: Référence du champ du modèle 163
Paramètres 163
Remarques 164
Examples 164
Champs numériques 164
BinaryField 167
CharField 167
DateTimeField 167
Clé étrangère 168
Chapitre 41: Relations plusieurs-à-plusieurs 170
Examples 170
Avec un modèle traversant 170
Simple plusieurs à plusieurs relations. 171
Utiliser les champs ManyToMany 171
Chapitre 42: Routage d'URL 172
Examples 172
Comment Django gère une requête 172
Définir l'espace de noms URL pour une application réutilisable (Django 1.9+) 174
Chapitre 43: Routeurs de base de données 176

Examples 176
Ajout d'un fichier de routage de base de données 176
Spécifier différentes bases de données dans le code 177
Chapitre 44: Sécurité 178
Examples 178
Protection XSS (Cross Site Scripting) 178
Protection de clickjacking 179
Protection contre la falsification de sites inter-sites (CSRF) 180
Chapitre 45: Structure du projet 182
Examples 182
Repository> Project> Site / Conf 182
Noms de fichiers statiques et de fichiers dans les applications django 183
Chapitre 46: Tâches Async (Céleri) 184
Remarques 184
Examples 184
Exemple simple pour ajouter 2 nombres 184
Chapitre 47: Templating 186
Examples 186
Les variables 186
Templating in Class Views Vues 187
Création de modèles dans les vues basées sur les fonctions 187
Filtres de modèle 188
Empêcher les méthodes sensibles d'être appelées dans des modèles 189
Utilisation de {% extend%}, {% include%} et {% blocks%} 189
résumé 189
Guider 190
Chapitre 48: Test d'unité 192
Examples 192
Testing - un exemple complet 192
Tester les modèles Django efficacement 193
Tester le contrôle d'accès dans les vues Django 194
La base de données et les tests 196

Limiter le nombre de tests exécutés ... 197
Chapitre 49: Transactions de base de données ... 199
Examples ... 199
Transactions atomiques ... 199
Problème ... 199
Solution ... 199
Chapitre 50: Utiliser Redis avec Django - Caching Backend ... 201
Remarques ... 201
Examples ... 201
Utiliser django-redis-cache ... 201
Utiliser django-redis ... 201
Chapitre 51: Vues basées sur la classe ... 203
Remarques ... 203
Examples ... 203
Vues basées sur la classe ... 203
views.py ... 203
urls.py ... 203
Données de contexte ... 203
views.py ... 204
book.html ... 204
Vues de liste et de détails ... 204
app / models.py ... 204
app / views.py ... 204
app / templates / app / pokemon_list.html ... 205
app / templates / app / pokemon_detail.html ... 205
app / urls.py ... 205
Création de forme et d'objet ... 206
app / views.py ... 206
app / templates / app / pokemon_form.html (extrait) ... 207
app / templates / app / pokemon_confirm_delete.html (extrait) ... 207

app / models.py 207
Exemple minimal 208
Vues basées sur la classe Django: exemple de CreateView 208
Une vue, plusieurs formulaires 209
Chapitre 52: Vues génériques 211
Introduction 211
Remarques 211
Examples 211
Exemple minimum: vues fonctionnelle vs vues génériques 211
Personnalisation des vues génériques 212
Vues génériques avec des mixins 213
Chapitre 53: Widgets de formulaire 215
Examples 215
Widget de saisie de texte simple 215
Widget composite 215
Crédits 217

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: django

It is an unofficial and free Django ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Django.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Django

Remarques

Django se présente comme "le framework Web pour les perfectionnistes avec des délais" et "Django facilite la création de meilleures applications Web plus rapidement et avec moins de code". Cela peut être vu comme une architecture MVC. Au fond, il a:

- un serveur Web léger et autonome pour le développement et les tests
- un système de sérialisation et de validation de formulaires pouvant traduire entre des formulaires HTML et des valeurs propres au stockage dans la base de données
- un système de modèles utilisant le concept d'héritage emprunté à la programmation orientée objet
- un cadre de mise en cache pouvant utiliser plusieurs méthodes de cache pour les classes de middleware pouvant intervenir à différentes étapes du traitement des requêtes et exécuter des fonctions personnalisées
- un système de répartition interne qui permet aux composants d'une application de communiquer des événements entre eux via des signaux prédéfinis
- un système d'internationalisation, y compris la traduction des composants de Django dans différentes langues
- un système de sérialisation capable de produire et de lire des représentations XML et / ou JSON des instances de modèle Django
- un système d'extension des capacités du moteur de template
- une interface avec le framework de test intégré de Python

Versions

Version	Date de sortie
1.11	2017-04-04
1.10	2016-08-01
1,9	2015-12-01
1.8	2015-04-02
1,7	2014-09-02
1.6	2013-11-06
1,5	2013-02-26
1.4	2012-03-23
1.3	2011-03-23

Version	Date de sortie
1.2	2010-05-17
1.1	2009-07-29
1.0	2008-09-03

Examples

Lancer un projet

Django est un framework de développement web basé sur Python. Django **1.11** (la dernière version stable) nécessite l'installation de Python **2.7** , **3.4** , **3.5** ou **3.6** . En supposant que `pip` est disponible, l'installation est aussi simple que d'exécuter la commande suivante. Gardez à l'esprit que l'omission de la version ci-dessous installera la dernière version de django:

```
$ pip install django
```

Pour installer une version spécifique de django, supposons que la version soit django **1.10.5** , exécutez la commande suivante:

```
$ pip install django==1.10.5
```

Les applications Web créées avec Django doivent résider dans un projet Django. Vous pouvez utiliser la commande `django-admin` pour lancer un nouveau projet dans le répertoire actuel:

```
$ django-admin startproject myproject
```

où `myproject` est un nom qui identifie de manière unique le projet et peut être composé de **chiffres** , de **lettres** et de **traits de soulignement** .

Cela créera la structure de projet suivante:

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

Pour exécuter l'application, démarrez le serveur de développement

```
$ cd myproject
$ python manage.py runserver
```

Maintenant que le serveur fonctionne, `http://127.0.0.1:8000/` vous sur `http://127.0.0.1:8000/` avec

votre navigateur Web. Vous verrez la page suivante:

It worked!
Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [app_label]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

Par défaut, la commande `runserver` démarre le serveur de développement sur l'adresse IP interne du port `8000` . Ce serveur redémarre automatiquement lorsque vous modifiez votre code. Mais si vous ajoutez de nouveaux fichiers, vous devrez redémarrer manuellement le serveur.

Si vous souhaitez modifier le port du serveur, transmettez-le en tant qu'argument de ligne de commande.

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

Si vous souhaitez modifier l'adresse IP du serveur, transmettez-la avec le port.

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

Notez que `runserver` est uniquement pour les versions de débogage et les tests locaux. Les programmes de serveur spécialisés (tels qu'Apache) doivent toujours être utilisés en production.

Ajouter une application Django

Un projet Django contient généralement plusieurs `apps` . Ceci est simplement un moyen de structurer votre projet en modules plus petits et maintenables. Pour créer une application, accédez à votre dossier de projet (où `manage.py` est) et exécutez la commande `startapp` (modifiez *myapp en fonction* de ce que vous voulez):

```
python manage.py startapp myapp
```

Cela générera le dossier *myapp* et certains fichiers nécessaires pour vous, comme `models.py` et `views.py` .

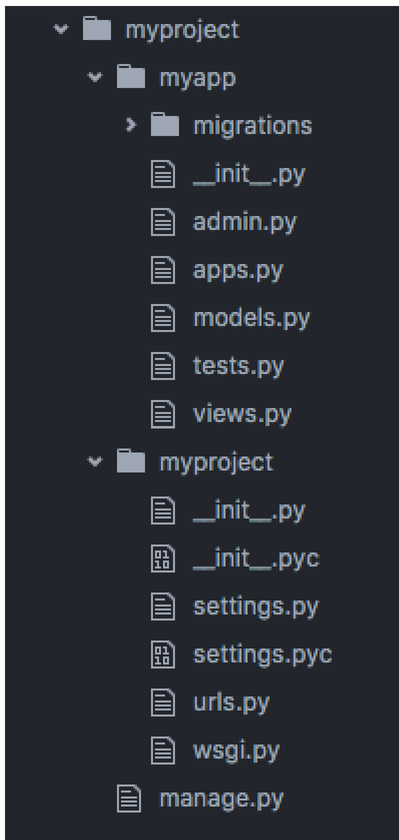
Afin de rendre Django au courant de *myapp* , ajoutez-le à vos `settings.py` :

```
# myproject/settings.py

# Application definition
INSTALLED_APPS = [
    ...
    'myapp',
```

```
]
```

La structure de dossier d'un projet Django peut être modifiée en fonction de vos préférences. Parfois, le dossier du projet est renommé en `/src` pour éviter de répéter les noms de dossier. Une structure de dossier typique ressemble à ceci:



Concepts Django

django-admin est un outil de ligne de commande **fourni** avec Django. Il est livré avec plusieurs commandes utiles pour démarrer et gérer un projet Django. La commande est la même que `./manage.py` , à la différence `./manage.py` que vous n'avez pas besoin d'être dans le répertoire du projet. La variable d'environnement `DJANGO_SETTINGS_MODULE` doit être définie.

Un **projet Django** est une base de code Python qui contient un fichier de paramètres Django. Un projet peut être créé par l'administrateur Django via la commande `django-admin startproject NAME` . Le projet a généralement un fichier appelé `manage.py` au niveau supérieur et un fichier URL racine appelé `urls.py` `manage.py` est une version spécifique à `django-admin` et vous permet d'exécuter des commandes de gestion sur ce projet. Par exemple, pour exécuter votre projet localement, utilisez `python manage.py runserver` . Un projet est composé d'applications Django.

Une **application Django** est un package Python qui contient un fichier de modèles (`models.py` par défaut) et d'autres fichiers tels que des URL et des vues spécifiques à une application. Une application peut être créée via la commande `django-admin startapp NAME` (cette commande doit être exécutée depuis le répertoire de votre projet). Pour qu'une application fasse partie d'un projet, elle doit être incluse dans la liste `INSTALLED_APPS` dans `settings.py` . Si vous avez utilisé la configuration standard, Django est livré avec plusieurs applications de ses propres applications préinstallées qui

gèrent des choses comme l' authentification pour vous. Les applications peuvent être utilisées dans plusieurs projets Django.

L' **ORM Django** collecte tous les modèles de base de données définis dans `models.py` et crée des tables de base de données en fonction de ces classes de modèles. Pour ce faire, commencez par configurer votre base de données en modifiant le paramètre `DATABASES` dans `settings.py` . Ensuite, une fois que vous avez défini vos modèles de base de données , exécutez les `python manage.py makemigrations` suivies de `python manage.py migrate` pour créer ou mettre à jour le schéma de votre base de données en fonction de vos modèles.

Un exemple complet de bonjour au monde.

Étape 1 Si vous avez déjà installé Django, vous pouvez ignorer cette étape.

```
pip install Django
```

Étape 2 Créer un nouveau projet

```
django-admin startproject hello
```

Cela va créer un dossier nommé `hello` qui contiendra les fichiers suivants:

```
hello/
├── hello/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

Etape 3 Dans le module `hello` (le dossier contenant le `__init.py__`), créez un fichier appelé `views.py` :

```
hello/
├── hello/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   ├── views.py  <- here
│   └── wsgi.py
└── manage.py
```

et mettre dans le contenu suivant:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse('Hello, World')
```

Ceci s'appelle une fonction de vue.

Étape 4 Modifiez `hello/urls.py` comme suit:

```
from django.conf.urls import url
from django.contrib import admin
from hello import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', views.hello)
]
```

qui lie la fonction de vue `hello()` à une URL.

Étape 5 Démarrez le serveur.

```
python manage.py runserver
```

Étape 6

Naviguez jusqu'à `http://localhost:8000/` dans un navigateur et vous verrez:

> Bonjour le monde

Environnement virtuel

Bien que cela ne soit pas strictement requis, il est fortement recommandé de démarrer votre projet dans un "environnement virtuel". Un environnement virtuel est un **conteneur** (un répertoire) **contenant** une version spécifique de Python et un ensemble de modules (dépendances), qui n'interfère pas avec le Python natif du système d'exploitation ni avec d'autres projets sur le même ordinateur.

En configurant un environnement virtuel différent pour chaque projet sur lequel vous travaillez, divers projets Django peuvent s'exécuter sur différentes versions de Python et peuvent gérer leurs propres ensembles de dépendances, sans risque de conflit.

Python 3.3+

Python 3.3+ inclut déjà un module `venv` standard, que vous pouvez généralement appeler `pyvenv` . Dans les environnements où la commande `pyvenv` n'est pas disponible, vous pouvez accéder aux mêmes fonctionnalités en `python3 -m venv` directement le module `python3 -m venv` .

Pour créer l'environnement virtuel:

```
$ pyvenv <env-folder>
# Or, if pyvenv is not available
$ python3 -m venv <env-folder>
```

Python 2

Si vous utilisez Python 2, vous pouvez d'abord l'installer en tant que module séparé de pip:

```
$ pip install virtualenv
```

Et puis créez l'environnement à l'aide de la commande `virtualenv` place:

```
$ virtualenv <env-folder>
```

Activer (n'importe quelle version)

L'environnement virtuel est maintenant configuré. Pour l'utiliser, il doit être *activé* dans le terminal que vous souhaitez utiliser.

Pour "activer" l'environnement virtuel (toute version de Python)

Linux aime:

```
$ source <env-folder>/bin/activate
```

Windows comme:

```
<env-folder>\Scripts\activate.bat
```

Cela modifie votre invite pour indiquer que l'environnement virtuel est actif. `(<env-folder>) $`

A partir de maintenant, tout ce qui est installé à l'aide de `pip` sera installé dans votre dossier env virtuel, pas à l'échelle du système.

Pour quitter l'environnement virtuel utilisation `deactivate` :

```
(<env-folder>) $ deactivate
```

Sinon, utilisez virtualenvwrapper

Vous pouvez également envisager d'utiliser virtualenvwrapper, ce qui rend la création et l'activation de virtualenv très pratique tout en le séparant de votre code:

```
# Create a virtualenv
mkvirtualenv my_virtualenv

# Activate a virtualenv
workon my_virtualenv
```

```
# Deactivate the current virtualenv
deactivate
```

Alternativement: utiliser pyenv + pyenv-viritualenv

Dans les environnements où vous devez gérer plusieurs versions de Python, vous pouvez bénéficier de virtualenv avec pyenv-virtualenv:

```
# Create a virtualenv for specific Python version
pyenv virtualenv 2.7.10 my-virtual-env-2.7.10

# Create a vritualenv for active python verion
pyenv virtualenv venv34

# Activate, deactivate virtualenv
pyenv activate <name>
pyenv deactivate
```

Lors de l'utilisation de virtualenvs, il est souvent utile de définir vos `PYTHONPATH` et `DJANGO_SETTINGS_MODULE` dans le script `postactivate` .

```
#!/bin/sh
# This hook is sourced after this virtualenv is activated

# Set PYTHONPATH to isolate the virtualenv so that only modules installed
# in the virtualenv are available
export PYTHONPATH="/home/me/path/to/your/project_root:$VIRTUAL_ENV/lib/python3.4"

# Set DJANGO_SETTINGS_MODULE if you don't use the default `myproject.settings`
# or if you use `django-admin` rather than `manage.py`
export DJANGO_SETTINGS_MODULE="myproject.settings.dev"
```

Définissez le chemin de votre projet

Il est également utile de définir le chemin de votre projet dans un fichier `.project` spécial situé dans votre `<env-folder>` . Pour ce faire, chaque fois que vous activez votre environnement virtuel, le répertoire actif est remplacé par le chemin spécifié.

Créez un nouveau fichier appelé `<env-folder>/.project` . Le contenu du fichier doit UNIQUEMENT être le chemin du répertoire du projet.

```
/path/to/project/directory
```

Maintenant, lancez votre environnement virtuel (en utilisant le `source <env-folder>/bin/activate` ou `workon my_virtualenv`) et votre terminal changera de répertoire en `/path/to/project/directory` .

Exemple de fichier unique Hello World

Cet exemple vous montre un moyen minimal de créer une page Hello World dans Django. Cela vous aidera à réaliser que la commande `django-admin startproject example` crée fondamentalement un tas de dossiers et de fichiers et que vous n'avez pas nécessairement besoin de cette structure pour exécuter votre projet.

1. Créez un fichier appelé `file.py`

2. Copiez et collez le code suivant dans ce fichier.

```
import sys

from django.conf import settings

settings.configure(
    DEBUG=True,
    SECRET_KEY='thisisthesecretkey',
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)

from django.conf.urls import url
from django.http import HttpResponse

# Your code goes below this line.

def index(request):
    return HttpResponse('Hello, World!')

urlpatterns = [
    url(r'^$', index),
]

# Your code goes above this line

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

3. Accédez au terminal et exécutez le fichier avec cette commande `python file.py runserver` .

4. Ouvrez votre navigateur et accédez à 127.0.0.1:8000 .

Projet compatible avec le déploiement avec support Docker.

Le modèle de projet par défaut de Django est correct, mais une fois que vous avez déployé votre code et que, par exemple, les devops mettent la main sur le projet, les choses se gâtent. Ce que vous pouvez faire est de séparer votre code source du reste qui doit être dans votre référentiel.

Vous pouvez trouver un modèle de projet Django utilisable sur GitHub .

Structure du projet

```
PROJECT_ROOT
├── devel.dockerfile
├── docker-compose.yml
├── nginx
│   └── project_name.conf
├── README.md
├── setup.py
└── src
    ├── manage.py
    └── project_name
        ├── __init__.py
        └── service
            ├── __init__.py
            ├── settings
            │   ├── common.py
            │   ├── development.py
            │   ├── __init__.py
            │   └── staging.py
            ├── urls.py
            └── wsgi.py
```

J'aime garder le répertoire de `service` nommé `service` pour chaque projet grâce à ce que je peux utiliser le même `Dockerfile` dans tous mes projets. La répartition des exigences et des paramètres est déjà bien documentée ici:
Utiliser plusieurs fichiers de besoins
Utiliser plusieurs paramètres

Dockerfile

En supposant que seuls les développeurs utilisent Docker (ce ne sont pas tous les développeurs qui lui font confiance de nos jours). Cela pourrait être un environnement de dev `devel.dockerfile` :

```
FROM python:2.7
ENV PYTHONUNBUFFERED 1

RUN mkdir /run/service
ADD . /run/service
WORKDIR /run/service

RUN pip install -U pip
RUN pip install -I -e .[develop] --process-dependency-links

WORKDIR /run/service/src
ENTRYPOINT ["python", "manage.py"]
CMD ["runserver", "0.0.0.0:8000"]
```

L'ajout des seules exigences permettra de tirer parti du cache Docker lors de la création. Il vous suffit de reconstruire en fonction des modifications apportées aux exigences.

Composer

Docker compose est pratique, en particulier lorsque vous avez plusieurs services à exécuter localement. `docker-compose.yml` :

```
version: '2'
services:
  web:
    build:
      context: .
      dockerfile: devel.dockerfile
    volumes:
      - "./src/{{ project_name }}:/run/service/src/{{ project_name }}"
      - "./media:/run/service/media"
    ports:
      - "8000:8000"
    depends_on:
      - db
  db:
    image: mysql:5.6
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE={{ project_name }}
  nginx:
    image: nginx
    ports:
      - "80:80"
    volumes:
      - "./nginx:/etc/nginx/conf.d"
      - "./media:/var/media"
    depends_on:
      - web
```

Nginx

Votre environnement de développement doit être aussi proche que possible de l'environnement de production, donc j'aime utiliser Nginx dès le début. Voici un exemple de fichier de configuration nginx:

```
server {
    listen   80;
    client_max_body_size 4G;
    keepalive_timeout 5;

    location /media/ {
        autoindex on;
        alias /var/media/;
    }

    location / {
        proxy_pass_header Server;
        proxy_set_header Host $http_host;
        proxy_redirect off;
```

```
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Scheme $scheme;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Ssl on;
        proxy_connect_timeout 600;
        proxy_read_timeout 600;
        proxy_pass http://web:8000/;
    }
}
```

Usage

```
$ cd PROJECT_ROOT
$ docker-compose build web  # build the image - first-time and after requirements change
$ docker-compose up  # to run the project
$ docker-compose run --rm --service-ports --no-deps  # to run the project - and be able to use
PDB
$ docker-compose run --rm --no-deps <management_command>  # to use other than runserver
commands, like makemigrations
$ docker exec -ti web bash  # For accessing django container shell, using it you will be
inside /run/service directory, where you can run ./manage shell, or other stuff
$ docker-compose start  # Starting docker containers
$ docker-compose stop  # Stopping docker containers
```

Lire Démarrer avec Django en ligne: https://riptutorial.com/fr/django/topic/200/demarrer-avec-django

Chapitre 2: Administration

Examples

Changer de liste

Disons que vous avez une application `myblog` simple avec le modèle suivant:

```
from django.conf import settings
from django.utils import timezone

class Article(models.Model):
    title = models.CharField(max_length=70)
    slug = models.SlugField(max_length=70, unique=True)
    author = models.ForeignKey(settings.AUTH_USER_MODEL, models.PROTECT)
    date_published = models.DateTimeField(default=timezone.now)
    is_draft = models.BooleanField(default=True)
    content = models.TextField()
```

La liste de changement de Django Admin est la page qui répertorie tous les objets d'un modèle donné.

```
from django.contrib import admin
from myblog.models import Article

@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    pass
```

Par défaut, il utilisera la `__str__()` (ou `__unicode__()` si vous êtes sur python2) de votre modèle pour afficher l'objet "name". Cela signifie que si vous ne le remplacez pas, vous verrez une liste d'articles, tous nommés "Objet article". Pour modifier ce comportement, vous pouvez définir la `__str__()` :

```
class Article(models.Model):
    def __str__(self):
        return self.title
```

Maintenant, tous vos articles doivent avoir un nom différent et plus explicite que "Objet article".

Cependant, vous souhaiterez peut-être afficher d'autres données dans cette liste. Pour cela, utilisez `list_display` :

```
@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_display = ['__str__', 'author', 'date_published', 'is_draft']
```

`list_display` n'est pas limité aux champs et propriétés du modèle. cela peut aussi être une méthode de votre `ModelAdmin` :

```
from django.forms.utils import flatatt
from django.urls import reverse
from django.utils.html import format_html

@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_display = ['title', 'author_link', 'date_published', 'is_draft']

    def author_link(self, obj):
        author = obj.author
        opts = author._meta
        route = '{}_{}_change'.format(opts.app_label, opts.model_name)
        author_edit_url = reverse(route, args=[author.pk])
        return format_html(
            '<a{}>{}</a>', flatatt({'href': author_edit_url}), author.first_name)

    # Set the column name in the change list
    author_link.short_description = "Author"
    # Set the field to use when ordering using this column
    author_link.admin_order_field = 'author__firstname'
```

Styles CSS et scripts JS supplémentaires pour la page d'administration

Supposons que vous ayez un modèle `Customer` simple:

```
class Customer(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    is_premium = models.BooleanField(default=False)
```

Vous l'enregistrez dans l'administrateur de Django et ajoutez un champ de recherche par `first_name` et `last_name` :

```
@admin.register(Customer)
class CustomerAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name', 'is_premium']
    search_fields = ['first_name', 'last_name']
```

Après cela, les champs de recherche apparaissent dans la page de la liste des administrateurs avec l'espace réservé par défaut: " *mot-clé* ". Mais que se passe-t-il si vous souhaitez remplacer cet espace réservé par " *Rechercher par nom* "?

Vous pouvez le faire en transmettant un fichier Javascript personnalisé dans admin `Media` :

```
@admin.register(Customer)
class CustomerAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name', 'is_premium']
    search_fields = ['first_name', 'last_name']

    class Media:
        #this path may be any you want,
        #just put it in your static folder
        js = ('js/admin/placeholder.js', )
```

Vous pouvez utiliser la barre d'outils de débogage du navigateur pour trouver l'id ou la classe Django définie sur cette barre de recherche, puis écrire votre code js:

```
$(function () {
   $('#searchbar').attr('placeholder', 'Search by name')
})
```

De plus, la classe `Media` vous permet d'ajouter des fichiers CSS avec un objet dictionnaire:

```
class Media:
    css = {
        'all': ('css/admin/styles.css',)
         }
```

Par exemple, nous devons afficher chaque élément de la colonne `first_name` dans une couleur spécifique.
Par défaut, la colonne create table de Django pour chaque élément de `list_display` , toutes les balises `<td>` auront la classe css comme `field-'list_display_name'` , dans notre cas il s'agira de `field_first_name`

```
.field_first_name {
     background-color: #e6f2ff;
 }
```

Si vous souhaitez personnaliser d'autres comportements en ajoutant JS ou certains styles css, vous pouvez toujours vérifier les identifiants et les classes d'éléments dans l'outil de débogage du navigateur.

Traiter des clés étrangères référençant de grandes tables

Par défaut, Django rend les champs `ForeignKey` comme une entrée `<select>` . Cela peut entraîner un **chargement très lent des** pages si vous avez des milliers ou des dizaines de milliers d'entrées dans la table référencée. Et même si vous n’avez que des centaines d’entrées, il est très difficile de rechercher une entrée particulière parmi toutes.

Un module externe très pratique pour cela est django-autocomplete-light (DAL). Cela permet d'utiliser des champs de saisie semi-automatique au lieu des champs `<select>` .

views.py

```
from dal import autocomplete

class CityAutocomp(autocomplete.Select2QuerySetView):
    def get_queryset(self):
        qs = City.objects.all()
        if self.q:
            qs = qs.filter(name__istartswith=self.q)
        return qs
```

urls.py

```
urlpatterns = [
    url(r'^city-autocomp/$', CityAutocomp.as_view(), name='city-autocomp'),
]
```

forms.py

```
from dal import autocomplete

class PlaceForm(forms.ModelForm):
    city = forms.ModelChoiceField(
        queryset=City.objects.all(),
        widget=autocomplete.ModelSelect2(url='city-autocomp')
    )

    class Meta:
        model = Place
```

```
        fields = ['__all__']
```

admin.py

```
@admin.register(Place)
class PlaceAdmin(admin.ModelAdmin):
    form = PlaceForm
```

Lire Administration en ligne: https://riptutorial.com/fr/django/topic/1219/administration

Chapitre 3: Agrégations de modèles

Introduction

Les agrégations sont des méthodes permettant l'exécution d'opérations sur des lignes (individuelles et / ou des groupes) d'objets dérivés d'un modèle.

Examples

Moyenne, Minimum, Maximum, Somme de Queryset

```
class Product(models.Model):
    name = models.CharField(max_length=20)
    price = models.FloatField()
```

Obtenir le prix moyen de tous les produits:

```
>>> from django.db.models import Avg, Max, Min, Sum
>>> Product.objects.all().aggregate(Avg('price'))
# {'price__avg': 124.0}
```

Pour obtenir le prix minimum de tous les produits:

```
>>> Product.objects.all().aggregate(Min('price'))
# {'price__min': 9}
```

Pour obtenir le prix maximum de tous les produits:

```
>>> Product.objects.all().aggregate(Max('price'))
# {'price__max':599 }
```

Pour obtenir la somme des prix de tous les produits:

```
>>> Product.objects.all().aggregate(Sum('price'))
# {'price__sum':92456 }
```

Compter le nombre de relations étrangères

```
class Category(models.Model):
    name = models.CharField(max_length=20)


class Product(models.Model):
    name = models.CharField(max_length=64)
    category = models.ForeignKey(Category, on_delete=models.PROTECT)
```

Pour obtenir le nombre de produits pour chaque catégorie:

```
>>> categories = Category.objects.annotate(Count('product'))
```

Cela ajoute l' `<field_name>__count` à chaque instance renvoyée:

```
>>> categories.values_list('name', 'product__count')
[('Clothing', 42), ('Footwear', 12), ...]
```

Vous pouvez fournir un nom personnalisé pour votre attribut en utilisant un argument de mot-clé:

```
>>> categories = Category.objects.annotate(num_products=Count('product'))
```

Vous pouvez utiliser le champ annoté dans les jeux de requêtes:

```
>>> categories.order_by('num_products')
[<Category: Footwear>, <Category: Clothing>]

>>> categories.filter(num_products__gt=20)
[<Category: Clothing>]
```

GROUB BY ... COUNT / SUM Django équivalent ORM

Nous pouvons effectuer une `GROUP BY ... COUNT` ou un `GROUP BY ... SUM` requêtes SQL équivalent sur Django ORM, avec l'utilisation de `annotate()` , les `values()` , `order_by()` et les `django.db.models` s » `Count` et `Sum` méthodes respectueusement:

Laissez notre modèle être:

```
   class Books(models.Model):
       title  = models.CharField()
       author = models.CharField()
       price = models.FloatField()
```

`GROUP BY ... COUNT` :

- Supposons que nous voulons compter le nombre d'objets du livre par auteur distinct dans notre tableau `Books` :

  ```
  result = Books.objects.values('author')
                        .order_by('author')
                        .annotate(count=Count('author'))
  ```

- Le `result` contient maintenant un jeu de requête avec deux colonnes: `author` et `count` :

  ```
    author    | count
  ------------|-------
   OneAuthor  |   5
  OtherAuthor |   2
     ...      |  ...
  ```

GROUB BY ... SUM :

- Supposons que nous voulons faire la somme du prix de tous les livres par auteur distinct qui existent dans notre tableau `Books` :

```
result = Books.objects.values('author')
                      .order_by('author')
                      .annotate(total_price=Sum('price'))
```

- Maintenant, le `result` contient un ensemble de requêtes avec deux colonnes: `author` et `total_price` :

```
  author    | total_price
------------|-------------
 OneAuthor  |    100.35
OtherAuthor |     50.00
    ...     |      ...
```

Lire Agrégations de modèles en ligne: https://riptutorial.com/fr/django/topic/3775/agregations-de-modeles

Chapitre 4: ArrayField - un champ spécifique à PostgreSQL

Syntaxe

- à partir de django.contrib.postgres.fields importer ArrayField
- class ArrayField (base_field, size = None, ** options)
- FooModel.objects.filter (array_field_name__contains = [objets, à, cocher])
- FooModel.objects.filter (array_field_name__contained_by = [objets, à, cocher])

Remarques

Notez que même si le paramètre de `size` est transmis à PostgreSQL, PostgreSQL ne l'exigera pas.

Lors de l'utilisation d' `ArrayField` il convient de garder à l'esprit ce mot d'avertissement de la documentation des baies Postgresql .

> Conseil: les tableaux ne sont pas des ensembles; La recherche d'éléments de tableau spécifiques peut être un signe de mauvaise conception de la base de données. Envisagez d'utiliser une table séparée avec une ligne pour chaque élément qui serait un élément de tableau. Cela sera plus facile à rechercher et sera probablement mieux adapté à un grand nombre d'éléments.

Examples

Un ArrayField de base

Pour créer un ArrayField PostgreSQL, nous devons donner à ArrayField le type de données que nous voulons stocker en tant que champ en tant que premier argument. Comme nous allons stocker les cotes de livres, nous utiliserons `FloatField` .

```
from django.db import models, FloatField
from django.contrib.postgres.fields import ArrayField

class Book(models.Model):
    ratings = ArrayField(FloatField())
```

Spécification de la taille maximale d'un ArrayField

```
from django.db import models, IntegerField
from django.contrib.postgres.fields import ArrayField

class IceCream(models.Model):
    scoops = ArrayField(IntegerField()  # we'll use numbers to ID the scoops
```

```
                    , size=6)  # our parlor only lets you have 6 scoops
```

Lorsque vous utilisez le paramètre size, il est transmis à postgresql, qui l'accepte et l'ignore ensuite! Il est donc tout à fait possible d'ajouter 7 entiers au champ `scoops` ci-dessus en utilisant la console postgresql.

Demander l'appartenance à ArrayField avec contains

Cette requête renvoie tous les cônes avec un scoop au chocolat et un scoop à la vanille.

```
VANILLA, CHOCOLATE, MINT, STRAWBERRY = 1, 2, 3, 4  # constants for flavors
choco_vanilla_cones = IceCream.objects.filter(scoops__contains=[CHOCOLATE, VANILLA])
```

N'oubliez pas d'importer le modèle `IceCream` partir de votre fichier `models.py` .

Gardez également à l'esprit que django ne créera pas d'index pour `ArrayField` . Si vous envisagez de les rechercher, vous aurez besoin d'un index qui devra être créé manuellement avec un appel à RunSQL dans votre fichier de migration.

Nid ArrayFields

Vous pouvez imbriquer `ArrayField` s en passant un autre `ArrayField` comme `base_field` .

```
from django.db import models, IntegerField
from django.contrib.postgres.fields import ArrayField

class SudokuBoard(models.Model):
    numbers = ArrayField(
        ArrayField(
            models.IntegerField(),
            size=9,
        ),
        size=9,
    )
```

Interrogation pour tous les modèles qui contiennent des éléments dans une liste avec contain_by

Cette requête renvoie tous les cônes avec un scoop à la menthe ou un scoop à la vanille.

```
minty_vanilla_cones = IceCream.objects.filter(scoops__contained_by=[MINT, VANILLA])
```

Lire ArrayField - un champ spécifique à PostgreSQL en ligne:
https://riptutorial.com/fr/django/topic/1693/arrayfield---un-champ-specifique-a-postgresql

Chapitre 5: Backends d'authentification

Examples

Backend d'authentification d'e-mail

L'authentification par défaut de Django fonctionne sur les champs `username` d' `username` et `password` . Le backend d'authentification par courrier électronique authentifiera les utilisateurs en fonction de leur `email` et de leur `password` .

```
from django.contrib.auth import get_user_model

class EmailBackend(object):
    """
    Custom Email Backend to perform authentication via email
    """
    def authenticate(self, username=None, password=None):
        user_model = get_user_model()
        try:
            user = user_model.objects.get(email=username)
            if user.check_password(password): # check valid password
                return user # return user to be authenticated
        except user_model.DoesNotExist: # no matching user exists
            return None

    def get_user(self, user_id):
        user_model = get_user_model()
        try:
            return user_model.objects.get(pk=user_id)
        except user_model.DoesNotExist:
            return None
```

Ajoutez ce moteur d'authentification au paramètre `AUTHENTICATION_BACKENDS` .

```
# settings.py
AUTHENTICATION_BACKENDS = (
    'my_app.backends.EmailBackend',
    ...
    )
```

Lire Backends d'authentification en ligne: https://riptutorial.com/fr/django/topic/1282/backends-d-authentification

Chapitre 6: Balises de modèle et filtres

Examples

Filtres personnalisés

Les filtres vous permettent d'appliquer une fonction à une variable. Cette fonction peut prendre **0** ou **1** argument. Voici la syntaxe:

```
{{ variable|filter_name }}
{{ variable|filter_name:argument }}
```

Les filtres peuvent être chaînés, ce qui est parfaitement valable:

```
{{ variable|filter_name:argument|another_filter }}
```

Si traduit en python la ligne ci-dessus donnerait quelque chose comme ceci:

```
print(another_filter(filter_name(variable, argument)))
```

Dans cet exemple, nous allons écrire un filtre personnalisé `verbose_name` qui s'applique à un modèle (instance ou classe) ou à un QuerySet. Il renverra le nom verbeux d'un modèle ou son nom verbeux si l'argument est défini sur `True` .

```
@register.filter
def verbose_name(model, plural=False):
    """Return the verbose name of a model.
    `model` can be either:
      - a Model class
      - a Model instance
      - a QuerySet
      - any object refering to a model through a `model` attribute.

    Usage:
      - Get the verbose name of an object
          {{ object|verbose_name }}
      - Get the plural verbose name of an object from a QuerySet
          {{ objects_list|verbose_name:True }}
    """
    if not hasattr(model, '_meta'):
        # handle the case of a QuerySet (among others)
        model = model.model
    opts = model._meta
    if plural:
        return opts.verbose_name_plural
    else:
        return opts.verbose_name
```

Tags simples

La manière la plus simple de définir une balise de modèle personnalisée consiste à utiliser une `simple_tag` . Celles-ci sont très simples à configurer. Le nom de la fonction sera le nom de la balise (bien que vous puissiez la remplacer), et les arguments seront des jetons ("mots" séparés par des espaces, sauf les espaces entre guillemets). Il prend même en charge les arguments de mots clés.

Voici un tag inutile qui illustrera notre exemple:

```
{% useless 3 foo 'hello world' foo=True bar=baz.hello|capfirst %}
```

Laissez `foo` et `baz` être des variables de contexte comme suit:

```
{'foo': "HELLO", 'baz': {'hello': "world"}}
```

Disons que nous voulons que cette balise très inutile se présente comme ceci:

```
HELLO;hello world;bar:World;foo:True<br/>
HELLO;hello world;bar:World;foo:True<br/>
HELLO;hello world;bar:World;foo:True<br/>
```

Type d'argument concaténation répétée 3 fois (3 étant le premier argument).

Voici à quoi peut ressembler l'implémentation du tag:

```
from django.utils.html import format_html_join

@register.simple_tag
def useless(repeat, *args, **kwargs):
    output = ';'.join(args + ['{}:{}'.format(*item) for item in kwargs.items()])
    outputs = [output] * repeat
    return format_html_join('\n', '{}<br/>', ((e,) for e in outputs))
```

`format_html_join` permet de marquer `<br/>` comme HTML sécurisé, mais pas le contenu des `outputs` .

Balises personnalisées avancées utilisant Node

Parfois, ce que vous voulez faire est trop complexe pour un `filter` ou un `simple_tag` . Pour ce faire, vous devrez créer une fonction de compilation et un rendu.

Dans cet exemple, nous allons créer une balise template `verbose_name` avec la syntaxe suivante:

Exemple	La description
`{% verbose_name obj %}`	Nom verbeux d'un modèle
`{% verbose_name obj 'status' %}`	Nom verbeux du champ "statut"
`{% verbose_name obj plural %}`	Nom verbeux du modèle d'un modèle

Exemple	La description
`{% verbose_name obj plural capfirst %}`	Capitalisation nominatif pluriel d'un modèle
`{% verbose_name obj 'foo' capfirst %}`	Nom verbeux en majuscule d'un champ
`{% verbose_name obj field_name %}`	Nom verbeux d'un champ à partir d'une variable
`{% verbose_name obj 'foo'|add:'_bar' %}`	Nom verbeux d'un champ "foo_bar"

La raison pour laquelle nous ne pouvons pas faire cela avec une simple balise est que `plural` et `capfirst` ne sont ni des variables ni des chaînes, ce sont des "mots-clés". Nous pourrions évidemment décider de les passer comme des chaînes `'plural'` et `'capfirst'` , mais cela pourrait entrer en conflit avec des champs portant ces noms. Est-ce que `{% verbose_name obj 'plural' %}` signifie "nom verbeux pluriel d' `obj` " ou "nom verbeux d' `obj.plural` "?

Commençons par créer la fonction de compilation:

```
@register.tag(name='verbose_name')
def do_verbose_name(parser, token):
    """
    - parser: the Parser object. We will use it to parse tokens into
              nodes such as variables, strings, ...
    - token: the Token object. We will use it to iterate each token
             of the template tag.
    """
    # Split tokens within spaces (except spaces inside quotes)
    tokens = token.split_contents()
    tag_name = tokens[0]
    try:
        # each token is a string so we need to parse it to get the actual
        # variable instead of the variable name as a string.
        model = parser.compile_filter(tokens[1])
    except IndexError:
        raise TemplateSyntaxError(
            "'{}' tag requires at least 1 argument.".format(tag_name))

    field_name = None
    flags = {
        'plural': False,
        'capfirst': False,
    }

    bits = tokens[2:]
    for bit in bits:
        if bit in flags.keys():
            # here we don't need `parser.compile_filter` because we expect
            # 'plural' and 'capfirst' flags to be actual strings.
            if flags[bit]:
                raise TemplateSyntaxError(
                    "'{}' tag only accept one occurrence of '{}' flag".format(
                        tag_name, bit)
                )
            flags[bit] = True
            continue
        if field_name:
            raise TemplateSyntaxError((
```

```
                "'{}' tag only accept one field name at most. {} is the second "
                "field name encountered."
            ).format(tag_name, bit)
        field_name = parser.compile_filter(bit)

    # VerboseNameNode is our renderer which code is given right below
    return VerboseNameNode(model, field_name, **flags)
```

Et maintenant le moteur de rendu:

```
class VerboseNameNode(Node):

    def __init__(self, model, field_name=None, **flags):
        self.model = model
        self.field_name = field_name
        self.plural = flags.get('plural', False)
        self.capfirst = flags.get('capfirst', False)

    def get_field_verbose_name(self):
        if self.plural:
            raise ValueError("Plural is not supported for fields verbose name.")
        return self.model._meta.get_field(self.field_name).verbose_name

    def get_model_verbose_name(self):
       if self.plural:
           return self.model._meta.verbose_name_plural
       else:
           return self.model._meta.verbose_name

    def render(self, context):
        """This is the main function, it will be called to render the tag.
        As you can see it takes context, but we don't need it here.
        For instance, an advanced version of this template tag could look for an
        `object` or `object_list` in the context if `self.model` is not provided.
        """
        if self.field_name:
            verbose_name = self.get_field_verbose_name()
        else:
            verbose_name = self.get_model_verbose_name()
        if self.capfirst:
            verbose_name = verbose_name.capitalize()
        return verbose_name
```

Lire Balises de modèle et filtres en ligne: https://riptutorial.com/fr/django/topic/1305/balises-de-modele-et-filtres

Chapitre 7: Céleri en cours d'exécution avec superviseur

Examples

Configuration du céleri

CÉLERI

1. Installation - installation du `pip install django-celery`

2. Ajouter

3. Structure de projet de base

```
- src/
  - bin/celery_worker_start # will be explained later on
  - logs/celery_worker.log
  - stack/__init __.py
  - stack/celery.py
  - stack/settings.py
  - stack/urls.py
  - manage.py
```

4. Ajoutez le fichier `celery.py` à votre `stack/stack/` dossier.

```
from __future__ import absolute_import
import os
from celery import Celery
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'stack.settings')
from django.conf import settings  # noqa
app = Celery('stack')
app.config_from_object('django.conf:settings')
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

5. à votre `stack/stack/__init__.py` ajoutez le code suivant:

```
from __future__ import absolute_import
from .celery import app as celery_app  # noqa
```

6. Créez une tâche et marquez-la par exemple sous la forme `@shared_task()`

```
@shared_task()
def add(x, y):
    print("x*y={}".format(x*y))
```

7. Travailleur de céleri en cours d'exécution "à la main":

`celery -A stack worker -l info` si vous souhaitez également ajouter

Superviseur en cours d'exécution

1. Créez un script pour démarrer le travailleur céleri. Insérez votre script dans votre application. Par exemple: `stack/bin/celery_worker_start`

```
#!/bin/bash

NAME="StackOverflow Project - celery_worker_start"

PROJECT_DIR=/home/stackoverflow/apps/proj/proj/
ENV_DIR=/home/stackoverflow/apps/proj/env/

echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd "${PROJECT_DIR}"

if [ -d "${ENV_DIR}" ]
then
    . "${ENV_DIR}bin/activate"
fi

celery -A stack --loglevel='INFO'
```

2. Ajoutez des droits d'exécution à votre script nouvellement créé:

```
chmod u+x bin/celery_worker_start
```

3. Installez le superviseur (ignorez ce test si le superviseur est déjà installé)

```
apt-get install supervisor
```

4. Ajoutez le fichier de configuration pour votre superviseur afin de démarrer votre céleri. Placez-le dans `/etc/supervisor/conf.d/stack_supervisor.conf`

```
[program:stack-celery-worker]
command = /home/stackoverflow/apps/stack/src/bin/celery_worker_start
user = polsha
stdout_logfile = /home/stackoverflow/apps/stack/src/logs/celery_worker.log
redirect_stderr = true
environment = LANG = en_US.UTF-8,LC_ALL = en_US.UTF-8
numprocs = 1
autostart = true
autorestart = true
startsecs = 10
stopwaitsecs = 600
priority = 998
```

5. Relire et mettre à jour le superviseur

```
sudo supervisorctl reread
    stack-celery-worker: available
sudo supervisorctl update
```

```
    stack-celery-worker: added process group
```

6. Commandes de base

```
sudo supervisorctl status stack-celery-worker
    stack-celery-worker      RUNNING    pid 18020, uptime 0:00:50
sudo supervisorctl stop stack-celery-worker
    stack-celery-worker: stopped
sudo supervisorctl start stack-celery-worker
    stack-celery-worker: started
sudo supervisorctl restart stack-celery-worker
    stack-celery-worker: stopped
    stack-celery-worker: started
```

Céleri + RabbitMQ avec Superviseur

Celery nécessite un courtier pour gérer le passage des messages. Nous utilisons RabbitMQ car il est facile à installer et il est bien pris en charge.

Installer rabbitmq en utilisant la commande suivante

```
sudo apt-get install rabbitmq-server
```

Une fois l'installation terminée, créez un utilisateur, ajoutez un hôte virtuel et définissez les autorisations.

```
sudo rabbitmqctl add_user myuser mypassword
sudo rabbitmqctl add_vhost myvhost
sudo rabbitmqctl set_user_tags myuser mytag
sudo rabbitmqctl set_permissions -p myvhost myuser ".*" ".*" ".*"
```

Pour démarrer le serveur:

```
sudo rabbitmq-server
```

Nous pouvons installer le céleri avec pip:

```
pip install celery
```

Dans votre fichier de paramètres Django, votre URL de courtier ressemblerait alors à

```
BROKER_URL = 'amqp://myuser:mypassword@localhost:5672/myvhost'
```

Maintenant, lancez le travailleur du céleri

```
celery -A your_app worker -l info
```

Cette commande lance un agent Celery pour exécuter les tâches définies dans votre application Django.

Supervisor est un programme Python qui vous permet de contrôler et d’exécuter tous les processus Unix. Il peut également redémarrer les processus en panne. Nous l'utilisons pour nous assurer que les travailleurs du céleri fonctionnent toujours.

Tout d'abord, installez le superviseur

```
sudo apt-get install supervisor
```

Créez votre fichier_proj.conf dans votre superviseur conf.d (/etc/supervisor/conf.d/your_proj.conf):

```
[program:your_proj_celery]
command=/home/your_user/your_proj/.venv/bin/celery --app=your_proj.celery:app worker -l info
directory=/home/your_user/your_proj
numprocs=1
stdout_logfile=/home/your_user/your_proj/logs/celery-worker.log
stderr_logfile=/home/your_user/your_proj/logs/low-worker.log
autostart=true
autorestart=true
startsecs=10
```

Une fois notre fichier de configuration créé et enregistré, nous pouvons informer le superviseur de notre nouveau programme par la commande supervisorctl. Tout d'abord, nous indiquons à Supervisor de rechercher toute configuration de programme nouvelle ou modifiée dans le répertoire /etc/supervisor/conf.d avec:

```
sudo supervisorctl reread
```

Suivi en lui disant d'activer tous les changements avec:

```
sudo supervisorctl update
```

Une fois nos programmes lancés, il y aura sans doute un moment où nous voudrons nous arrêter, redémarrer ou voir leur statut.

```
sudo supervisorctl status
```

Pour redémarrer votre instance de céleri:

```
sudo supervisorctl restart your_proj_celery
```

Lire Céleri en cours d'exécution avec superviseur en ligne: https://riptutorial.com/fr/django/topic/7091/celeri-en-cours-d-execution-avec-superviseur

Chapitre 8: Commandes de gestion

Introduction

Les commandes de gestion sont des scripts puissants et flexibles pouvant effectuer des actions sur votre projet Django ou la base de données sous-jacente. En plus des diverses commandes par défaut, il est possible d'écrire votre propre commande!

Comparé aux scripts Python classiques, l'utilisation de la structure de commande de gestion signifie que des tâches d'installation fastidieuses sont effectuées automatiquement pour vous.

Remarques

Les commandes de gestion peuvent être appelées à partir de:

- `django-admin <command> [options]`
- `python -m django <command> [options]`
- `python manage.py <command> [options]`
- `./manage.py <command> [options]` si manage.py a des droits d'exécution (`chmod +x manage.py`)

Pour utiliser des commandes de gestion avec Cron:

```
*/10 * * * * pythonuser /var/www/dev/env/bin/python /var/www/dev/manage.py <command> [options]
> /dev/null
```

Examples

Création et exécution d'une commande de gestion

Pour effectuer des actions dans Django en utilisant la ligne de commande ou d'autres services (où l'utilisateur / la demande n'est pas utilisé), vous pouvez utiliser les `management commands` .

Les modules Django peuvent être importés selon vos besoins.

Pour chaque commande, un fichier séparé doit être créé: `myapp/management/commands/my_command.py` (Les répertoires de `management` et de `commands` doivent avoir un fichier __init__.py vide)

```
from django.core.management.base import BaseCommand, CommandError

# import additional classes/modules as needed
# from myapp.models import Book

class Command(BaseCommand):
    help = 'My custom django management command'

    def add_arguments(self, parser):
       parser.add_argument('book_id', nargs='+', type=int)
       parser.add_argument('author' , nargs='+', type=str)
```

```
    def handle(self, *args, **options):
        bookid = options['book_id']
        author = options['author']
        # Your code goes here

        # For example:
        # books = Book.objects.filter(author="bob")
        # for book in books:
        #    book.name = "Bob"
        #    book.save()
```

Ici, le nom de classe **Command** est obligatoire et étend **BaseCommand** ou l'une de ses sous-classes.

Le nom de la commande de gestion est le nom du fichier le contenant. Pour exécuter la commande dans l'exemple ci-dessus, utilisez ce qui suit dans votre répertoire de projet:

```
python manage.py my_command
```

> Notez que le démarrage d'une commande peut prendre quelques secondes (à cause de l'importation des modules). Ainsi, dans certains cas, il est conseillé de créer des processus `daemon` au lieu de `management commands` de `management commands` .

Plus d'informations sur les commandes de gestion

Obtenir la liste des commandes existantes

Vous pouvez obtenir la liste des commandes disponibles de la manière suivante:

```
>>> python manage.py help
```

Si vous ne comprenez aucune commande ou recherchez des arguments facultatifs, vous pouvez utiliser l'argument **-h** comme ceci

```
>>> python manage.py command_name -h
```

Ici command_name sera le nom de votre commande désirée, cela vous montrera le texte d'aide de la commande.

```
>>> python manage.py runserver -h
>>> usage: manage.py runserver [-h] [--version] [-v {0,1,2,3}]
                           [--settings SETTINGS] [--pythonpath PYTHONPATH]
                           [--traceback] [--no-color] [--ipv6] [--nothreading]
                           [--noreload] [--nostatic] [--insecure]
                           [addrport]

Starts a lightweight Web server for development and also serves static files.

positional arguments:
  addrport              Optional port number, or ipaddr:port
```

```
optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -v {0,1,2,3}, --verbosity {0,1,2,3}
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=verbose output, 3=very verbose output
  --settings SETTINGS   The Python path to a settings module, e.g.
                        "myproject.settings.main". If this isn't provided, the
                        DJANGO_SETTINGS_MODULE environment variable will be
                        used.
  --pythonpath PYTHONPATH
                        A directory to add to the Python path, e.g.
                        "/home/djangoprojects/myproject".
  --traceback           Raise on CommandError exceptions
  --no-color            Don't colorize the command output.
  --ipv6, -6            Tells Django to use an IPv6 address.
  --nothreading         Tells Django to NOT use threading.
  --noreload            Tells Django to NOT use the auto-reloader.
  --nostatic            Tells Django to NOT automatically serve static files
                        at STATIC_URL.
  --insecure            Allows serving static files even if DEBUG is False.
```

Liste des commandes disponibles

Utiliser django-admin au lieu de manage.py

Vous pouvez vous débarrasser de `manage.py` et utiliser la commande `django-admin` place. Pour ce faire, vous devrez faire manuellement ce que `manage.py` :

- Ajoutez le chemin de votre projet à votre PYTHONPATH
- Définissez le DJANGO_SETTINGS_MODULE

```
export PYTHONPATH="/home/me/path/to/your_project"
export DJANGO_SETTINGS_MODULE="your_project.settings"
```

Ceci est particulièrement utile dans un virtualenv où vous pouvez définir ces variables d'environnement dans le script `postactivate` .

`django-admin` commande `django-admin` a l'avantage d'être disponible où que vous soyez sur votre système de fichiers.

Commandes de gestion intégrée

Django est livré avec un certain nombre de commandes de gestion intégrées, utilisant `python manage.py [command]` ou, lorsque manage.py a + x (exécutable) droits simplement `./manage.py [command]` . Les éléments suivants sont parmi les plus fréquemment utilisés:

Obtenir une liste de toutes les commandes disponibles

```
./manage.py help
```

Exécutez votre serveur Django sur localhost: 8000; essentiel pour les tests locaux

```
./manage.py runserver
```

Exécutez une console python (ou ipython si installé) avec les paramètres Django de votre projet préchargé (toute tentative d'accès à des parties de votre projet dans un terminal Python échouera).

```
./manage.py shell
```

Créez un nouveau fichier de migration de base de données en fonction des modifications apportées à vos modèles. Voir Migrations

```
./manage.py makemigrations
```

Appliquez toutes les migrations non appliquées à la base de données en cours.

```
./manage.py migrate
```

Exécutez la suite de tests de votre projet. Voir les tests unitaires

```
./manage.py test
```

Prenez tous les fichiers statiques de votre projet et mettez-les dans le dossier spécifié dans `STATIC_ROOT` pour qu'ils puissent être `STATIC_ROOT` en production.

```
./manage.py collectstatic
```

Permet de créer un superutilisateur.

```
./manage.py createsuperuser
```

Modifiez le mot de passe d'un utilisateur spécifié.

```
./manage.py changepassword username
```

Liste complète des commandes disponibles

Lire Commandes de gestion en ligne: https://riptutorial.com/fr/django/topic/1661/commandes-de-gestion

Chapitre 9: Comment réinitialiser les migrations django

Introduction

Lorsque vous développez une application Django, vous pouvez gagner beaucoup de temps en nettoyant et en réinitialisant vos migrations.

Examples

Réinitialisation de la migration Django: suppression de la base de données existante et migration en tant que nouvelle

Supprimer / Supprimer votre base de données Si vous utilisez SQLite pour votre base de données, supprimez simplement ce fichier. Si vous utilisez MySQL / Postgres ou tout autre système de base de données, vous devrez supprimer la base de données et recréer une nouvelle base de données.

Vous devrez maintenant supprimer tout le fichier de migration, à l'exception du fichier "init.py" situé dans le dossier migrations du dossier de votre application.

Le dossier des migrations se trouve généralement à

```
/your_django_project/your_app/migrations
```

Maintenant que vous avez supprimé la base de données et le fichier de migration, lancez simplement les commandes suivantes lors de la première migration du projet django.

```
python manage.py makemigrations
python manage.py migrate
```

Lire Comment réinitialiser les migrations django en ligne: https://riptutorial.com/fr/django/topic/9513/comment-reinitialiser-les-migrations-django

Chapitre 10: Comment utiliser Django avec Cookiecutter?

Examples

Installer et configurer le projet django en utilisant Cookiecutter

Voici les conditions préalables à l'installation de Cookiecutter:

- pépin
- virtualenv
- PostgreSQL

Créez un virtualenv pour votre projet et activez-le:

```
$ mkvirtualenv <virtualenv name>
$ workon <virtualenv name>
```

Maintenant, installez Cookiecutter en utilisant:

```
$ pip install cookiecutter
```

Modifiez les répertoires dans le dossier où vous souhaitez que votre projet vive. Exécutez maintenant la commande suivante pour générer un projet django:

```
$ cookiecutter https://github.com/pydanny/cookiecutter-django.git
```

Cette commande exécute cookiecutter avec le repo cookiecutter-django, nous demandant de saisir les détails spécifiques au projet. Appuyez sur "Entrée" sans rien saisir pour utiliser les valeurs par défaut, qui sont affichées entre [crochets] après la question.

```
project_name [project_name]: example_project
repo_name [example_project]:
author_name [Your Name]: Atul Mishra
email [Your email]: abc@gmail.com
description [A short description of the project.]: Demo Project
domain_name [example.com]: example.com
version [0.1.0]: 0.1.0
timezone [UTC]: UTC
now [2016/03/08]: 2016/03/08
year [2016]: 2016
use_whitenoise [y]: y
use_celery [n]: n
use_mailhog [n]: n
use_sentry [n]: n
use_newrelic [n]: n
use_opbeat [n]: n
windows [n]: n
```

```
use_python2 [n]: n
```

Vous trouverez plus de détails sur les options de génération de projet dans la documentation officielle. Le projet est maintenant configuré.

Lire Comment utiliser Django avec Cookiecutter? en ligne: https://riptutorial.com/fr/django/topic/5385/comment-utiliser-django-avec-cookiecutter-

Chapitre 11: Configuration de la base de données

Examples

MySQL / MariaDB

Django supporte MySQL 5.5 et supérieur.

Assurez-vous d'avoir quelques paquets installés:

```
$ sudo apt-get install mysql-server libmysqlclient-dev
$ sudo apt-get install python-dev python-pip              # for python 2
$ sudo apt-get install python3-dev python3-pip            # for python 3
```

Ainsi que l'un des pilotes MySQL Python (`mysqlclient` recommandé pour Django):

```
$ pip install mysqlclient    # python 2 and 3
$ pip install MySQL-python   # python 2
$ pip install pymysql        # python 2 and 3
```

> L'encodage de la base de données ne peut pas être défini par Django, mais doit être configuré au niveau de la base de données. Recherchez le jeu de caractères par `default-character-set` dans my.cnf (ou `/etc/mysql/mariadb.conf/*.cnf`) et définissez l'encodage:

```
    [mysql]
    #default-character-set = latin1    #default on some systems.
    #default-character-set = utf8mb4   #default on some systems.
    default-character-set = utf8

    ...
    [mysqld]
    #character-set-server  = utf8mb4
    #collation-server      = utf8mb4_general_ci
    character-set-server  = utf8
    collation-server      = utf8_general_ci
```

Configuration de la base de données pour MySQL ou MariaDB

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'DB_NAME',
        'USER': 'DB_USER',
        'PASSWORD': 'DB_PASSWORD',
        'HOST': 'localhost',   # Or an IP Address that your database is hosted on
```

```
        'PORT': '3306',
        #optional:
        'OPTIONS': {
            'charset' : 'utf8',
            'use_unicode' : True,
             'init_command': 'SET '
                'storage_engine=INNODB,'
                'character_set_connection=utf8,'
                'collation_connection=utf8_bin'
                #'sql_mode=STRICT_TRANS_TABLES,'    # see note below
                #'SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED',
        },
        'TEST_CHARSET': 'utf8',
        'TEST_COLLATION': 'utf8_general_ci',
    }
}
```

Si vous utilisez le connecteur MySQL d'Oracle, votre ligne `ENGINE` doit ressembler à ceci:

```
'ENGINE': 'mysql.connector.django',
```

Lorsque vous créez une base de données, assurez-vous que pour spécifier l'encodage et le classement:

```
CREATE DATABASE mydatabase CHARACTER SET utf8 COLLATE utf8_bin
```

> Depuis MySQL 5.7 et les nouvelles installations de MySQL 5.6, la valeur par défaut de l'option sql_mode contient **STRICT_TRANS_TABLES** . Cette option escalade les avertissements en erreurs lorsque les données sont tronquées lors de l'insertion. Django recommande vivement d'activer un *mode strict* pour MySQL afin d'éviter toute perte de données (STRICT_TRANS_TABLES ou STRICT_ALL_TABLES). Pour activer ajouter à */etc/my.cnf* `sql-mode = STRICT_TRANS_TABLES`

PostgreSQL

Assurez-vous d'avoir quelques paquets installés:

```
sudo apt-get install libpq-dev
pip install psycopg2
```

Paramètres de base de données pour PostgreSQL:

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'myprojectDB',
        'USER': 'myprojectuser',
        'PASSWORD': 'password',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
```

```
}
```

Dans les anciennes versions, vous pouvez également utiliser l'alias `django.db.backends.postgresql_psycopg2` .

Lorsque vous utilisez Postresql, vous aurez accès à certaines fonctionnalités supplémentaires:

Modelfields:

```
ArrayField          # A field for storing lists of data.
HStoreField         # A field for storing mappings of strings to strings.
JSONField           # A field for storing JSON encoded data.
IntegerRangeField  # Stores a range of integers
BigIntegerRangeField # Stores a big range of integers
FloatRangeField    # Stores a range of floating point values.
DateTimeRangeField # Stores a range of timestamps
```

sqlite

sqlite est la valeur par défaut pour Django. *Il ne doit pas être utilisé en production car il est généralement lent.*

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'db/development.sqlite3',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    },
}
```

Agencements

Les fixations sont des données initiales pour la base de données. Lorsque vous avez déjà des données existantes, la méthode la plus simple consiste à utiliser la commande `dumpdata`

```
 ./manage.py dumpdata > databasedump.json                # full database
 ./manage.py dumpdata myapp > databasedump.json          # only 1 app
 ./manage.py dumpdata myapp.mymodel > databasedump.json  # only 1 model (table)
```

Cela créera un fichier json qui pourra être réimporté en utilisant

```
./manage.py loaddata databasedump.json
```

Lorsque vous utilisez le `loadddata` sans spécifier de fichier, Django recherchera un dossier de `fixtures` dans votre application ou la liste des répertoires fournis dans les paramètres `FIXTURE_DIRS`

, et utilisera son contenu à la place.

```
/myapp
   /fixtures
        myfixtures.json
        morefixtures.xml
```

Les formats de fichiers possibles sont: `JSON, XML or YAML`

Exemple de JSON de luminaires:

```
[
  {
    "model": "myapp.person",
    "pk": 1,
    "fields": {
      "first_name": "John",
      "last_name": "Lennon"
    }
  },
  {
    "model": "myapp.person",
    "pk": 2,
    "fields": {
      "first_name": "Paul",
      "last_name": "McCartney"
    }
  }
]
```

Exemple de montage YAML:

```
- model: myapp.person
  pk: 1
  fields:
    first_name: John
    last_name: Lennon
- model: myapp.person
  pk: 2
  fields:
    first_name: Paul
    last_name: McCartney
```

Exemple XML de luminaires:

```
<?xml version="1.0" encoding="utf-8"?>
<django-objects version="1.0">
    <object pk="1" model="myapp.person">
        <field type="CharField" name="first_name">John</field>
        <field type="CharField" name="last_name">Lennon</field>
    </object>
    <object pk="2" model="myapp.person">
        <field type="CharField" name="first_name">Paul</field>
        <field type="CharField" name="last_name">McCartney</field>
    </object>
</django-objects>
```

Moteur Django Cassandra

- Installez pip: `$ pip install django-cassandra-engine`
- Ajoutez Getting Started à INSTALLED_APPS dans votre fichier settings.py: `INSTALLED_APPS = ['django_cassandra_engine']`
- Réglage de Cange DATABASES Standart:

  Standart

```
DATABASES = {
    'default': {
        'ENGINE': 'django_cassandra_engine',
        'NAME': 'db',
        'TEST_NAME': 'test_db',
        'HOST': 'db1.example.com,db2.example.com',
        'OPTIONS': {
            'replication': {
                'strategy_class': 'SimpleStrategy',
                'replication_factor': 1
            }
        }
    }
}
```

  Cassandra crée un nouvel utilisateur cqlsh:

```
DATABASES = {
'default': {
    'ENGINE': 'django_cassandra_engine',
    'NAME': 'db',
    'TEST_NAME': 'test_db',
    'USER_NAME'='cassandradb',
    'PASSWORD'= '123cassandra',
    'HOST': 'db1.example.com,db2.example.com',
    'OPTIONS': {
        'replication': {
            'strategy_class': 'SimpleStrategy',
            'replication_factor': 1
        }
    }
}
```

}

Lire Configuration de la base de données en ligne: https://riptutorial.com/fr/django/topic/4933/configuration-de-la-base-de-donnees

Chapitre 12: CRUD à Django

Examples

** Exemple CRUD le plus simple **

Si vous trouvez ces étapes peu familières, envisagez plutôt de commencer ici . Notez que ces étapes proviennent de la documentation de débordement de pile.

```
django-admin startproject myproject
cd myproject
python manage.py startapp myapp
```

myproject / settings.py Installer l'application

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

Créez un fichier appelé `urls.py` dans le répertoire **myapp** et `urls.py` le à jour avec la vue suivante.

```
from django.conf.urls import url
from myapp import views

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

Mettez à jour l'autre fichier `urls.py` avec le contenu suivant.

```
from django.conf.urls import url
from django.contrib import admin
from django.conf.urls import include
from myapp import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^myapp/', include('myapp.urls')),
    url(r'^admin/', admin.site.urls),
]
```

Créez un dossier nommé `templates` dans le répertoire **myapp** . Créez ensuite un fichier nommé `index.html` dans le répertoire des **modèles** . Remplissez-le avec le contenu suivant.

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    <h3>Add a Name</h3>
    <button>Create</button>
</body>
</html>
```

Nous avons également besoin d'une vue pour afficher **index.html** que nous pouvons créer en éditant le fichier **views.py** comme **suit** :

```
from django.shortcuts import render, redirect

# Create your views here.
def index(request):
    return render(request, 'index.html', {})
```

Vous avez maintenant la base sur laquelle vous allez travailler. L'étape suivante consiste à créer un modèle. Voici l'exemple le plus simple possible. Dans votre dossier **models.py** , ajoutez le code suivant.

```
from __future__ import unicode_literals

from django.db import models

# Create your models here.
class Name(models.Model):
    name_value = models.CharField(max_length=100)

    def __str__(self): # if Python 2 use __unicode__
        return self.name_value
```

Cela crée un modèle d'objet Name que nous ajouterons à la base de données à l'aide des commandes suivantes à partir de la ligne de commande.

```
python manage.py createsuperuser
python manage.py makemigrations
python manage.py migrate
```

Vous devriez voir certaines opérations effectuées par Django. Celles-ci configurent les tables et créent un super-utilisateur capable d'accéder à la base de données admin à partir d'une vue d'administrateur alimentée par Django. En parlant de cela, permet d'enregistrer notre nouveau modèle avec la vue admin. Accédez à **admin.py** et ajoutez le code suivant.

```
from django.contrib import admin
from myapp.models import Name
# Register your models here.

admin.site.register(Name)
```

De retour à la ligne de commande, vous pouvez maintenant lancer le serveur avec la commande `python manage.py runserver` . Vous devriez pouvoir visiter http: // localhost: 8000 / et voir votre application. Veuillez vous rendre sur http: // localhost: 8000 / admin pour pouvoir ajouter un nom à votre projet. Connectez-vous et ajoutez un nom sous la table MYAPP, nous avons gardé cela simple pour l'exemple, donc assurez-vous qu'il y a moins de 100 caractères.

Pour accéder au nom, vous devez l'afficher quelque part. Modifiez la fonction d'index dans **views.py** pour extraire tous les objets Name de la base de données.

```
from django.shortcuts import render, redirect
from myapp.models import Name

# Create your views here.
def index(request):
    names_from_db = Name.objects.all()
    context_dict = {'names_from_context': names_from_db}
    return render(request, 'index.html', context_dict)
```

Maintenant, éditez le fichier **index.html** comme suit.

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    {% if names_from_context %}
        <ul>
            {% for name in names_from_context %}
                <li>{{ name.name_value }}  <button>Delete</button>
<button>Update</button></li>
            {% endfor %}
        </ul>
    {% else %}
        <h3>Please go to the admin and add a Name under 'MYAPP'</h3>
    {% endif %}
    <h3>Add a Name</h3>
    <button>Create</button>
</body>
</html>
```

Cela démontre le Read in CRUD. Dans le répertoire **myapp** , créez un fichier forms.py. Ajoutez le code suivant:

```
from django import forms
from myapp.models import Name

class NameForm(forms.ModelForm):
    name_value = forms.CharField(max_length=100, help_text = "Enter a name")

    class Meta:
        model = Name
        fields = ('name_value',)
```

Mettez à jour l' **index.html** de la manière suivante:

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    {% if names_from_context %}
        <ul>
            {% for name in names_from_context %}
                <li>{{ name.name_value }}  <button>Delete</button>
<button>Update</button></li>
            {% endfor %}
        </ul>
    {% else %}
        <h3>Please go to the admin and add a Name under 'MYAPP'</h3>
    {% endif %}
    <h3>Add a Name</h3>
    <form id="name_form" method="post" action="/">
        {% csrf_token %}
        {% for field in form.visible_fields %}
            {{ field.errors }}
            {{ field.help_text }}
            {{ field }}
        {% endfor %}
        <input type="submit" name="submit" value="Create">
    </form>
</body>
</html>
```

Ensuite, mettez à jour le **fichier views.py** de la manière suivante:

```
from django.shortcuts import render, redirect
from myapp.models import Name
from myapp.forms import NameForm

# Create your views here.
def index(request):
    names_from_db = Name.objects.all()

    form = NameForm()

    context_dict = {'names_from_context': names_from_db, 'form': form}

    if request.method == 'POST':
        form = NameForm(request.POST)

        if form.is_valid():
            form.save(commit=True)
            return render(request, 'index.html', context_dict)
        else:
            print(form.errors)

    return render(request, 'index.html', context_dict)
```

Redémarrez votre serveur et vous devriez maintenant avoir une version de l'application avec le C

in create terminée.

TODO ajouter mise à jour et supprimer

Lire CRUD à Django en ligne: https://riptutorial.com/fr/django/topic/7317/crud-a-django

Chapitre 13: Déploiement

Examples

Lancer l'application Django avec Gunicorn

1. Installez gunicorn

   ```
   pip install gunicorn
   ```

2. À partir du dossier de projet django (même dossier où se trouve manage.py), exécutez la commande suivante pour exécuter le projet django actuel avec gunicorn

   ```
   gunicorn [projectname].wsgi:application -b 127.0.0.1:[port number]
   ```

   Vous pouvez utiliser l'option `--env` pour définir le chemin d'accès aux paramètres

   ```
   gunicorn --env DJANGO_SETTINGS_MODULE=[projectname].settings [projectname].wsgi
   ```

   ou exécuter en tant que processus démon en utilisant l'option `-D`

3. Après un démarrage réussi du gunicorn, les lignes suivantes apparaîtront dans la console

   `Starting gunicorn 19.5.0`

   `Listening at: http://127.0.0.1:[port number] ([pid])`

   `....` (autres informations supplémentaires sur le serveur gunicorn)

Déploiement avec Heroku

1. Téléchargez Heroku Toolbelt .

2. Accédez à la racine des sources de votre application Django. Vous aurez besoin de tk

3. Tapez `heroku create [app_name]` . Si vous ne donnez pas de nom à une application, Heroku en générera un au hasard pour vous. L'URL de votre application sera `http://[app name].herokuapp.com`

4. Créez un fichier texte nommé `Procfile` . Ne mettez pas une extension à la fin.

   ```
   web: <bash command to start production server>
   ```

   > Si vous avez un processus de travail, vous pouvez l'ajouter également. Ajoutez une autre ligne au format suivant: `worker-name: <bash command to start worker>`

5. Ajouter un requirements.txt.

- Si vous utilisez un environnement virtuel, exécutez `pip freeze > requirements.txt`
- Sinon, *obtenez un environnement virtuel!* . Vous pouvez également lister manuellement les

paquets Python dont vous avez besoin, mais cela ne sera pas couvert dans ce tutoriel.

6. **C'est le temps de déploiement!**

    1. `git push heroku master`

        Heroku a besoin d'un référentiel git ou d'un dossier déroulant pour effectuer des déploiements. Vous pouvez également configurer le rechargement automatique à partir d'un dépôt GitHub sur `heroku.com` , mais nous ne couvrirons pas cela dans ce tutoriel.

    2. `heroku ps:scale web=1`

        Cela fait passer le nombre de "dynos" Web à un. Vous pouvez en apprendre plus sur les dynos ici.

    3. `heroku open` ou naviguer sur `http://app-name.herokuapp.com`

        Astuce: `heroku open` ouvre l'URL de votre application heroku dans le navigateur par défaut.

7. Ajouter des **modules complémentaires** . Vous devrez configurer votre application Django pour la lier aux bases de données fournies dans Heroku en tant que "modules complémentaires". Cet exemple ne couvre pas cela, mais un autre exemple est en préparation pour le déploiement de bases de données dans Heroku.

Déploiement à distance simple fabfile.py

Fabric est une bibliothèque et un outil de ligne de commande Python (2.5-2.7) permettant de rationaliser l'utilisation de SSH pour les tâches de déploiement d'applications ou d'administration de systèmes. Il vous permet d'exécuter des fonctions Python arbitraires via la ligne de commande.

Installez le tissu à l’aide du `pip install fabric`
Créez `fabfile.py` dans votre répertoire racine:

```
#myproject/fabfile.py
from fabric.api import *

@task
def dev():
    # details of development server
    env.user = # your ssh user
    env.password = #your ssh password
    env.hosts = # your ssh hosts (list instance, with comma-separated hosts)
    env.key_filename = # pass to ssh key for github in your local keyfile

@task
def release():
    # details of release server
    env.user = # your ssh user
    env.password = #your ssh password
    env.hosts = # your ssh hosts (list instance, with comma-separated hosts)
    env.key_filename = # pass to ssh key for github in your local keyfile
```

```
@task
def run():
    with cd('path/to/your_project/'):
        with prefix('source ../env/bin/activate'):
        # activate venv, suppose it appear in one level higher
            # pass commands one by one
            run('git pull')
            run('pip install -r requirements.txt')
            run('python manage.py migrate --noinput')
            run('python manage.py collectstatic --noinput')
            run('touch reload.txt')
```

Pour exécuter le fichier, utilisez simplement la commande `fab` :

```
$ fab dev run  # for release server, `fab release run`
```

Remarque: vous ne pouvez pas configurer les clés ssh pour github et tapez simplement login et mot de passe manuellement, alors que fabfile fonctionne, de même avec les clés.

Utilisation du modèle de démarrage Heroku Django.

Si vous envisagez d’héberger votre site Web Django sur Heroku, vous pouvez démarrer votre projet en utilisant le modèle de démarrage Heroku Django:

```
django-admin.py startproject --template=https://github.com/heroku/heroku-django-
template/archive/master.zip --name=Procfile YourProjectName
```

Il a une configuration prête pour la production pour les fichiers statiques, les paramètres de base de données, Gunicorn, etc. Cela vous fera gagner du temps, c'est tout prêt pour l'hébergement sur Heroku, construisez simplement votre site web en haut de ce modèle.

Pour déployer ce modèle sur Heroku:

```
git init
git add -A
git commit -m "Initial commit"

heroku create
git push heroku master

heroku run python manage.py migrate
```

C'est tout!

Instructions de déploiement Django. Nginx + Gunicorn + Superviseur sur Linux (Ubuntu)

Trois outils de base.

1. nginx - un serveur HTTP et un proxy inversé open source, haute performance et gratuit,

avec de hautes performances;
2. gunicorn - 'Green Unicorn' est un serveur HTTP WSGI Python pour UNIX (nécessaire pour gérer votre serveur);
3. supervisor - un système client / serveur qui permet à ses utilisateurs de surveiller et de contrôler un certain nombre de processus sur des systèmes d'exploitation de type UNIX. Utilisé lorsque votre application ou le système tombe en panne, redémarre votre caméra django / celery / celery, etc.

Pour que ce soit simple, supposons que votre application se trouve dans ce répertoire: `/home/root/app/src/` et nous allons utiliser `root` utilisateur `root` (mais vous devez créer un utilisateur séparé pour votre application). De plus, notre environnement virtuel sera situé dans `/home/root/app/env/` path.

NGINX

Commençons par nginx. Si nginx n'est pas déjà sur la machine, installez-le avec `sudo apt-get install nginx` . Plus tard, vous devez créer un nouveau fichier de configuration dans votre répertoire nginx `/etc/nginx/sites-enabled/yourapp.conf` . S'il existe un fichier nommé `default.conf` , supprimez-le.

Code ci-dessous dans un fichier de configuration nginx, qui essaiera d'exécuter votre service en utilisant un fichier de socket; Plus tard, il y aura une configuration de gunicorn. Le fichier Socket est utilisé ici pour communiquer entre nginx et gunicorn. Cela peut aussi être fait avec les ports.

```
# your application name; can be whatever you want
upstream yourappname {
    server        unix:/home/root/app/src/gunicorn.sock fail_timeout=0;
}

server {
    # root folder of your application
    root        /home/root/app/src/;

    listen        80;
    # server name, your main domain, all subdomains and specific subdomains
    server_name   yourdomain.com *.yourdomain.com somesubdomain.yourdomain.com

    charset       utf-8;

    client_max_body_size                        100m;

    # place where logs will be stored;
    # folder and files have to be already located there, nginx will not create
    access_log        /home/root/app/src/logs/nginx-access.log;
    error_log         /home/root/app/src/logs/nginx-error.log;

    # this is where your app is served (gunicorn upstream above)
    location / {
        uwsgi_pass  yourappname;
        include     uwsgi_params;
    }

    # static files folder, I assume they will be used
    location /static/ {
```

```
        alias         /home/root/app/src/static/;
    }

    # media files folder
    location /media/ {
        alias         /home/root/app/src/media/;
    }

}
```

GUNICORN

Maintenant, notre script GUNICORN, qui sera responsable de l'exécution de l'application django sur le serveur. La première chose à faire est d'installer gunicorn dans un environnement virtuel avec `pip install gunicorn` .

```
#!/bin/bash

ME="root"
DJANGODIR=/home/root/app/src # django app dir
SOCKFILE=/home/root/app/src/gunicorn.sock # your sock file - do not create it manually
USER=root
GROUP=webapps
NUM_WORKERS=3
DJANGO_SETTINGS_MODULE=yourapp.yoursettings
DJANGO_WSGI_MODULE=yourapp.wsgi
echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd $DJANGODIR

source /home/root/app/env/bin/activate
export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
export PYTHONPATH=$DJANGODIR:$PYTHONPATH

# Create the run directory if it doesn't exist
RUNDIR=$(dirname $SOCKFILE)
test -d $RUNDIR || mkdir -p $RUNDIR

# Start your Django Gunicorn
# Programs meant to be run under supervisor should not daemonize themselves (do not use --
daemon)
exec /home/root/app/env/bin/gunicorn ${DJANGO_WSGI_MODULE}:application \
  --name root \
  --workers $NUM_WORKERS \
  --user=$USER --group=$GROUP \
  --bind=unix:$SOCKFILE \
  --log-level=debug \
  --log-file=-
```

afin de pouvoir exécuter le script de démarrage gunicorn, il doit avoir le mode d'exécution activé pour

`sudo chmod u+x /home/root/app/src/gunicorn_start`

maintenant vous pourrez démarrer votre serveur `./gunicorn_start` utilisant simplement

```
./gunicorn_start
```

SUPERVISEUR

Comme indiqué au début, nous voulons que notre application soit redémarrée en cas d’échec d’un superviseur. Si superviseur pas encore sur le serveur, installez avec le `sudo apt-get install supervisor` .

Au premier installez le superviseur. Ensuite, créez un fichier `.conf` dans votre répertoire principal `/etc/supervisor/conf.d/your_conf_file.conf`

contenu du fichier de configuration:

```
[program:yourappname]
command = /home/root/app/src/gunicorn_start
user = root
stdout_logfile = /home/root/app/src/logs/gunicorn_supervisor.log
redirect_stderr = true
```

Bref bref, `[program:youappname]` est requis au début, ce sera notre identifiant. Le fichier `stdout_logfile` est également un fichier dans lequel les journaux seront stockés, à la fois l’accès et les erreurs.

Cela fait, nous devons dire à notre superviseur que nous venons d'ajouter un nouveau fichier de configuration. Pour ce faire, il existe différents processus pour différentes versions d'Ubuntu.

Pour `Ubuntu version 14.04 or lesser` , lancez simplement ces commandes:

`sudo supervisorctl reread` -> relit tous les fichiers de configuration du catalogue des superviseurs, cela devrait s'imprimer: votre **nomappli: disponible**

`sudo supervisorctl update` -> met à jour le superviseur pour les fichiers de configuration nouvellement ajoutés; devrait imprimer votre **nomapplication: groupe de processus ajouté**

Pour `Ubuntu 16.04` Exécuter:

```
sudo service supervisor restart
```

et pour vérifier si votre application fonctionne correctement, exécutez simplement

`sudo supervisorctl status yourappname`

Cela devrait afficher:

`yourappname RUNNING pid 18020, uptime 0:00:50`

Pour obtenir une démonstration en direct de cette procédure, naviguez sur cette vidéo .

Déployer localement sans configurer apache / nginx

Déploiement recommandé du mode de production pour utiliser Apache / Nginx pour servir le contenu statique. Ainsi, lorsque `DEBUG` est faux, le contenu statique et le contenu du support ne peuvent pas être chargés. Cependant, nous pouvons charger le contenu statique en déploiement sans avoir à configurer le serveur Apache / Nginx pour notre application en utilisant:

```
python manage.py runserver --insecure
```

Ceci est uniquement destiné au déploiement local (par exemple LAN) et ne doit jamais être utilisé en production et n'est disponible que si l'application `staticfiles` trouve dans le paramètre `INSTALLED_APPS` votre projet.

Lire Déploiement en ligne: https://riptutorial.com/fr/django/topic/2792/deploiement

Chapitre 14: Des modèles

Introduction

Dans le cas de base, un modèle est une classe Python mappée sur une seule table de base de données. Les attributs de la classe mappent aux colonnes de la table et une instance de la classe représente une ligne dans la table de base de données. Les modèles héritent de `django.db.models.Model` qui fournit une API riche pour ajouter et filtrer les résultats de la base de données.

Créez votre premier modèle

Examples

Créer votre premier modèle

Les modèles sont généralement définis dans le fichier `models.py` sous le sous-répertoire de votre application. La classe `Model` du module `django.db.models` est une bonne classe de départ pour étendre vos modèles. Par exemple:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey('Author', on_delete=models.CASCADE,
related_name='authored_books')
    publish_date = models.DateField(null=True, blank=True)

    def __str__(self): # __unicode__ in python 2.*
        return self.title
```

Chaque attribut dans un modèle représente une colonne dans la base de données.

- `title` est un texte d'une longueur maximale de 100 caractères
- `author` est une `ForeignKey` qui représente une relation avec un autre modèle / tableau, en l'occurrence `Author` (utilisé uniquement à titre d'exemple). `on_delete` indique à la base de données quoi faire avec l'objet si l'objet associé (un `Author`) est supprimé. (Il convient de noter que django 1.9 `on_delete` peut être utilisé comme deuxième argument de position. Dans django 2, il s'agit d'un argument **obligatoire** et il est conseillé de le traiter immédiatement. Dans les versions antérieures, il sera `CASCADE` par défaut sur `CASCADE` .)
- `publish_date` stocke une date. Les `null` et `blank` sont définies sur `True` pour indiquer qu'il ne s'agit pas d'un champ obligatoire (vous pouvez l'ajouter ultérieurement ou le laisser vide).

Avec les attributs que nous définissons, une méthode `__str__` renvoie le titre du livre qui sera utilisé comme représentation sous forme de `string` , plutôt que la valeur par défaut.

Appliquer les modifications à la base de données (Migrations)

Après avoir créé un nouveau modèle ou modifié des modèles existants, vous devrez générer des migrations pour vos modifications, puis appliquer les migrations à la base de données spécifiée. Cela peut être fait en utilisant le système de migration intégré de Django. A l'aide de l'utilitaire `manage.py` , dans le répertoire racine du projet:

```
python manage.py makemigrations <appname>
```

La commande ci-dessus créera les scripts de migration nécessaires sous le sous-répertoire `migrations` de votre application. Si vous omettez le paramètre `<appname>` , toutes les applications définies dans l'argument `INSTALLED_APPS` de `settings.py` seront traitées. Si vous le trouvez nécessaire, vous pouvez modifier les migrations.

Vous pouvez vérifier quelles migrations sont nécessaires sans créer la migration en utilisant l'option --dry-run, par exemple:

```
python manage.py makemigrations --dry-run
```

Pour appliquer les migrations:

```
python manage.py migrate <appname>
```

La commande ci-dessus exécutera les scripts de migration générés lors de la première étape et mettra à jour physiquement la base de données.

Si le modèle de la base de données existante est modifié, la commande suivante est nécessaire pour effectuer les modifications nécessaires.

```
python manage.py migrate --run-syncdb
```

Django va créer la table avec le nom `<appname>_<classname>` par défaut. Parfois, vous ne voulez pas l'utiliser. Si vous souhaitez modifier le nom par défaut, vous pouvez annoncer le nom de la table en définissant la `db_table` dans la classe `Meta` :

```
from django.db import models

class YourModel(models.Model):
    parms = models.CharField()
    class Meta:
        db_table = "custom_table_name"
```

Si vous voulez voir quel code SQL sera exécuté par une certaine migration, lancez simplement cette commande:

```
python manage.py sqlmigrate <app_label> <migration_number>
```

Django> 1.10
La nouvelle option `makemigrations --check` permet à la commande de sortir avec un statut différent de zéro lorsque des modifications du modèle sans migration sont détectées.

Voir Migrations pour plus de détails sur les migrations.

Créer un modèle avec des relations

Relation plusieurs à un

```
from django.db import models

class Author(models.Model):
   name = models.CharField(max_length=50)

#Book has a foreignkey (many to one) relationship with author
class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    publish_date = models.DateField()
```

Option la plus générique. Peut être utilisé partout où vous souhaitez représenter une relation

Relation plusieurs à plusieurs

```
class Topping(models.Model):
    name = models.CharField(max_length=50)

# One pizza can have many toppings and same topping can be on many pizzas
class Pizza(models.Model):
    name = models.CharField(max_length=50)
    toppings = models.ManyToManyField(Topping)
```

En interne, cela est représenté via une autre table. Et `ManyToManyField` devrait être mis sur des modèles qui seront édités sur un formulaire. Par exemple: `Appointment` - `Appointment` aura un `ManyToManyField` appelé `Customer` , `Pizza` a `Toppings` et ainsi de suite.

Relation plusieurs-à-plusieurs utilisant des cours

```
class Service(models.Model):
     name = models.CharField(max_length=35)

class Client(models.Model):
    name = models.CharField(max_length=35)
    age = models.IntegerField()
    services = models.ManyToManyField(Service, through='Subscription')

class Subscription(models.Model):
     client = models.ForeignKey(Client)
     service = models.ForeignKey(Service)
     subscription_type = models.CharField(max_length=1, choices=SUBSCRIPTION_TYPES)
     created_at = models.DateTimeField(default=timezone.now)
```

De cette façon, nous pouvons réellement conserver plus de métadonnées sur une relation entre deux entités. Comme on peut le voir, un client peut être abonné à plusieurs services via plusieurs types d'abonnement. La seule différence dans ce cas est que pour ajouter de nouvelles instances à la relation M2M, on ne peut pas utiliser la méthode de raccourci `pizza.toppings.add(topping)` , mais un nouvel objet de la classe *through* doit être créé, `Subscription.objects.create(client=client, service=service, subscription_type='p')`

> Dans d'autres langues, les `through tables` sont également connues sous le nom de `Intersection table JoinColumn` , `Intersection table` ou `mapping table`

Relation individuelle

```
class Employee(models.Model):
   name = models.CharField(max_length=50)
   age = models.IntegerField()
   spouse = models.OneToOneField(Spouse)

class Spouse(models.Model):
   name = models.CharField(max_length=50)
```

Utilisez ces champs lorsque vous n'aurez qu'une relation de composition entre les deux modèles.

Requêtes de base sur la base de données Django

Django ORM est une abstraction puissante qui vous permet de stocker et d'extraire des données de la base de données sans écrire vous-même des requêtes SQL.

Supposons les modèles suivants:

```
class Author(models.Model):
   name = models.CharField(max_length=50)

class Book(models.Model):
   name = models.CharField(max_length=50)
   author = models.ForeignKey(Author)
```

En supposant que vous avez ajouté le code ci-dessus à une application django et exécutez la commande `migrate` (afin que votre base de données soit créée). Démarrer le shell Django par

```
python manage.py shell
```

Cela démarre le shell python standard mais avec les bibliothèques Django pertinentes importées, afin que vous puissiez vous concentrer directement sur les parties importantes.

Commencez par importer les modèles que nous venons de définir (je suppose que cela se fait dans un fichier `models.py`)

```
from .models import Book, Author
```

Exécutez votre première requête de sélection:

```
>>> Author.objects.all()
[]
>>> Book.objects.all()
[]
```

Permet de créer un objet auteur et livre:

```
>>> hawking = Author(name="Stephen hawking")
>>> hawking.save()
>>> history_of_time = Book(name="history of time", author=hawking)
>>> history_of_time.save()
```

ou utilisez la fonction create pour créer des objets de modèle et les enregistrer dans un code de ligne

```
>>> wings_of_fire = Book.objects.create(name="Wings of Fire", author="APJ Abdul Kalam")
```

Maintenant permet de lancer la requête

```
>>> Book.objects.all()
[<Book: Book object>]
>>> book = Book.objects.first() #getting the first book object
>>> book.name
u'history of time'
```

Ajoutons une clause where à notre requête select

```
>>> Book.objects.filter(name='nothing')
[]
>>> Author.objects.filter(name__startswith='Ste')
[<Author: Author object>]
```

Pour obtenir des détails sur l'auteur d'un livre donné

```
>>> book = Book.objects.first() #getting the first book object
>>> book.author.name # lookup on related model
u'Stephen hawking'
```

Pour obtenir tous les livres publiés par Stephen Hawking (Lookup book par son auteur)

```
>>> hawking.book_set.all()
[<Book: Book object>]
```

`_set` est la notation utilisée pour les "recherches inversées", c'est-à-dire que lorsque le champ de recherche est sur le modèle Book, nous pouvons utiliser `book_set` sur un objet auteur pour obtenir tous ses livres.

Une table non gérée de base.

À un certain moment de votre utilisation de Django, vous pourriez vouloir interagir avec des tables déjà créées ou avec des vues de base de données. Dans ces cas, vous ne voudriez pas que Django gère les tables lors de ses migrations. Pour configurer cela, vous devez ajouter une seule variable à la classe `Meta` votre modèle: `managed = False` .

Voici un exemple de la façon dont vous pouvez créer un modèle non géré pour interagir avec une vue de base de données:

```
class Dummy(models.Model):
    something = models.IntegerField()

    class Meta:
       managed = False
```

Cela peut être mappé sur une vue définie dans SQL comme suit.

```
CREATE VIEW myapp_dummy AS
SELECT id, something FROM complicated_table
WHERE some_complicated_condition = True
```

Une fois ce modèle créé, vous pouvez l'utiliser comme n'importe quel autre modèle:

```
>>> Dummy.objects.all()
[<Dummy: Dummy object>, <Dummy: Dummy object>, <Dummy: Dummy object>]
>>> Dummy.objects.filter(something=42)
[<Dummy: Dummy object>]
```

Modèles avancés

Un modèle peut fournir beaucoup plus d'informations que les seules données relatives à un objet. Voyons un exemple et décomposons-le en ce qui est utile pour:

```
from django.db import models
from django.urls import reverse
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class Book(models.Model):
    slug = models.SlugField()
    title = models.CharField(max_length=128)
    publish_date = models.DateField()

    def get_absolute_url(self):
        return reverse('library:book', kwargs={'pk':self.pk})

    def __str__(self):
        return self.title

    class Meta:
        ordering = ['publish_date', 'title']
```

Clé primaire automatique

Vous remarquerez peut-être l'utilisation de `self.pk` dans la méthode `get_absolute_url` . Le champ `pk` est un alias de la clé primaire d'un modèle. De plus, django ajoutera automatiquement une clé primaire si elle manque. C'est une chose de moins à s'inquiéter et vous permet de définir la clé étrangère sur n'importe quel modèle et de l'obtenir facilement.

URL absolue

La première fonction définie est `get_absolute_url` . De cette façon, si vous avez un livre, vous pouvez obtenir un lien vers celui-ci sans manipuler la balise URL, la résolution, l'attribut, etc. Appelez simplement `book.get_absolute_url` et vous obtenez le bon lien. En bonus, votre objet dans l'administrateur de django gagnera un bouton "voir sur le site".

Représentation de chaîne

Avoir une méthode `__str__` vous permet d'utiliser l'objet lorsque vous avez besoin de l'afficher. Par exemple, avec la méthode précédente, ajouter un lien au livre dans un modèle est aussi simple que `<a href="{{ book.get_absolute_url }}">{{ book }}</a>` . Droit au but. Cette méthode contrôle également ce qui est affiché dans la liste déroulante admin, par exemple pour une clé étrangère.

Le décorateur de classe vous permet de définir la méthode une fois pour `__str__` et `__unicode__` sur python 2 tout en ne causant aucun problème sur python 3. Si vous souhaitez que votre application s'exécute sur les deux versions, c'est la voie à suivre.

Terrain de limaces

Le champ slug est similaire à un champ char mais accepte moins de symboles. Par défaut, seules les lettres, les chiffres, les traits de soulignement ou les tirets. C'est utile si vous voulez identifier un objet en utilisant une belle représentation, par exemple dans url.

La classe Meta

La classe `Meta` nous permet de définir beaucoup plus d'informations sur l'ensemble de la collection. Ici, seul le classement par défaut est défini. C'est utile avec l'objet ListView par exemple. Il faut idéalement une courte liste de champs à utiliser pour le tri. Ici, le livre sera trié d'abord par date de publication puis par titre si la date est la même.

Les autres attributs fréquents sont `verbose_name` et `verbose_name_plural` . Par défaut, ils sont générés à partir du nom du modèle et devraient convenir. Mais la forme plurielle est naïve, en ajoutant simplement un «s» au singulier, de sorte que vous pourriez vouloir le définir explicitement dans certains cas.

Valeurs calculées

Une fois qu'un objet de modèle a été extrait, il devient une instance entièrement réalisée de la classe. En tant que tel, toutes les méthodes supplémentaires sont accessibles dans des formulaires et des sérialiseurs (comme Django Rest Framework).

L'utilisation des propriétés python est un moyen élégant de représenter des valeurs supplémentaires qui ne sont pas stockées dans la base de données en raison de circonstances variables.

```
def expire():
    return timezone.now() + timezone.timedelta(days=7)
```

```
class Coupon(models.Model):
    expiration_date = models.DateField(default=expire)

    @property
    def is_expired(self):
        return timezone.now() > self.expiration_date
```

Alors que la plupart des cas, vous pouvez compléter les données avec des annotations sur vos ensembles de requêtes, les valeurs calculées en tant que propriétés de modèle sont idéales pour les calculs qui ne peuvent pas être évalués simplement dans le cadre d'une requête.

De plus, les propriétés, puisqu'elles sont déclarées dans la classe python et non dans le schéma, ne sont pas disponibles pour la recherche.

Ajout d'une représentation sous forme de chaîne d'un modèle

Pour créer une présentation lisible par un humain d'un objet de modèle, vous devez implémenter la méthode `Model.__str__()` (ou `Model.__unicode__()` sur python2). Cette méthode sera appelée à chaque fois que vous appelez `str()` sur une instance de votre modèle (y compris, par exemple, lorsque le modèle est utilisé dans un modèle). Voici un exemple:

1. Créez un modèle de livre.

```
# your_app/models.py

from django.db import models

class Book(models.Model):
    name = models.CharField(max_length=50)
    author = models.CharField(max_length=50)
```

2. Créez une instance du modèle et enregistrez-la dans la base de données:

```
>>> himu_book = Book(name='Himu Mama', author='Humayun Ahmed')
>>> himu_book.save()
```

3. Exécutez `print()` sur l'instance:

```
>>> print(himu_book)
<Book: Book object>
```

<Book: Objet livre> , la sortie par défaut, ne nous aide pas. Pour corriger cela, ajoutons une méthode `__str__` .

```
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class Book(models.Model):
    name = models.CharField(max_length=50)
    author = models.CharField(max_length=50)
```

```
    def __str__(self):
        return '{} by {}'.format(self.name, self.author)
```

Notez que le décorateur `python_2_unicode_compatible` n'est nécessaire que si vous voulez que votre code soit compatible avec python 2. Ce décorateur copie la méthode `__str__` pour créer une méthode `__unicode__` . Importez-le depuis `django.utils.encoding` .

Maintenant, si nous appelons la fonction d'impression, l'instance du livre à nouveau:

```
>>> print(himu_book)
Himu Mama by Humayun Ahmed
```

Beaucoup mieux!

La représentation sous forme de chaîne est également utilisée lorsque le modèle est utilisé dans les `ModelForm` for `ForeignKeyField` et `ManyToManyField` .

Model mixins

Dans les mêmes cas, différents modèles peuvent avoir les mêmes champs et les mêmes procédures dans le cycle de vie du produit. Pour gérer ces similarités sans avoir à répéter le code, on pourrait utiliser l'héritage. Au lieu d'hériter une classe entière, le modèle de conception de **mixin** nous permet d'hériter (*ou d'inclure* certains) des méthodes et des attributs. Regardons un exemple:

```
class PostableMixin(models.Model):
    class Meta:
        abstract=True

    sender_name = models.CharField(max_length=128)
    sender_address = models.CharField(max_length=255)
    receiver_name = models.CharField(max_length=128)
    receiver_address = models.CharField(max_length=255)
    post_datetime = models.DateTimeField(auto_now_add=True)
    delivery_datetime = models.DateTimeField(null=True)
    notes = models.TextField(max_length=500)

class Envelope(PostableMixin):
    ENVELOPE_COMMERCIAL = 1
    ENVELOPE_BOOKLET = 2
    ENVELOPE_CATALOG = 3

    ENVELOPE_TYPES = (
        (ENVELOPE_COMMERCIAL, 'Commercial'),
        (ENVELOPE_BOOKLET, 'Booklet'),
        (ENVELOPE_CATALOG, 'Catalog'),
    )

    envelope_type = models.PositiveSmallIntegerField(choices=ENVELOPE_TYPES)

class Package(PostableMixin):
    weight = models.DecimalField(max_digits=6, decimal_places=2)
    width = models.DecimalField(max_digits=5, decimal_places=2)
    height = models.DecimalField(max_digits=5, decimal_places=2)
```

```
    depth = models.DecimalField(max_digits=5, decimal_places=2)
```

Pour transformer un modèle en une classe abstraite, vous devez mentionner `abstract=True` dans sa classe `Meta` interne. Django ne crée aucune table pour les modèles abstraits de la base de données. Cependant, pour les modèles `Envelope` et `Package` , les tables correspondantes seraient créées dans la base de données.

En plus des champs, certaines méthodes de modélisation seront nécessaires sur plusieurs modèles. Ainsi, ces méthodes pourraient être ajoutées aux mixins pour empêcher la répétition du code. Par exemple, si nous créons une méthode pour définir la date de livraison sur `PostableMixin` elle sera accessible à partir de ses deux enfants:

```
class PostableMixin(models.Model):
    class Meta:
        abstract=True

    ...
    ...

    def set_delivery_datetime(self, dt=None):
        if dt is None:
            from django.utils.timezone import now
            dt = now()

        self.delivery_datetime = dt
        self.save()
```

Cette méthode pourrait être utilisée comme suit sur les enfants:

```
>> envelope = Envelope.objects.get(pk=1)
>> envelope.set_delivery_datetime()

>> pack = Package.objects.get(pk=1)
>> pack.set_delivery_datetime()
```

Clé primaire UUID

Un modèle par défaut utilisera une clé primaire auto-incrémentée (entier). Cela vous donnera une séquence de touches 1, 2, 3.

Différents types de clé primaire peuvent être définis sur un modèle avec une petite modification du modèle.

Un UUID est un identifiant unique, il s’agit d’un identifiant aléatoire de 32 caractères qui peut être utilisé comme identifiant. C'est une bonne option à utiliser lorsque vous ne souhaitez pas que des identifiants séquentiels soient affectés à des enregistrements de votre base de données. Utilisée sur PostgreSQL, cette option stocke un type de données uuid, sinon dans un caractère (32).

```
import uuid
from django.db import models

class ModelUsingUUID(models.Model):
```

```
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
```

La clé générée sera au format `7778c552-73fc-4bc4-8bf9-5a2f6f7b7f47`

Héritage

L'héritage entre les modèles peut se faire de deux manières:

- une classe abstraite commune (voir l'exemple "Model mixins")
- un modèle commun avec plusieurs tables

L'héritage multi-tables créera une table pour les champs communs et un pour chaque exemple de modèle enfant:

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

créera 2 tables, une pour `Place` et une pour `Restaurant` avec un champ `OneToOne` masqué à `Place` pour les champs communs.

Notez que cela nécessitera une requête supplémentaire sur les tables de lieux chaque fois que vous récupérerez un objet de restaurant.

Lire Des modèles en ligne: https://riptutorial.com/fr/django/topic/888/des-modeles

Chapitre 15: Des vues

Introduction

Une fonction de vue, ou vue en abrégé, est simplement une fonction Python qui prend une requête Web et renvoie une réponse Web. -Django Documentation-

Examples

[Introduction] Simple View (Hello World Equivalent)

Créons une vue très simple pour répondre à un modèle "Hello World" au format HTML.

1. Pour ce faire, allez dans `my_project/my_app/views.py` (Ici nous `my_project/my_app/views.py` nos fonctions de vue) et ajoutez la vue suivante:

    ```
    from django.http import HttpResponse

    def hello_world(request):
        html = "<html><title>Hello World!</title><body>Hello World!</body></html>"
        return HttpResponse(html)
    ```

2. Pour appeler cette vue, nous devons configurer un modèle d'URL dans `my_project/my_app/urls.py` :

    ```
    from django.conf.urls import url

    from . import views

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

3. Démarrer le serveur: `python manage.py runserver`

    Maintenant, si nous `http://localhost:8000/hello_world/` , notre template (la chaîne html) sera rendu dans notre navigateur.

Lire Des vues en ligne: https://riptutorial.com/fr/django/topic/7490/des-vues

Chapitre 16: Django à partir de la ligne de commande.

Remarques

Bien que Django soit principalement destiné aux applications Web, il dispose d'un ORM puissant et facile à utiliser qui peut également être utilisé pour les applications et les scripts en ligne de commande. Deux approches différentes peuvent être utilisées. La première consiste à créer une commande de gestion personnalisée et la seconde à initialiser l'environnement Django au début de votre script.

Examples

Django à partir de la ligne de commande.

En supposant que vous ayez configuré un projet django et que le fichier de paramètres se trouve dans une application nommée main, vous initialiserez votre code

```
import os, sys

# Setup environ
sys.path.append(os.getcwd())
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "main.settings")

# Setup django
import django
django.setup()

# rest of your imports go here

from main.models import MyModel

# normal python code that makes use of Django models go here

for obj in MyModel.objects.all():
    print obj
```

Ce qui précède peut être exécuté comme

```
python main/cli.py
```

Lire Django à partir de la ligne de commande. en ligne:
https://riptutorial.com/fr/django/topic/5848/django-a-partir-de-la-ligne-de-commande-

Chapitre 17: Django et les réseaux sociaux

Paramètres

Réglage	Est-ce que
Quelques configurations	Paramètres de base pratiques qui vont avec Django-Allauth (que j'utilise la plupart du temps). Pour plus d'options de configuration, voir Configurations
ACCOUNT_AUTHENTICATION_METHOD (= "nom d'utilisateur" ou "email" ou "username_email")	Spécifie la méthode de connexion à utiliser - que l'utilisateur se connecte en saisissant son nom d'utilisateur, son adresse électronique ou l'un des deux. Définir cette option sur «email» requiert ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_EMAIL_CONFIRMATION_EXPIRE_DAYS (= 3)	Détermine la date d'expiration des e-mails de confirmation par e-mail (nombre de jours).
ACCOUNT_EMAIL_REQUIRED (= Faux)	L'utilisateur est tenu de remettre une adresse e-mail lors de l'inscription. Cela va de pair avec le paramètre `ACCOUNT_AUTHENTICATION_METHOD`
ACCOUNT_EMAIL_VERIFICATION (= "optionnel")	Détermine la méthode de vérification du courrier électronique lors de l'inscription - choisissez l'une des options "obligatoire", "facultative" ou "aucune". Lorsqu'il est défini sur «obligatoire», l'utilisateur est bloqué pour se connecter jusqu'à ce que l'adresse e-mail soit vérifiée. Choisissez «optionnel» ou «aucun» pour autoriser les connexions avec une adresse e-mail non vérifiée. En cas de «facultatif», le courrier de vérification du courrier électronique est toujours envoyé, alors qu'en cas de «aucun», aucun courrier électronique de vérification du courrier électronique n'est envoyé.
ACCOUNT_LOGIN_ATTEMPTS_LIMIT (= 5)	Nombre de tentatives de connexion infructueuses. Lorsque ce nombre est dépassé, il est interdit à l'utilisateur de se connecter aux secondes

Réglage	Est-ce que
	ACCOUNT_LOGIN_ATTEMPTS_TIMEOUT spécifiées. Bien que cela protège la vue de connexion allauth, cela ne protège pas le login admin de Django d'être forcé.
ACCOUNT_LOGOUT_ON_PASSWORD_CHANGE (= Faux)	Détermine si l'utilisateur est automatiquement déconnecté après avoir modifié ou défini son mot de passe.
SOCIALACCOUNT_PROVIDERS (= dict)	Dictionnaire contenant les paramètres spécifiques au fournisseur.

Examples

Moyen facile: python-social-auth

python-social-auth est un framework qui simplifie le mécanisme d'authentification et d'autorisation sociale. Il contient de nombreux backends sociaux (Facebook, Twitter, Github, LinkedIn, etc.)

INSTALLER

Nous devons d'abord installer le paquet python-social-auth avec

```
pip install python-social-auth
```

ou téléchargez le code depuis github. C'est le bon moment pour l'ajouter à votre fichier `requirements.txt` .

CONFIGURER settings.py

Dans les paramètres.py ajoutez:

```
INSTALLED_APPS = (
    ...
    'social.apps.django_app.default',
    ...
)
```

CONFIGURER LES SAUVEGARDES

AUTHENTICATION_BACKENDS contient les backends que nous allons utiliser, et il suffit de mettre ce dont nous avons besoin.

```
AUTHENTICATION_BACKENDS = (
    'social.backends.open_id.OpenIdAuth',
    'social.backends.google.GoogleOpenId',
```

```
    'social.backends.google.GoogleOAuth2',
    'social.backends.google.GoogleOAuth',
    'social.backends.twitter.TwitterOAuth',
    'social.backends.yahoo.YahooOpenId',
    ...
    'django.contrib.auth.backends.ModelBackend',
)
```

Votre fichier `settings.py` n'a peut-être pas encore de champ `AUTHENTICATION_BACKENDS` . Si tel est le cas, ajoutez le champ. Veillez à ne pas manquer `'django.contrib.auth.backends.ModelBackend'`, car il gère la connexion par nom d'utilisateur / mot de passe.

Si nous utilisons par exemple Facebook et Linkedin Backends, nous devons ajouter les clés API

```
SOCIAL_AUTH_FACEBOOK_KEY = 'YOURFACEBOOKKEY'
SOCIAL_AUTH_FACEBOOK_SECRET = 'YOURFACEBOOKSECRET'
```

et

```
SOCIAL_AUTH_LINKEDIN_KEY = 'YOURLINKEDINKEY'
SOCIAL_AUTH_LINKEDIN_SECRET = 'YOURLINKEDINSECRET'
```

Note : Vous pouvez obtenir les clés intégrées dans les développeurs Facebook et les développeurs Linkedin et vous pouvez voir ici la liste complète et sa manière respective de spécifier la clé API et la clé Secret.

Remarque sur les clés secrètes: Les clés secrètes doivent être gardées secrètes. Voici une explication de Stack Overflow qui est utile. Ce tutoriel est utile pour apprendre sur les variables environnementales.

TEMPLATE_CONTEXT_PROCESSORS aidera aux redirections, backends et autres choses, mais au début, nous avons seulement besoin de ceux-ci:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'social.apps.django_app.context_processors.backends',
    'social.apps.django_app.context_processors.login_redirect',
    ...
)
```

Dans Django 1.8, la configuration de `TEMPLATE_CONTEXT_PREPROCESSORS` comme indiqué ci-dessus était obsolète. Si c'est le cas pour vous, vous l'ajouterez à l'intérieur du dict `TEMPLATES` . Le vôtre devrait ressembler à ceci:

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

```
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'social.apps.django_app.context_processors.backends',
                'social.apps.django_app.context_processors.login_redirect',
            ],
        },
    },
]
```

UTILISATION D'UN UTILISATEUR PERSONNALISÉ

Si vous utilisez un modèle utilisateur personnalisé et souhaitez vous y associer, ajoutez simplement la ligne suivante (toujours dans **settings.py**)

```
SOCIAL_AUTH_USER_MODEL = 'somepackage.models.CustomUser'
```

`CustomUser` est un modèle qui hérite ou abstrait de l'utilisateur par défaut.

CONFIGURER urls.py

```
# if you haven't imported inlcude make sure you do so at the top of your file
from django.conf.urls import url, include

urlpatterns = patterns('',
    ...
    url('', include('social.apps.django_app.urls', namespace='social'))
    ...
)
```

Ensuite, vous devez synchroniser la base de données pour créer les modèles nécessaires:

```
./manage.py migrate
```

Enfin on peut jouer!

Dans certains modèles, vous devez ajouter quelque chose comme ceci:

```
    <a href="{% url 'social:begin' 'facebook' %}?next={{ request.path }}">Login with
Facebook</a>
    <a href="{% url 'social:begin' 'linkedin' %}?next={{ request.path }}">Login with
Linkedin</a>
```

Si vous utilisez un autre backend, changez simplement "facebook" par le nom du backend.

Déconnexion des utilisateurs

Une fois que vous avez connecté les utilisateurs, vous voudrez probablement créer la fonctionnalité pour les déconnecter. Dans certains modèles, près de l'endroit où le modèle de connexion a été affiché, ajoutez la balise suivante:

```
<a href="{% url 'logout' %}">Logout</a>
```

ou

```
<a href="/logout">Logout</a>
```

Vous voudrez modifier votre fichier `urls.py` avec un code similaire à:

```
url(r'^logout/$', views.logout, name='logout'),
```

Enfin, modifiez votre fichier views.py avec un code similaire à:

```
def logout(request):
    auth_logout(request)
    return redirect('/')
```

Utiliser Django Allauth

Pour tous mes projets, Django-Allauth est resté facile à installer et propose de nombreuses fonctionnalités, notamment:

- Plus de 50 authentifications de réseaux sociaux
- Inscription mixte des comptes locaux et sociaux
- Comptes sociaux multiples
- Inscription instantanée facultative pour les comptes sociaux - pas de questions posées
- Gestion des adresses e-mail (plusieurs adresses e-mail, définition d'un élément principal)
- Flux de mot de passe oublié Flux de vérification de l'adresse de messagerie

Si vous êtes intéressé à vous salir les mains, Django-Allauth échappe à la tâche, avec des configurations supplémentaires pour modifier le processus et l'utilisation de votre système d'authentification.

Les étapes ci-dessous supposent que vous utilisez Django 1.10+

Étapes de configuration:

```
pip install django-allauth
```

Dans votre fichier `settings.py` , apportez les modifications suivantes:

```
# Specify the context processors as follows:
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # Already defined Django-related contexts here

                # `allauth` needs this from django. It is there by default,
                # unless you've devilishly taken it away.
                'django.template.context_processors.request',
            ],
```

```
        },
    },
]

AUTHENTICATION_BACKENDS = (
    # Needed to login by username in Django admin, regardless of `allauth`
    'django.contrib.auth.backends.ModelBackend',

    # `allauth` specific authentication methods, such as login by e-mail
    'allauth.account.auth_backends.AuthenticationBackend',
)

INSTALLED_APPS = (
# Up here is all your default installed apps from Django

# The following apps are required:
'django.contrib.auth',
'django.contrib.sites',

'allauth',
'allauth.account',
'allauth.socialaccount',

# include the providers you want to enable:
'allauth.socialaccount.providers.google',
'allauth.socialaccount.providers.facebook',
)

# Don't forget this little dude.
SITE_ID = 1
```

Fait avec les modifications dans le fichier `settings.py` ci-dessus, déplacez-vous sur le fichier `urls.py` Il peut s'agir de `yourapp/urls.py` ou de votre nom de projet `ProjectName/urls.py` Normalement, je préfère le nom du projet `ProjectName/urls.py`

```
urlpatterns = [
    # other urls here
    url(r'^accounts/', include('allauth.urls')),
    # other urls here
]
```

En ajoutant simplement l' `include('allauth.urls')` , vous obtenez ces URL gratuitement:

```
^accounts/ ^ ^signup/$ [name='account_signup']
^accounts/ ^ ^login/$ [name='account_login']
^accounts/ ^ ^logout/$ [name='account_logout']
^accounts/ ^ ^password/change/$ [name='account_change_password']
^accounts/ ^ ^password/set/$ [name='account_set_password']
^accounts/ ^ ^inactive/$ [name='account_inactive']
^accounts/ ^ ^email/$ [name='account_email']
^accounts/ ^ ^confirm-email/$ [name='account_email_verification_sent']
^accounts/ ^ ^confirm-email/(?P<key>[-:\w]+)/$ [name='account_confirm_email']
^accounts/ ^ ^password/reset/$ [name='account_reset_password']
^accounts/ ^ ^password/reset/done/$ [name='account_reset_password_done']
^accounts/ ^ ^password/reset/key/(?P<uidb36>[0-9A-Za-z]+)-(?P<key>.+)/$
[name='account_reset_password_from_key']
^accounts/ ^ ^password/reset/key/done/$ [name='account_reset_password_from_key_done']
^accounts/ ^social/
```

```
^accounts/ ^google/
^accounts/ ^twitter/
^accounts/ ^facebook/
^accounts/ ^facebook/login/token/$ [name='facebook_login_by_token']
```

Enfin, faites `python ./manage.py migrate` pour valider les migrations de Django-allauth dans Database.

Comme d'habitude, pour pouvoir vous connecter à votre application en utilisant n'importe quel réseau social que vous avez ajouté, vous devrez ajouter les détails du compte social du réseau.

Connectez-vous à l'administrateur Django (`localhost:8000/admin`) et sous `Social Applications` dans les détails de votre compte social.

Vous aurez peut-être besoin de comptes sur chaque fournisseur d'authentification pour obtenir des détails à remplir dans les sections Applications sociales.

Pour des configurations détaillées de ce que vous pouvez avoir et modifier, voir la page Configurations .

Lire Django et les réseaux sociaux en ligne: https://riptutorial.com/fr/django/topic/4743/django-et-les-reseaux-sociaux

Chapitre 18: Django Rest Framework

Examples

API simple en lecture seule

En supposant que vous ayez un modèle qui ressemble à ce qui suit, nous allons nous lancer dans une simple API en **lecture seule** barebone pilotée par Django REST Framework ("DRF").

models.py

```
class FeedItem(models.Model):
    title = models.CharField(max_length=100, blank=True)
    url = models.URLField(blank=True)
    style = models.CharField(max_length=100, blank=True)
    description = models.TextField(blank=True)
```

Le sérialiseur est le composant qui prendra toutes les informations du modèle Django (dans ce cas, le `FeedItem`) et le transformera en JSON. Il est très similaire à la création de classes de formulaires dans Django. Si vous avez de l'expérience à ce sujet, ce sera très confortable pour vous.

serializers.py

```
from rest_framework import serializers
from . import models

class FeedItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.FeedItem
        fields = ('title', 'url', 'description', 'style')
```

views.py

DRF offre de nombreuses classes d'affichage pour gérer une variété de cas d'utilisation. Dans cet exemple, nous allons seulement avoir une API en **lecture seule** , donc, plutôt que d'utiliser un ensemble de vues plus complet ou un ensemble de vues génériques associées, nous utiliserons une seule sous-classe de `ListAPIView` de DRF.

Le but de cette classe est de lier les données au sérialiseur et de les regrouper pour obtenir un objet réponse.

```
from rest_framework import generics
from . import serializers, models

class FeedItemList(generics.ListAPIView):
    serializer_class = serializers.FeedItemSerializer
    queryset = models.FeedItem.objects.all()
```

urls.py

Assurez-vous de diriger votre itinéraire vers votre vue DRF.

```
from django.conf.urls import url
from . import views

urlpatterns = [
    ...
    url(r'path/to/api', views.FeedItemList.as_view()),
]
```

Lire Django Rest Framework en ligne: https://riptutorial.com/fr/django/topic/7341/django-rest-framework

Chapitre 19: django-filter

Examples

Utilisez django-filter avec CBV

`django-filter` est un système générique de filtrage de QuangoSets basé sur les sélections des utilisateurs. La documentation l' utilise dans une vue basée sur les fonctions en tant que modèle de produit:

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField()
    description = models.TextField()
    release_date = models.DateField()
    manufacturer = models.ForeignKey(Manufacturer)
```

Le filtre sera comme suit:

```
import django_filters

class ProductFilter(django_filters.FilterSet):
    name = django_filters.CharFilter(lookup_expr='iexact')

    class Meta:
        model = Product
        fields = ['price', 'release_date']
```

Pour utiliser cela dans une CBV, remplacez `get_queryset()` du ListView, puis retournez le `querset` filtré:

```
from django.views.generic import ListView
from .filters import ProductFilter

class ArticleListView(ListView):
    model = Product

    def get_queryset(self):
        qs = self.model.objects.all()
        product_filtered_list = ProductFilter(self.request.GET, queryset=qs)
        return product_filtered_list.qs
```

Il est possible d'accéder aux objets filtrés dans vos vues, par exemple avec la pagination, dans `f.qs` Cela va paginer la liste des objets filtrés.

Lire django-filter en ligne: https://riptutorial.com/fr/django/topic/6101/django-filter

Chapitre 20: Enregistrement

Examples

Connexion au service Syslog

Il est possible de configurer Django pour générer un journal vers un service syslog local ou distant. Cette configuration utilise le SysLogHandler intégré à python.

```
from logging.handlers import SysLogHandler
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'standard': {
            'format' : "[YOUR PROJECT NAME] [%(asctime)s] %(levelname)s [%(name)s:%(lineno)s]
%(message)s",
            'datefmt' : "%d/%b/%Y %H:%M:%S"
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
        'syslog': {
            'class': 'logging.handlers.SysLogHandler',
            'formatter': 'standard',
            'facility': 'user',
            # uncomment next line if rsyslog works with unix socket only (UDP reception
disabled)
            #'address': '/dev/log'
        }
    },
    'loggers': {
        'django':{
            'handlers': ['syslog'],
            'level': 'INFO',
            'disabled': False,
            'propagate': True
        }
    }
}

# loggers for my apps, uses INSTALLED_APPS in settings
# each app must have a configured logger
# level can be changed as desired: DEBUG, INFO, WARNING...
MY_LOGGERS = {}
for app in INSTALLED_APPS:
    MY_LOGGERS[app] = {
        'handlers': ['syslog'],
        'level': 'DEBUG',
        'propagate': True,
    }
LOGGING['loggers'].update(MY_LOGGERS)
```

Configuration de base de Django

En interne, Django utilise le système de journalisation Python. Il existe plusieurs façons de configurer la journalisation d'un projet. Voici une base:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': "[%(asctime)s] %(levelname)s [%(name)s:%(lineno)s] %(message)s",
            'datefmt': "%Y-%m-%d %H:%M:%S"
        },
    },
    'handlers': {
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'default'
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
            'propagate': True,
            'level': 'INFO',
        },
    }
}
```

Les formateurs

Il peut être utilisé pour configurer l'apparence des journaux lorsqu'ils sont imprimés sur la sortie. Vous pouvez définir de nombreux formateurs en définissant une chaîne de clé pour chaque formateur. Un formateur est ensuite utilisé lors de la déclaration d'un gestionnaire.

Manieurs

Peut être utilisé pour configurer où les journaux seront imprimés. Dans l'exemple ci-dessus, ils sont envoyés à stdout et à stderr. Il existe différentes classes de gestionnaires:

```
'rotated_logs': {
    'class': 'logging.handlers.RotatingFileHandler',
    'filename': '/var/log/my_project.log',
    'maxBytes': 1024 * 1024 * 5,  # 5 MB
    'backupCount': 5,
    'formatter': 'default'
    'level': 'DEBUG',
},
```

Cela produira des logs dans le fichier tergeted par `filename` . Dans cet exemple, un nouveau fichier journal sera créé lorsque la taille actuelle atteindra la taille de 5 Mo (l’ancien sera renommé en my_project.log.1) et les 5 derniers fichiers seront conservés pour l’archivage.

```
'mail_admins': {
    'level': 'ERROR',
    'class': 'django.utils.log.AdminEmailHandler'
},
```

Cela enverra chaque journal par eamil aux utilisateurs spécifiés dans la variable de réglage `ADMINS` . Le niveau est défini sur `ERROR` , de sorte que seuls les journaux avec le niveau `ERROR` seront envoyés par courrier électronique. Ceci est extrêmement utile pour rester informé sur les erreurs potentielles 50x sur un serveur de production.

D'autres gestionnaires peuvent être utilisés avec Django. Pour une liste complète, veuillez lire la documentation correspondante. Comme pour les formateurs, vous pouvez définir plusieurs gestionnaires dans un même projet, définissant pour chacun une chaîne de clé différente. Chaque gestionnaire peut être utilisé dans un enregistreur spécifique.

Bûcherons

Dans `LOGGING` , la dernière partie configure pour chaque module le niveau de consignation minimal, le ou les gestionnaires à utiliser, etc.

Lire Enregistrement en ligne: https://riptutorial.com/fr/django/topic/1231/enregistrement

Chapitre 21: Expressions F ()

Introduction

Une expression F () est un moyen pour Django d'utiliser un objet Python pour faire référence à la valeur du champ modèle ou de la colonne annotée dans la base de données sans avoir à extraire la valeur dans la mémoire Python. Cela permet aux développeurs d'éviter certaines conditions de concurrence et de filtrer les résultats en fonction des valeurs du champ du modèle.

Syntaxe

- à partir de django.db.models import F

Examples

Éviter les conditions de course

Voir cette question si vous ne savez pas quelles sont les conditions de course.

Le code suivant peut être soumis à des conditions de course:

```
article = Article.objects.get(pk=69)
article.views_count += 1
article.save()
```

Si `views_count` est égal à `1337` , cela entraînera une telle requête:

```
UPDATE app_article SET views_count = 1338 WHERE id=69
```

Si deux clients accèdent à cet article en même temps, *il* se *peut* que la deuxième requête HTTP exécute `Article.objects.get(pk=69)` avant que le premier exécute `article.save()` . Ainsi, les deux requêtes auront `views_count = 1337` , incrémentez-le et enregistrez `views_count = 1338` dans la base de données, alors qu'il devrait en fait être `1339` .

Pour résoudre ce problème, utilisez une expression `F()` :

```
article = Article.objects.get(pk=69)
article.views_count = F('views_count') + 1
article.save()
```

Cela, d'autre part, se traduira par une telle requête:

```
UPDATE app_article SET views_count = views_count + 1 WHERE id=69
```

Mise à jour du jeu de requête en vrac

Supposons que nous voulions supprimer 2 relevés supérieurs de tous les articles de l'auteur avec l'ID `51` .
Faire cela uniquement avec Python exécuterait `N` requêtes (`N` étant le nombre d'articles dans le jeu de requête):

```
for article in Article.objects.filter(author_id=51):
    article.upvotes -= 2
    article.save()
    # Note that there is a race condition here but this is not the focus
    # of this example.
```

Que faire si, au lieu d'extraire tous les articles en Python, de les parcourir en boucle, de réduire les relevés et de sauvegarder chaque mise à jour dans la base de données, il y avait un autre moyen?
En utilisant une expression `F()` , vous pouvez le faire en une seule requête:

```
Article.objects.filter(author_id=51).update(upvotes=F('upvotes') - 2)
```

Qui peut être traduit dans la requête SQL suivante:

```
UPDATE app_article SET upvotes = upvotes - 2 WHERE author_id = 51
```

Pourquoi est-ce mieux?

- Au lieu de faire le travail avec Python, nous transmettons la charge à la base de données, qui est adaptée pour effectuer de telles requêtes.
- Réduit efficacement le nombre de requêtes de base de données nécessaires pour obtenir le résultat souhaité.

Exécuter des opérations arithmétiques entre les champs

`F()` expressions `F()` peuvent être utilisées pour exécuter des opérations arithmétiques (`+` , `-` , `*` etc.) entre les champs de modèle, afin de définir une recherche / connexion algébrique entre elles.

- Laissez le modèle être:

  ```
  class MyModel(models.Model):
      int_1 = models.IntegerField()
      int_2 = models.IntegerField()
  ```

- Maintenant nous allons supposer que nous voulons récupérer tous les objets de `MyModel` tableau qui a les `int_1` et `int_2` champs satisfont à cette équation: `int_1 + int_2 >= 5` . En utilisant `annotate()` et `filter()` nous obtenons:

  ```
  result = MyModel.objects.annotate(
               diff=F(int_1) + F(int_2)
           ).filter(diff__gte=5)
  ```

  `result` contient désormais tous les objets susmentionnés.

Bien que l'exemple utilise des champs `Integer` , cette méthode fonctionnera sur tous les champs sur lesquels une opération arithmétique peut être appliquée.

Lire Expressions F () en ligne: https://riptutorial.com/fr/django/topic/2765/expressions-f---

Chapitre 22: Extension ou substitution de modèle d'utilisateur

Examples

Modèle utilisateur personnalisé avec courrier électronique en tant que champ de connexion principal.

models.py:

```
from __future__ import unicode_literals
from django.db import models
from django.contrib.auth.models import (
        AbstractBaseUser, BaseUserManager, PermissionsMixin)
from django.utils import timezone
from django.utils.translation import ugettext_lazy as _


class UserManager(BaseUserManager):
    def _create_user(self, email,password, is_staff, is_superuser, **extra_fields):
        now = timezone.now()
        if not email:
            raise ValueError('users must have an email address')
        email = self.normalize_email(email)
        user = self.model(email = email,
                            is_staff = is_staff,
                            is_superuser = is_superuser,
                            last_login = now,
                            date_joined = now,
                            **extra_fields)
        user.set_password(password)
        user.save(using = self._db)
        return user

    def create_user(self, email, password=None, **extra_fields):
        user = self._create_user(email, password, False, False, **extra_fields)
        return user

    def create_superuser(self, email, password, **extra_fields):
        user = self._create_user(email, password, True, True, **extra_fields)
        return user

class User(AbstractBaseUser,PermissionsMixin):
    """My own custom user class"""

    email = models.EmailField(max_length=255, unique=True, db_index=True,
verbose_name=_('email address'))
    date_joined = models.DateTimeField(auto_now_add=True)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    objects = UserManager()

    USERNAME_FIELD = 'email'
```

```
    REQUIRED_FIELDS = []

    class Meta:
        verbose_name = _('user')
        verbose_name_plural = _('users')

    def get_full_name(self):
    """Return the email."""
        return self.email

    def get_short_name(self):
    """Return the email."""
        return self.email
```

forms.py:

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from .models import User


class RegistrationForm(UserCreationForm):
    email = forms.EmailField(widget=forms.TextInput(
        attrs={'class': 'form-control','type':'text','name': 'email'}),
        label="Email")
    password1 = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name':'password1'}),
        label="Password")
    password2 = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name': 'password2'}),
        label="Password (again)")

    '''added attributes so as to customise for styling, like bootstrap'''
    class Meta:
        model = User
        fields = ['email','password1','password2']
        field_order = ['email','password1','password2']

    def clean(self):
    """
    Verifies that the values entered into the password fields match
    NOTE : errors here will appear in 'non_field_errors()'
    """
        cleaned_data = super(RegistrationForm, self).clean()
        if 'password1' in self.cleaned_data and 'password2' in self.cleaned_data:
            if self.cleaned_data['password1'] != self.cleaned_data['password2']:
                raise forms.ValidationError("Passwords don't match. Please try again!")
        return self.cleaned_data

    def save(self, commit=True):
        user = super(RegistrationForm,self).save(commit=False)
        user.set_password(self.cleaned_data['password1'])
        if commit:
            user.save()
        return user

#The save(commit=False) tells Django to save the new record, but dont commit it to the
database yet

class AuthenticationForm(forms.Form): # Note: forms.Form NOT forms.ModelForm
```

```
    email = forms.EmailField(widget=forms.TextInput(
        attrs={'class': 'form-control','type':'text','name': 'email','placeholder':'Email'}),
        label='Email')
    password = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name':
'password','placeholder':'Password'}),
        label='Password')

    class Meta:
        fields = ['email', 'password']
```

views.py:

```
from django.shortcuts import redirect, render, HttpResponse
from django.contrib.auth import login as django_login, logout as django_logout, authenticate
as django_authenticate
#importing as such so that it doesn't create a confusion with our methods and django's default
methods

from django.contrib.auth.decorators import login_required
from .forms import AuthenticationForm, RegistrationForm


def login(request):
    if request.method == 'POST':
        form = AuthenticationForm(data = request.POST)
        if form.is_valid():
            email = request.POST['email']
            password = request.POST['password']
            user = django_authenticate(email=email, password=password)
            if user is not None:
                if user.is_active:
                    django_login(request,user)
                    return redirect('/dashboard') #user is redirected to dashboard
    else:
        form = AuthenticationForm()

    return render(request,'login.html',{'form':form,})

def register(request):
    if request.method == 'POST':
        form = RegistrationForm(data = request.POST)
        if form.is_valid():
            user = form.save()
            u = django_authenticate(user.email = user, user.password = password)
            django_login(request,u)
            return redirect('/dashboard')
    else:
        form = RegistrationForm()

    return render(request,'register.html',{'form':form,})

def logout(request):
    django_logout(request)
    return redirect('/')

@login_required(login_url ="/")
def dashboard(request):
    return render(request, 'dashboard.html',{})
```

settings.py:

```
AUTH_USER_MODEL = 'myapp.User'
```

admin.py

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.models import Group
from .models import User


class UserAdmin(BaseUserAdmin):
    list_display = ('email','is_staff')
    list_filter = ('is_staff',)
    fieldsets = ((None,
                  {'fields':('email','password')}), ('Permissions',{'fields':('is_staff',)}),)
    add_fieldsets = ((None, {'classes': ('wide',), 'fields': ('email', 'password1',
'password2')}),)
    search_fields =('email',)
    ordering = ('email',)
    filter_horizontal = ()

admin.site.register(User, UserAdmin)
admin.site.unregister(Group)
```

Utilisez le `email` comme nom d'utilisateur et débarrassez-vous du champ` username`

Si vous voulez vous débarrasser du champ `username` et utiliser le `email` comme identifiant d'utilisateur unique, vous devrez créer un modèle d' `User` personnalisé étendant `AbstractBaseUser` au lieu de `AbstractUser` . En effet, le `username` et le `email` sont définis dans `AbstractUser` et vous ne pouvez pas les remplacer. Cela signifie que vous devrez également redéfinir tous les champs que vous souhaitez définir dans `AbstractUser` .

```
from django.contrib.auth.models import (
    AbstractBaseUser, PermissionsMixin, BaseUserManager,
)
from django.db import models
from django.utils import timezone
from django.utils.translation import ugettext_lazy as _

class UserManager(BaseUserManager):

    use_in_migrations = True

    def _create_user(self, email, password, **extra_fields):
        if not email:
            raise ValueError('The given email must be set')
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user
```

```
    def create_user(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', False)
        extra_fields.setdefault('is_superuser', False)
        return self._create_user(email, password, **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)

        if extra_fields.get('is_staff') is not True:
            raise ValueError('Superuser must have is_staff=True.')
        if extra_fields.get('is_superuser') is not True:
            raise ValueError('Superuser must have is_superuser=True.')

    return self._create_user(email, password, **extra_fields)


class User(AbstractBaseUser, PermissionsMixin):
    """PermissionsMixin contains the following fields:
        - `is_superuser`
        - `groups`
        - `user_permissions`
     You can omit this mix-in if you don't want to use permissions or
     if you want to implement your own permissions logic.
     """

    class Meta:
        verbose_name = _("user")
        verbose_name_plural = _("users")
        db_table = 'auth_user'
        # `db_table` is only needed if you move from the existing default
        # User model to a custom one. This enables to keep the existing data.

    USERNAME_FIELD = 'email'
    """Use the email as unique username."""

    REQUIRED_FIELDS = ['first_name', 'last_name']

    GENDER_MALE = 'M'
    GENDER_FEMALE = 'F'
    GENDER_CHOICES = [
        (GENDER_MALE, _("Male")),
        (GENDER_FEMALE, _("Female")),
    ]

    email = models.EmailField(
        verbose_name=_("email address"), unique=True,
        error_messages={
            'unique': _(
                "A user is already registered with this email address"),
        },
    )
    gender = models.CharField(
        max_length=1, blank=True, choices=GENDER_CHOICES,
        verbose_name=_("gender"),
    )
    first_name = models.CharField(
        max_length=30, verbose_name=_("first name"),
    )
    last_name = models.CharField(
        max_length=30, verbose_name=_("last name"),
```

```
    )
    is_staff = models.BooleanField(
        verbose_name=_("staff status"),
        default=False,
        help_text=_(
            "Designates whether the user can log into this admin site."
        ),
    )
    is_active = models.BooleanField(
        verbose_name=_("active"),
        default=True,
        help_text=_(
            "Designates whether this user should be treated as active. "
            "Unselect this instead of deleting accounts."
        ),
    )
    date_joined = models.DateTimeField(
        verbose_name=_("date joined"), default=timezone.now,
    )

    objects = UserManager()
```

Étendre facilement le modèle d'utilisateur de Django

Notre classe `UserProfile`

Créez une classe de modèle `UserProfile` avec la relation de `OneToOne` avec le modèle d' `User` par défaut:

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save

class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    photo = FileField(verbose_name=_("Profile Picture"),
                      upload_to=upload_to("main.UserProfile.photo", "profiles"),
                      format="Image", max_length=255, null=True, blank=True)
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)
    phone = models.CharField(max_length=20, blank=True, default='')
    city = models.CharField(max_length=100, default='', blank=True)
    country = models.CharField(max_length=100, default='', blank=True)
    organization = models.CharField(max_length=100, default='', blank=True)
```

Django Signals at work

En utilisant les signaux Django, créez un nouveau `UserProfile` `User` immédiatement après la `UserProfile` un objet `User` . Cette fonction peut être `UserProfile` sous la classe de modèle `UserProfile` dans le même fichier ou la placer où vous le souhaitez. Je m'en fous, autant que vous le référencez correctement.

```
def create_profile(sender, **kwargs):
    user = kwargs["instance"]
    if kwargs["created"]:
        user_profile = UserProfile(user=user)
```

```
        user_profile.save()
post_save.connect(create_profile, sender=User)
```

`inlineformset_factory` à la rescousse

Maintenant, pour vos `views.py` , vous pourriez avoir quelque chose comme ceci:

```
from django.shortcuts import render, HttpResponseRedirect
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from .models import UserProfile
from .forms import UserForm
from django.forms.models import inlineformset_factory
from django.core.exceptions import PermissionDenied
@login_required() # only logged in users should access this
def edit_user(request, pk):
    # querying the User object with pk from url
    user = User.objects.get(pk=pk)

    # prepopulate UserProfileForm with retrieved user values from above.
    user_form = UserForm(instance=user)

    # The sorcery begins from here, see explanation https://blog.khophi.co/extending-django-
user-model-userprofile-like-a-pro/
    ProfileInlineFormset = inlineformset_factory(User, UserProfile, fields=('website', 'bio',
'phone', 'city', 'country', 'organization'))
    formset = ProfileInlineFormset(instance=user)

    if request.user.is_authenticated() and request.user.id == user.id:
        if request.method == "POST":
            user_form = UserForm(request.POST, request.FILES, instance=user)
            formset = ProfileInlineFormset(request.POST, request.FILES, instance=user)

            if user_form.is_valid():
                created_user = user_form.save(commit=False)
                formset = ProfileInlineFormset(request.POST, request.FILES,
instance=created_user)

                if formset.is_valid():
                    created_user.save()
                    formset.save()
                    return HttpResponseRedirect('/accounts/profile/')

        return render(request, "account/account_update.html", {
            "noodle": pk,
            "noodle_form": user_form,
            "formset": formset,
        })
    else:
        raise PermissionDenied
```

Notre modèle

Ensuite, `account_update.html` tout dans votre template `account_update.html` comme `account_update.html` :

```
{% load material_form %}
<!-- Material form is just a materialize thing for django forms -->
```

```
<div class="col s12 m8 offset-m2">
      <div class="card">
        <div class="card-content">
        <h2 class="flow-text">Update your information</h2>
          <form action="." method="POST" class="padding">
            {% csrf_token %} {{ noodle_form.as_p }}
            <div class="divider"></div>
            {{ formset.management_form }}
                {{ formset.as_p }}
            <button type="submit" class="btn-floating btn-large waves-light waves-effect"><i
class="large material-icons">done</i></button>
            <a href="#" onclick="window.history.back(); return false;" title="Cancel"
class="btn-floating waves-effect waves-light red"><i class="material-icons">history</i></a>

        </form>
        </div>
    </div>
</div>
```

Extrait ci- dessus tiré de l' extension du profil utilisateur Django comme un pro

Spécification d'un modèle d'utilisateur personnalisé

Le modèle d' `User` intégré de Django n'est pas toujours adapté à certains types de projets. Sur certains sites, il peut être plus judicieux d'utiliser une adresse e-mail plutôt qu'un nom d'utilisateur, par exemple.

Vous pouvez remplacer le modèle d' `User` par défaut en ajoutant votre modèle d' `User` personnalisé au paramètre `AUTH_USER_MODEL` , dans le fichier de paramètres de vos projets:

```
AUTH_USER_MODEL = 'myapp.MyUser'
```

Notez qu'il est fortement recommandé de créer `AUTH_USER_MODEL` avant de créer des migrations ou d'exécuter `manage.py migrate` pour la première fois. En raison des limitations de la fonction de dépendance synamique de Django.

Par exemple, sur votre blog, vous voudrez peut-être que d'autres auteurs puissent se connecter avec une adresse e-mail au lieu du nom d'utilisateur habituel. Nous créons donc un modèle d' `User` personnalisé avec une adresse e-mail comme `USERNAME_FIELD` :

```
from django.contrib.auth.models import AbstractBaseUser

class CustomUser(AbstractBaseUser):
     email = models.EmailField(unique=True)

     USERNAME_FIELD = 'email'
```

En héritant de `AbstractBaseUser` nous pouvons construire un modèle d' `User` conforme. `AbstractBaseUser` fournit l'implémentation principale d'un modèle d' `User` .

Pour que la commande Django `manage.py createsuperuser` sache quels autres champs sont nécessaires, nous pouvons spécifier un `REQUIRED_FIELDS` . Cette valeur n'a aucun effet dans

d'autres parties de Django!

```
class CustomUser(AbstractBaseUser):
    ...
    first_name = models.CharField(max_length=254)
    last_name = models.CharField(max_length=254)
    ...
    REQUIRED_FIELDS = ['first_name', 'last_name']
```

Pour être compatible avec une autre partie de Django, nous devons toujours spécifier la valeur `is_active` , les fonctions `get_full_name()` et `get_short_name()` :

```
class CustomUser(AbstractBaseUser):
    ...
    is_active = models.BooleanField(default=False)
    ...
    def get_full_name(self):
        full_name = "{0} {1}".format(self.first_name, self.last_name)
        return full_name.strip()

    def get_short_name(self):
        return self.first_name
```

Vous devez également créer un `UserManager` personnalisé pour votre modèle `User` , ce qui permet à Django d'utiliser les fonctions `create_user()` et `create_superuser()` :

```
from django.contrib.auth.models import BaseUserManager

class CustomUserManager(BaseUserManager):
    def create_user(self, email, first_name, last_name, password=None):
        if not email:
            raise ValueError('Users must have an email address')

        user = self.model(
            email=self.normalize_email(email),
        )

        user.set_password(password)
        user.first_name = first_name
        user.last_name = last_name
        user.save(using=self._db)
        return user

    def create_superuser(self, email, first_name, last_name, password):
        user = self.create_user(
            email=email,
            first_name=first_name,
            last_name=last_name,
            password=password,
        )

        user.is_admin = True
        user.is_active = True
        user.save(using=self.db)
        return user
```

Référencement du modèle d'utilisateur

Votre code ne fonctionnera pas dans les projets où vous référencez le modèle `User` (*et où le paramètre `AUTH_USER_MODEL` a été modifié*) directement.

Par exemple: si vous souhaitez créer `Post` modèle `Post` pour un blog avec un modèle d' `User` personnalisé, vous devez spécifier le modèle d' `User` personnalisé comme suit:

```
from django.conf import settings
from django.db import models

class Post(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
```

Lire Extension ou substitution de modèle d'utilisateur en ligne:
https://riptutorial.com/fr/django/topic/1209/extension-ou-substitution-de-modele-d-utilisateur

Chapitre 23: Formes

Examples

Exemple ModelForm

Créez un ModelForm à partir d'une classe Model existante, en sous- `ModelForm` :

```
from django import forms

class OrderForm(forms.ModelForm):
    class Meta:
        model = Order
        fields = ['item', 'order_date', 'customer', 'status']
```

Définir un formulaire Django à partir de zéro (avec des widgets)

Les formulaires peuvent être définis, de la même manière que les modèles, en sous- `django.forms.Form` .
Différentes options de saisie sur le terrain sont disponibles, telles que `CharField` , `URLField` , `IntegerField` , etc.

Vous trouverez ci-dessous la définition d'un formulaire de contact simple:

```
from django import forms

class ContactForm(forms.Form):
    contact_name = forms.CharField(
        label="Your name", required=True,
        widget=forms.TextInput(attrs={'class': 'form-control'}))
    contact_email = forms.EmailField(
        label="Your Email Address", required=True,
        widget=forms.TextInput(attrs={'class': 'form-control'}))
    content = forms.CharField(
        label="Your Message", required=True,
        widget=forms.Textarea(attrs={'class': 'form-control'}))
```

Widget est la représentation par Django des balises HTML saisies par l'utilisateur et peut être utilisée pour rendre le HTML personnalisé pour les champs de formulaire (par exemple: une zone de texte est rendue pour le contenu saisi ici)

`attrs` attributs sont des attributs qui seront copiés tels `attrs` HTML rendu pour le formulaire.

Par exemple: `content.render("name", "Your Name")` donne

```
<input title="Your name" type="text" name="name" value="Your Name" class="form-control" />
```

Suppression du champ modelForm en fonction de la condition de views.py

Si nous avons un modèle comme suit,

```
from django.db import models
from django.contrib.auth.models import User

class UserModuleProfile(models.Model):
    user = models.OneToOneField(User)
    expired = models.DateTimeField()
    admin = models.BooleanField(default=False)
    employee_id = models.CharField(max_length=50)
    organisation_name = models.ForeignKey('Organizations', on_delete=models.PROTECT)
    country = models.CharField(max_length=100)
    position = models.CharField(max_length=100)

    def __str__(self):
        return self.user
```

Et une forme de modèle qui utilise ce modèle comme suit,

```
from .models import UserModuleProfile, from django.contrib.auth.models import User
from django import forms

class UserProfileForm(forms.ModelForm):
    admin = forms.BooleanField(label="Make this User
Admin",widget=forms.CheckboxInput(),required=False)
    employee_id = forms.CharField(label="Employee Id ")
    organisation_name = forms.ModelChoiceField(label='Organisation
Name',required=True,queryset=Organizations.objects.all(),empty_label="Select an Organization")
    country = forms.CharField(label="Country")
    position = forms.CharField(label="Position")

    class Meta:
        model = UserModuleProfile
        fields = ('admin','employee_id','organisation_name','country','position',)

    def __init__(self, *args, **kwargs):
        admin_check = kwargs.pop('admin_check', False)
        super(UserProfileForm, self).__init__(*args, **kwargs)
        if not admin_check:
            del self.fields['admin']
```

Notez que sous la classe Meta dans le formulaire, j'ai ajouté une fonction d' **initialisation** que nous pouvons utiliser lors de l'initialisation du formulaire depuis views.py pour supprimer un champ de formulaire (ou d'autres actions). Je vais expliquer cela plus tard.

Donc, ce formulaire peut être utilisé à des fins d'inscription des utilisateurs et nous voulons tous les champs définis dans la classe Meta du formulaire. Mais que se passe-t-il si nous voulons utiliser le même formulaire lorsque nous modifions l'utilisateur, mais lorsque nous le faisons, nous ne voulons pas afficher le champ admin du formulaire?

Nous pouvons simplement envoyer un argument supplémentaire lorsque nous initialisons le formulaire en fonction d’une logique et supprimons le champ admin du backend.

```
def edit_profile(request,user_id):
    context = RequestContext(request)
```

```
    user = get_object_or_404(User, id=user_id)
    profile = get_object_or_404(UserModuleProfile, user_id=user_id)
    admin_check = False
    if request.user.is_superuser:
        admin_check = True
    # If it's a HTTP POST, we're interested in processing form data.
    if request.method == 'POST':
        # Attempt to grab information from the raw form information.
        profile_form =
UserProfileForm(data=request.POST,instance=profile,admin_check=admin_check)
        # If the form is valid...
        if profile_form.is_valid():
            form_bool = request.POST.get("admin", "xxx")
            if form_bool == "xxx":
                form_bool_value = False
            else:
                form_bool_value = True
            profile = profile_form.save(commit=False)
            profile.user = user
            profile.admin = form_bool_value
            profile.save()
            edited = True
        else:
            print profile_form.errors

    # Not a HTTP POST, so we render our form using ModelForm instance.
    # These forms will be blank, ready for user input.
    else:
        profile_form = UserProfileForm(instance = profile,admin_check=admin_check)

    return render_to_response(
            'usermodule/edit_user.html',
            {'id':user_id, 'profile_form': profile_form, 'edited': edited, 'user':user},
            context)
```

Comme vous pouvez le voir, j'ai montré ici un exemple d'édition simple utilisant le formulaire que nous avons créé précédemment. Notez que lorsque j'ai initialisé le formulaire, j'ai transmis une variable `admin_check` supplémentaire contenant `True` ou `False` .

```
profile_form = UserProfileForm(instance = profile,admin_check=admin_check)
```

Maintenant, si vous remarquez le formulaire que nous avons écrit plus tôt, vous pouvez voir que dans l' **init,** nous essayons d'attraper le `admin_check` que nous passons ici. Si la valeur est False, il suffit de supprimer le champ `admin` du formulaire et de l'utiliser. Et comme il s'agit d'un modèle, le champ admin ne peut pas être nul dans le modèle. Nous vérifions simplement si le post d'administration contenait un champ admin dans le post du formulaire, sinon nous avons défini `False` dans le code de la vue.

```
form_bool = request.POST.get("admin", "xxx")
if form_bool == "xxx":
    form_bool_value = False
else:
    form_bool_value = True
```

Téléchargement de fichiers avec les formulaires Django

Tout d'abord, nous devons ajouter `MEDIA_ROOT` et `MEDIA_URL` à notre fichier `settings.py`

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

Ici aussi, vous travaillerez avec `ImageField`, donc souvenez-vous, dans ce cas, installez la bibliothèque Pillow (`pip install pillow`). Sinon, vous aurez une telle erreur:

```
ImportError: No module named PIL
```

Pillow est un fork de PIL, la bibliothèque de création d'images Python, qui n'est plus maintenue. L'oreiller est rétrocompatible avec PIL.

Django est livré avec deux champs de formulaire pour télécharger des fichiers sur le serveur, `FileField` et `ImageField`. Voici un exemple d'utilisation de ces deux champs dans notre formulaire.

forms.py:

```
from django import forms


class UploadDocumentForm(forms.Form):
    file = forms.FileField()
    image = forms.ImageField()
```

views.py:

```
from django.shortcuts import render
from .forms import UploadDocumentForm


def upload_doc(request):
    form = UploadDocumentForm()
    if request.method == 'POST':
        form = UploadDocumentForm(request.POST, request.FILES)  # Do not forget to add: 
request.FILES
        if form.is_valid():
            # Do something with our files or simply save them
            # if saved, our files would be located in media/ folder under the project's base 
folder
            form.save()
    return render(request, 'upload_doc.html', locals())
```

upload_doc.html:

```
<html>
    <head>File Uploads</head>
    <body>
        <form enctype="multipart/form-data" action="" method="post"> <!-- Do not forget to 
add: enctype="multipart/form-data" -->
            {% csrf_token %}
            {{ form }}
            <input type="submit" value="Save">
        </form>
```

```
    </body>
</html>
```

Validation des champs et Validation du modèle (Modification de l'e-mail de l'utilisateur)

Il existe déjà des formulaires implémentés dans Django pour modifier le mot de passe de l'utilisateur, par exemple SetPasswordForm .

Il n'y a cependant pas de formulaire pour modifier le courrier électronique de l'utilisateur et je pense que l'exemple suivant est important pour comprendre comment utiliser un formulaire correctement.

L'exemple suivant effectue les vérifications suivantes:

- Les e-mails ont en effet changé - très utile si vous devez valider l'e-mail ou mettre à jour le courrier chimp;
- Les e-mails et les e-mails de confirmation sont les mêmes: le formulaire contient deux champs pour le courrier électronique. La mise à jour est donc moins sujette aux erreurs.

Et à la fin, il enregistre le nouvel e-mail dans l'objet utilisateur (met à jour l'e-mail de l'utilisateur). Notez que la `__init__()` nécessite un objet utilisateur.

```
class EmailChangeForm(forms.Form):
    """
    A form that lets a user change set their email while checking for a change in the
    e-mail.
    """
    error_messages = {
        'email_mismatch': _("The two email addresses fields didn't match."),
        'not_changed': _("The email address is the same as the one already defined."),
    }

    new_email1 = forms.EmailField(
        label=_("New email address"),
        widget=forms.EmailInput,
    )

    new_email2 = forms.EmailField(
        label=_("New email address confirmation"),
        widget=forms.EmailInput,
    )

    def __init__(self, user, *args, **kwargs):
        self.user = user
        super(EmailChangeForm, self).__init__(*args, **kwargs)

    def clean_new_email1(self):
        old_email = self.user.email
        new_email1 = self.cleaned_data.get('new_email1')
        if new_email1 and old_email:
            if new_email1 == old_email:
                raise forms.ValidationError(
                    self.error_messages['not_changed'],
                    code='not_changed',
```

```
                )
        return new_email1

    def clean_new_email2(self):
        new_email1 = self.cleaned_data.get('new_email1')
        new_email2 = self.cleaned_data.get('new_email2')
        if new_email1 and new_email2:
            if new_email1 != new_email2:
                raise forms.ValidationError(
                    self.error_messages['email_mismatch'],
                    code='email_mismatch',
                )
        return new_email2

    def save(self, commit=True):
        email = self.cleaned_data["new_email1"]
        self.user.email = email
        if commit:
            self.user.save()
        return self.user



def email_change(request):
    form = EmailChangeForm()
    if request.method=='POST':
        form = Email_Change_Form(user,request.POST)
        if form.is_valid():
            if request.user.is_authenticated:
                if form.cleaned_data['email1']  == form.cleaned_data['email2']:
                    user = request.user
                    u = User.objects.get(username=user)
                    # get the proper user
                    u.email = form.cleaned_data['email1']
                    u.save()
                    return HttpResponseRedirect("/accounts/profile/")
    else:
        return render_to_response("email_change.html", {'form':form},
                                   context_instance=RequestContext(request))
```

Lire Formes en ligne: https://riptutorial.com/fr/django/topic/1217/formes

Chapitre 24: Formsets

Syntaxe

- NewFormSet = formset_factory (SomeForm, extra = 2)
- formset = NewFormSet (initial = [{'some_field': 'Valeur du champ', 'other_field': 'Autre valeur du champ',}])

Examples

Formsets avec des données initialisées et unifiées

`Formset` est un moyen de rendre plusieurs formulaires dans une page, comme une grille de données. Ex: Ce `ChoiceForm` peut être associé à une question de tri. comme, les enfants sont les plus intelligents entre quel âge?

`appname/forms.py`

```
from django import forms
class ChoiceForm(forms.Form):
    choice = forms.CharField()
    pub_date = forms.DateField()
```

Dans vos vues, vous pouvez utiliser le constructeur `formset_factory` qui prend prendre `Form` comme paramètre, son `ChoiceForm` dans ce cas et `extra` qui décrit combien de formulaires supplémentaires autres que la forme / les formulaires initialisés doivent être rendus et vous pouvez `formset` objet `formset` comme n'importe quel autre itérable.

Si le formset n'est pas initialisé avec des données, il imprime le nombre de formulaires égal à `extra + 1` et si le formset est initialisé, il imprime `initialized + extra` lorsque `extra` nombre de formulaires vides autres que ceux initialisés est supérieur.

`appname/views.py`

```
import datetime
from django.forms import formset_factory
from appname.forms import ChoiceForm
    ChoiceFormSet = formset_factory(ChoiceForm, extra=2)
    formset = ChoiceFormSet(initial=[
      {'choice': 'Between 5-15 ?',
        'pub_date': datetime.date.today(),}
      ])
```

si vous `formset object` sur un `formset object` comme celui-ci pour formulaire dans formset: print (form.as_table ())

`Output in rendered template`

```
<tr>
<th><label for="id_form-0-choice">Choice:</label></th>
<td><input type="text" name="form-0-choice" value="Between 5-15 ?" id="id_form-0-choice"
/></td>
</tr>
<tr>
<th><label for="id_form-0-pub_date">Pub date:</label></th>
<td><input type="text" name="form-0-pub_date" value="2008-05-12" id="id_form-0-pub_date"
/></td>
</tr>
<tr>
<th><label for="id_form-1-choice">Choice:</label></th>
<td><input type="text" name="form-1-choice" id="id_form-1-choice" /></td>
</tr>
<tr>
<th><label for="id_form-1-pub_date">Pub date:</label></th>
<td><input type="text" name="form-1-pub_date" id="id_form-1-pub_date" /></td
</tr>
<tr>
<th><label for="id_form-2-choice">Choice:</label></th>
<td><input type="text" name="form-2-choice" id="id_form-2-choice" /></td>
</tr>
<tr>
<th><label for="id_form-2-pub_date">Pub date:</label></th>
<td><input type="text" name="form-2-pub_date" id="id_form-2-pub_date" /></td>
</tr>
```

Lire Formsets en ligne: https://riptutorial.com/fr/django/topic/6082/formsets

Chapitre 25: Fuseaux horaires

Introduction

Les fuseaux horaires sont souvent un problème pour les développeurs. Django met à votre disposition de nombreux utilitaires pour faciliter la collaboration avec les fuseaux horaires.

Même si votre projet fonctionne dans un seul fuseau horaire, il est toujours recommandé de stocker les données sous forme de temps UTC dans votre base de données pour gérer les cas d'heure avancée. Si vous travaillez sur plusieurs fuseaux horaires, le stockage des données horaires sous forme de temps UTC est indispensable.

Examples

Activer le support de fuseau horaire

Tout d'abord, assurez-vous que `USE_TZ = True` dans votre fichier `settings.py` . `TIME_ZONE` également une valeur de fuseau horaire par défaut sur `TIME_ZONE` par exemple `TIME_ZONE='UTC'` . Afficher une liste complète des fuseaux horaires ici .

Si `USE_TZ` False, `TIME_ZONE` sera le fuseau horaire que Django utilisera pour stocker toutes les datetimes. Lorsque `USE_TZ` est activé, `TIME_ZONE` est le fuseau horaire par défaut que Django utilisera pour afficher les datetimes dans les modèles et pour interpréter les dathale saisies dans les formulaires.

Avec la prise en charge du fuseau horaire activée, django stockera les données `datetime` dans la base de données en tant que `UTC` horaire `UTC`

Définition des fuseaux horaires de session

Les objets `datetime.datetime` de Python ont un attribut `tzinfo` utilisé pour stocker les informations de fuseau horaire. Lorsque l'attribut est défini, l'objet est considéré comme Aware, lorsque l'attribut n'est pas défini, il est considéré comme une Naive.

Pour vous assurer qu'un fuseau horaire est naïf ou conscient, vous pouvez utiliser `.is_naive()` et `.is_aware()`

Si `USE_TZ` activé dans votre fichier `settings.py` , une `datetime` heure sera associée à des informations de fuseau horaire tant que votre `TIME_ZONE` par défaut est défini dans `settings.py`

Bien que ce fuseau horaire par défaut puisse être bon dans certains cas, il est probable que cela ne soit pas suffisant, surtout si vous manipulez des utilisateurs dans plusieurs fuseaux horaires. Pour ce faire, le middleware doit être utilisé.

```
import pytz
```

```
from django.utils import timezone

# make sure you add `TimezoneMiddleware` appropriately in settings.py
class TimezoneMiddleware(object):
    """
    Middleware to properly handle the users timezone
    """

    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # make sure they are authenticated so we know we have their tz info.
        if request.user.is_authenticated():
            # we are getting the users timezone that in this case is stored in
            # a user's profile
            tz_str = request.user.profile.timezone
            timezone.activate(pytz.timezone(tz_str))
        # otherwise deactivate and the default time zone will be used anyway
        else:
            timezone.deactivate()

        response = self.get_response(request)
        return response
```

Il y a quelques nouvelles choses qui se passent. Pour en savoir plus sur le middleware et sur son contenu, consultez cette partie de la documentation . Dans `__call__` nous `__call__` le paramétrage des données de fuseau horaire. Au début, nous nous assurons que l'utilisateur est authentifié, pour s'assurer que nous avons des données de fuseau horaire pour cet utilisateur. Une fois que nous savons que nous le faisons, nous activons le fuseau horaire pour la session des utilisateurs en utilisant `timezone.activate()` . Pour convertir la chaîne de fuseau horaire en quelque chose utilisable par datetime, nous utilisons `pytz.timezone(str)` .

Désormais, lorsque les objets datetime sont accessibles dans les modèles, ils seront automatiquement convertis du format 'UTC' de la base de données vers le fuseau horaire de l'utilisateur. Accédez simplement à l'objet datetime et son fuseau horaire correctement.

```
{{ my_datetime_value }}
```

Si vous souhaitez un contrôle précis de l'utilisation du fuseau horaire de l'utilisateur, consultez les éléments suivants:

```
{% load tz %}
{% localtime on %}
    {# this time will be respect the users time zone #}
    {{ your_date_time }}
{% endlocaltime %}

{% localtime off %}
    {# this will not respect the users time zone #}
    {{ your_date_time }}
{% endlocaltime %}
```

Notez que cette méthode décrite ne fonctionne que dans Django 1.10 et suivants. Pour supporter

django avant 1.10 regardez dans MiddlewareMixin

Lire Fuseaux horaires en ligne: https://riptutorial.com/fr/django/topic/10566/fuseaux-horaires

Chapitre 26: Gestionnaires et ensembles de requêtes personnalisés

Examples

Définition d'un gestionnaire de base à l'aide de la méthode Querysets et de la méthode `as_manager`

Django manger est une interface par laquelle le modèle django interroge la base de données. Le champ d' `objects` utilisé dans la plupart des requêtes django est en fait le gestionnaire par défaut créé par django (il est créé uniquement si nous ne définissons pas de gestionnaires personnalisés).

Pourquoi définirions-nous un gestionnaire / groupe de requête personnalisé?

Pour éviter d'écrire des requêtes communes sur notre base de code et de les renvoyer à l'aide d'une abstraction plus facile à retenir. Exemple: Décidez pour vous-même quelle version est la plus lisible:

- `User.objects.filter(is_active=True)` que tous les utilisateurs actifs: `User.objects.filter(is_active=True)` VS `User.manager.active()`
- Obtenez tous les dermatologues actifs sur notre plateforme: `User.objects.filter(is_active=True).filter(is_doctor=True).filter(specialization='Dermatology')` VS `User.manager.doctors.with_specialization('Dermatology')`

Un autre avantage est que si demain nous décidons que tous les `psychologists` sont aussi des `dermatologists` , nous pouvons facilement modifier la requête dans notre gestionnaire et en finir avec elle.

Vous trouverez ci-dessous un exemple de création d'un `Manager` personnalisé défini en créant un `QuerySet` et en utilisant la méthode `as_manager` .

```
from django.db.models.query import QuerySet

class ProfileQuerySet(QuerySet):
    def doctors(self):
        return self.filter(user_type="Doctor", user__is_active=True)

    def with_specializations(self, specialization):
        return self.filter(specializations=specialization)

    def users(self):
        return self.filter(user_type="Customer", user__is_active=True)

ProfileManager = ProfileQuerySet.as_manager
```

Nous allons l'ajouter à notre modèle comme ci-dessous:

```
class Profile(models.Model):
    ...
    manager = ProfileManager()
```

REMARQUE : Une fois que nous avons défini un `manager` sur notre modèle, les `objects` ne seront plus définis pour le modèle.

select_related pour toutes les requêtes

Modèle avec ForeignKey

Nous allons travailler avec ces modèles:

```
from django.db import models

class Book(models.Model):
 name= models.CharField(max_length=50)
 author = models.ForeignKey(Author)

class Author(models.Model):
 name = models.CharField(max_length=50)
```

Supposons que nous `book.author.name` souvent (toujours) à `book.author.name`

En vue

Nous pourrions utiliser ce qui suit à chaque fois

```
books = Book.objects.select_related('author').all()
```

Mais ce n'est pas sec.

Gestionnaire personnalisé

```
class BookManager(models.Manager):

    def get_queryset(self):
        qs = super().get_queryset()
        return qs.select_related('author')

class Book(models.Model):
    ...
    objects = BookManager()
```

Note : l'appel à `super` doit être changé pour python 2.x

Maintenant, tout ce que nous devons utiliser dans les vues est

```
books = Book.objects.all()
```

et aucune requête supplémentaire ne sera faite dans un modèle / vue.

Définir des gestionnaires personnalisés

Très souvent, il s'agit de modèles qui ont quelque chose comme un champ `published` . Ces types de champs sont presque toujours utilisés lors de la récupération d'objets, de sorte que vous vous retrouvez à écrire quelque chose comme:

```
my_news = News.objects.filter(published=True)
```

trop de fois. Vous pouvez utiliser des gestionnaires personnalisés pour gérer ces situations, de sorte que vous puissiez ensuite écrire quelque chose comme:

```
my_news = News.objects.published()
```

ce qui est plus agréable et plus facile à lire par les autres développeurs.

Créez un fichier `managers.py` dans votre répertoire d'application et définissez une nouvelle classe `models.Manager` :

```
from django.db import models


class NewsManager(models.Manager):

    def published(self, **kwargs):
        # the method accepts **kwargs, so that it is possible to filter
        # published news
        # i.e: News.objects.published(insertion_date__gte=datetime.now)
        return self.filter(published=True, **kwargs)
```

utilisez cette classe en redéfinissant la propriété `objects` dans la classe de modèle:

```
from django.db import models

# import the created manager
from .managers import NewsManager

class News(models.Model):
    """ News model
    """
    insertion_date = models.DateTimeField('insertion date', auto_now_add=True)
    title = models.CharField('title', max_length=255)
    # some other fields here
    published = models.BooleanField('published')

    # assign the manager class to the objects property
    objects = NewsManager()
```

Maintenant, vous pouvez obtenir vos nouvelles publiées simplement de cette manière:

```
my_news = News.objects.published()
```

et vous pouvez également effectuer plus de filtrage:

```
my_news = News.objects.published(title__icontains='meow')
```

Lire Gestionnaires et ensembles de requêtes personnalisés en ligne: https://riptutorial.com/fr/django/topic/1400/gestionnaires-et-ensembles-de-requetes-personnalises

Chapitre 27: Intégration continue avec Jenkins

Examples

Script de pipeline Jenkins 2.0+

Les versions modernes de Jenkins (version 2.x) sont livrées avec un "plug-in de pipeline" qui peut être utilisé pour orchestrer des tâches complexes sans créer une multitude de tâches interconnectées, et vous permet de contrôler facilement la version de votre configuration.

Vous pouvez l'installer manuellement dans un travail de type "Pipeline" ou, si votre projet est hébergé sur Github, vous pouvez utiliser le "GitHub Organization Folder Plugin" pour configurer automatiquement les tâches pour vous.

Voici une configuration simple pour les sites Django qui nécessitent uniquement l'installation des modules python spécifiés sur le site.

```
#!/usr/bin/groovy

node {
  // If you are having issues with your project not getting updated,
  // try uncommenting the following lines.
  //stage 'Checkout'
  //checkout scm
  //sh 'git submodule update --init --recursive'

  stage 'Update Python Modules'
  // Create a virtualenv in this folder, and install or upgrade packages
  // specified in requirements.txt; https://pip.readthedocs.io/en/1.1/requirements.html
  sh 'virtualenv env && source env/bin/activate && pip install --upgrade -r requirements.txt'

  stage 'Test'
  // Invoke Django's tests
  sh 'source env/bin/activate && python ./manage.py runtests'
}
```

Script de pipeline Jenkins 2.0+, conteneurs Docker

Voici un exemple de script de pipeline qui génère un conteneur Docker, puis exécute les tests à l'intérieur de celui-ci. Le point d'entrée est supposé être `manage.py` ou `invoke` / `fabric` avec une commande `runtests` disponible.

```
#!/usr/bin/groovy

node {
  stage 'Checkout'
  checkout scm
  sh 'git submodule update --init --recursive'
```

```
  imageName = 'mycontainer:build'
  remotes = [
    'dockerhub-account',
  ]

  stage 'Build'
  def djangoImage = docker.build imageName

  stage 'Run Tests'
  djangoImage.run('', 'runtests')

  stage 'Push'
  for (int i = 0; i < remotes.size(); i++) {
      sh "docker tag ${imageName} ${remotes[i]}/${imageName}"
      sh "docker push ${remotes[i]}/${imageName}"
   }
}
```

Lire Intégration continue avec Jenkins en ligne:
https://riptutorial.com/fr/django/topic/5873/integration-continue-avec-jenkins

Chapitre 28: Internationalisation

Syntaxe

- gettext (message)
- ngettext (singulier, pluriel, nombre)
- ugettext (message)
- ungettext (singulier, pluriel, nombre)
- pgettext (contexte, message)
- npgettext (contexte, singulier, pluriel, nombre)
- gettext_lazy (message)
- ngettext_lazy (singulier, pluriel, nombre = aucun)
- ugettext_lazy (message)
- ungettext_lazy (singulier, pluriel, nombre = aucun)
- pgettext_lazy (contexte, message)
- npgettext_lazy (contexte, singulier, pluriel, nombre = aucun)
- gettext_noop (message)
- ugettext_noop (message)

Examples

Introduction à l'internationalisation

Mise en place

settings.py

```
from django.utils.translation import ugettext_lazy as _

USE_I18N = True  # Enable Internationalization
LANGUAGE_CODE = 'en'  # Language in which original texts are written
LANGUAGES = [  # Available languages
    ('en', _("English")),
    ('de', _("German")),
    ('fr', _("French")),
]

# Make sure the LocaleMiddleware is included, AFTER SessionMiddleware
# and BEFORE middlewares using internationalization (such as CommonMiddleware)
MIDDLEWARE_CLASSES = [
   'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.locale.LocaleMiddleware',
   'django.middleware.common.CommonMiddleware',
]
```

Marquage des chaînes comme traduisible

La première étape de la traduction consiste à *marquer les chaînes comme pouvant être traduites* . Cela les fait passer par l'une des fonctions de `gettext` (voir la section Syntaxe). Par exemple, voici un exemple de définition de modèle:

```
from django.utils.translation import ugettext_lazy as _
# It is common to import gettext as the shortcut `_` as it is often used
# several times in the same file.

class Child(models.Model):

    class Meta:
        verbose_name = _("child")
        verbose_name_plural = _("children")

    first_name = models.CharField(max_length=30, verbose_name=_("first name"))
    last_name = models.CharField(max_length=30, verbose_name=_("last name"))
    age = models.PositiveSmallIntegerField(verbose_name=_("age"))
```

Toutes les chaînes encapsulées dans `_()` sont maintenant marquées comme traduisibles. Une fois imprimées, elles seront toujours affichées en tant que chaîne encapsulée, quelle que soit la langue choisie (car aucune traduction n'est encore disponible).

Traduire les chaînes

Cet exemple est suffisant pour commencer avec la traduction. La plupart du temps, vous ne voudrez que marquer des chaînes comme traduisibles pour **anticiper l'internationalisation prospective** de votre projet. Ainsi, cela est couvert dans un autre exemple .

Traduction paresseuse vs non paresseuse

Lorsque vous utilisez une traduction non paresseuse, les chaînes sont traduites immédiatement.

```
>>> from django.utils.translation import activate, ugettext as _
>>> month = _("June")
>>> month
'June'
>>> activate('fr')
>>> _("June")
'juin'
>>> activate('de')
>>> _("June")
'Juni'
>>> month
'June'
```

Lorsque vous utilisez la paresse, la traduction ne se produit que lorsqu'elle est réellement utilisée.

```
>>> from django.utils.translation import activate, ugettext_lazy as _
```

```
>>> month = _("June")
>>> month
<django.utils.functional.lazy.<locals>.__proxy__ object at 0x7f61cb805780>
>>> str(month)
'June'
>>> activate('fr')
>>> month
<django.utils.functional.lazy.<locals>.__proxy__ object at 0x7f61cb805780>
>>> "month: {}".format(month)
'month: juin'
>>> "month: %s" % month
'month: Juni'
```

Vous devez utiliser la traduction paresseuse dans les cas où:

- La traduction ne peut pas être activée (langue non sélectionnée) lorsque `_("some string")` est évalué
- Certaines chaînes peuvent être évaluées uniquement au démarrage (par exemple, attributs de classe tels que les définitions de champs de modèle et de formulaire)

Traduction en gabarits

Pour activer la traduction dans les modèles, vous devez charger la bibliothèque `i18n` .

```
{% load i18n %}
```

La traduction de base est faite avec l'étiquette de modèle `trans` .

```
{% trans "Some translatable text" %}
{# equivalent to python `ugettext("Some translatable text")` #}
```

La balise de modèle `trans` prend en charge le contexte:

```
{% trans "May" context "month" %}
{# equivalent to python `pgettext("May", "month")` #}
```

Pour inclure des espaces réservés dans votre chaîne de traduction, comme dans:

```
_("My name is {first_name} {last_name}").format(first_name="John", last_name="Doe")
```

Vous devrez utiliser la `blocktrans` template `blocktrans` :

```
{% blocktrans with first_name="John" last_name="Doe" %}
  My name is {{ first_name }} {{ last_name }}
{% endblocktrans %}
```

Bien sûr, au lieu de `"John"` et `"Doe"` vous pouvez avoir des variables et des filtres:

```
{% blocktrans with first_name=user.first_name last_name=user.last_name|title %}
  My name is {{ first_name }} {{ last_name }}
{% endblocktrans %}
```

Si `first_name` et `last_name` sont déjà dans votre contexte, vous pouvez même omettez la `with` clause:

```
{% blocktrans %}My name is {{ first_name }} {{ last_name }}{% endblocktrans %}
```

Cependant, seules les variables de contexte de "niveau supérieur" peuvent être utilisées. **Cela ne fonctionnera pas:**

```
{% blocktrans %}
    My name is {{ user.first_name }} {{ user.last_name }}
{% endblocktrans %}
```

C'est principalement parce que le nom de la variable est utilisé comme espace réservé dans les fichiers de traduction.

La `blocktrans` modèle `blocktrans` accepte également la pluralisation.

```
{% blocktrans count nb=users|length }}
    There is {{ nb }} user.
{% plural %}
    There are {{ nb }} users.
{% endblocktrans %}
```

Enfin, quelle que soit la bibliothèque `i18n` , vous pouvez transmettre des chaînes traduisibles aux balises de modèle à l'aide de la syntaxe `_("")` .

```
{{ site_name|default:_("It works!") }}
{% firstof var1 var2 _("translatable fallback") %}
```

Ceci est un système de modèle de django intégré magique pour imiter une syntaxe d'appel de fonction mais ce n'est pas un appel de fonction. `_("It works!")` Est passé à la `default` étiquette de modèle comme une chaîne `'_("It works!")'` Qui est ensuite analysé une chaîne traduisible, tout comme le `name` serait analysé comme une variable et `"name"` serait analysé comme une chaîne.

Traduire les chaînes

Pour traduire des chaînes, vous devrez créer des fichiers de traduction. Pour ce faire, Django est livré avec la commande de gestion `makemessages` .

```
$ django-admin makemessages -l fr
processing locale fr
```

La commande ci-dessus détectera toutes les chaînes marquées comme traduisibles dans vos applications installées et créera un fichier de langue pour chaque application pour la traduction française. Par exemple, si vous n'avez qu'une seule application `myapp` contenant des chaînes traduisibles, cela créera un fichier `myapp/locale/fr/LC_MESSAGES/django.po` . Ce fichier peut ressembler à ceci:

```
# SOME DESCRIPTIVE TITLE
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2016-07-24 14:01+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: myapp/models.py:22
msgid "user"
msgstr ""

#: myapp/models.py:39
msgid "A user already exists with this email address."
msgstr ""

#: myapp/templates/myapp/register.html:155
#, python-format
msgid ""
"By signing up, you accept our <a href=\"%(terms_url)s\" "
"target=_blank>Terms of services</a>."
msgstr ""
```

Vous devrez d'abord remplir les espaces réservés (accentués par des majuscules). Ensuite, traduisez les chaînes. `msgid` est la chaîne marquée comme traduisible dans votre code. `msgstr` est l'endroit où vous devez écrire la traduction de la chaîne ci-dessus.

Lorsqu'une chaîne contient des espaces réservés, vous devrez également les inclure dans votre traduction. Par exemple, vous traduirez le dernier message comme suit:

```
#: myapp/templates/myapp/register.html:155
#, python-format
msgid ""
"By signing up, you accept our <a href=\"%(terms_url)s\" "
"target=_blank>Terms of services</a>."
msgstr ""
"En vous inscrivant, vous acceptez nos <a href=\"%(terms_url)s\" "
"target=_blank>Conditions d'utilisation</a>"
```

Une fois votre fichier de traduction terminé, vous devrez compiler les fichiers `.po` dans des fichiers `.mo` . Cela se fait en appelant la commande `compilemessages` management:

```
$ django-admin compilemessages
```

Ça y est, maintenant les traductions sont disponibles.

Pour mettre à jour vos fichiers de traduction lorsque vous apportez des modifications à votre code, vous pouvez réexécuter `django-admin makemessages -l fr` . Cela mettra à jour `.po` fichiers `.po` , conservera vos traductions existantes et ajoutera les nouvelles. Les chaînes supprimées seront toujours disponibles dans les commentaires. Pour mettre à jour `.po` fichiers `.po` pour toutes les langues, lancez `django-admin makemessages -a` . Une fois vos fichiers `.po` mis à jour, n'oubliez pas de lancer à nouveau `django-admin compilemessages` pour générer des fichiers `.mo` .

Cas d'utilisation Noop

`(u)gettext_noop` vous permet de marquer une chaîne comme traduisible sans la traduire réellement.

Un cas d'utilisation typique est lorsque vous souhaitez enregistrer un message pour les développeurs (en anglais) mais que vous souhaitez également l'afficher sur le client (dans la langue demandée). **Vous pouvez transmettre une variable à `gettext` , mais son contenu ne sera pas découvert en tant que chaîne traduisible car elle est, par définition, variable.** .

```
# THIS WILL NOT WORK AS EXPECTED
import logging
from django.contrib import messages

logger = logging.getLogger(__name__)

error_message = "Oops, something went wrong!"
logger.error(error_message)
messages.error(request, _(error_message))
```

Le message d'erreur n'apparaîtra pas dans le fichier `.po` et vous devrez vous souvenir qu'il existe pour l'ajouter manuellement. Pour résoudre ce problème, vous pouvez utiliser `gettext_noop` .

```
error_message = ugettext_noop("Oops, something went wrong!")
logger.error(error_message)
messages.error(request, _(error_message))
```

Maintenant la chaîne `"Oops, something went wrong!"` sera découvert et disponible dans le fichier `.po` lors de sa génération. Et l'erreur sera toujours enregistrée en anglais pour les développeurs.

Pièges communs

traductions floues

Parfois, les `makemessages` peuvent penser que la chaîne trouvée pour la traduction est quelque peu similaire à la traduction existante. Il le marquera dans le fichier `.po` avec un commentaire `fuzzy` spécial comme ceci:

```
#: templates/randa/map.html:91
#, fuzzy
msgid "Country"
msgstr "Länderinfo"
```

Même si la traduction est correcte ou si vous l'avez mise à jour pour la corriger, elle ne sera pas utilisée pour traduire votre projet, sauf si vous supprimez `fuzzy` ligne de commentaire `fuzzy` .

Cordes multilignes

`makemessages` analyse les fichiers dans différents formats, du texte brut au code python et n'est pas conçu pour suivre toutes les règles possibles pour avoir des chaînes multi-lignes dans ces formats. La plupart du temps, cela fonctionnera très bien avec des chaînes de caractères simples, mais si vous avez une construction comme celle-ci:

```
translation = _("firstline"
"secondline"
"thirdline")
```

Il ne ramasser `firstline` pour la traduction. La solution consiste à éviter d'utiliser des chaînes multilignes lorsque cela est possible.

Lire Internationalisation en ligne: https://riptutorial.com/fr/django/topic/2579/internationalisation

Chapitre 29: JSONField - un champ spécifique à PostgreSQL

Syntaxe

- JSONField (options **)

Remarques

- `JSONField` de Django `JSONField` fait les données dans une colonne `JSONB` Postgres, uniquement disponible dans Postgres 9.4 et versions ultérieures.
- `JSONField` est idéal lorsque vous souhaitez un schéma plus flexible. Par exemple, si vous souhaitez modifier les clés sans effectuer de migration de données ou si tous vos objets n'ont pas la même structure.
- Si vous stockez des données avec des clés statiques, envisagez d'utiliser plusieurs champs normaux au lieu de `JSONField` s, car interroger `JSONField` peut parfois `JSONField` assez fastidieux.

Enchaînement des requêtes

Vous pouvez enchaîner les requêtes ensemble. Par exemple, si un dictionnaire existe dans une liste, ajoutez deux traits de soulignement et votre requête de dictionnaire.

> N'oubliez pas de séparer les requêtes avec des traits de soulignement doubles.

Examples

Créer un JSONField

Disponible dans Django 1.9+

```
from django.contrib.postgres.fields import JSONField
from django.db import models

class IceCream(models.Model):
    metadata = JSONField()
```

Vous pouvez ajouter les `**options` normales si vous le souhaitez.

> ***!*** Notez que vous devez mettre `'django.contrib.postgres'` dans `INSTALLED_APPS` dans vos `settings.py`

Création d'un objet avec des données dans un JSONField

Passer des données sous forme Python native, par exemple `list` , `dict` , `str` , `None` , `bool` , etc.

```
IceCream.objects.create(metadata={
    'date': '1/1/2016',
    'ordered by': 'Jon Skeet',
    'buyer': {
         'favorite flavor': 'vanilla',
         'known for': ['his rep on SO', 'writing a book']
     },
    'special requests': ['hot sauce'],
})
```

> Voir la note dans la section "Remarques" sur l'utilisation de `JSONField` dans la pratique.

Interrogation des données de niveau supérieur

```
IceCream.objects.filter(metadata__ordered_by='Guido Van Rossum')
```

Interrogation de données imbriquées dans des dictionnaires

Obtenez tous les cornets de glace qui ont été commandés par des gens qui aiment le chocolat:

```
IceCream.objects.filter(metadata__buyer__favorite_flavor='chocolate')
```

> Voir la remarque dans la section "Remarques" sur le chaînage des requêtes.

Interrogation des données présentes dans les tableaux

Un entier sera interprété comme une recherche d'index.

```
IceCream.objects.filter(metadata__buyer__known_for__0='creating stack overflow')
```

> Voir la remarque dans la section "Remarques" sur le chaînage des requêtes.

Classement par valeurs JSONField

La commande directement sur `JSONField` n'est pas encore prise en charge dans Django. Mais c'est possible via RawSQL en utilisant les fonctions PostgreSQL pour jsonb:

```
from django.db.models.expressions import RawSQL
RatebookDataEntry.objects.all().order_by(RawSQL("data->>%s", ("json_objects_key",)))
```

Cet exemple commande par `data['json_objects_key']` dans les `data` nommées `JSONField` :

```
data = JSONField()
```

Lire JSONField - un champ spécifique à PostgreSQL en ligne:
https://riptutorial.com/fr/django/topic/1759/jsonfield---un-champ-specifique-a-postgresql

Chapitre 30: Le débogage

Remarques

Pdb

Pdb peut également imprimer toutes les variables existantes au niveau global ou local, en tapant respectivement l' `globals()` ou `locals()` dans (Pdb).

Examples

Utilisation du débogueur Python (Pdb)

L'outil de débogage de base de Django est pdb , une partie de la bibliothèque standard Python.

Init view script

Examinons un simple script `views.py` :

```
from django.http import HttpResponse


def index(request):
    foo = 1
    bar = 0

    bug = foo/bar

    return HttpResponse("%d goes here." % bug)
```

Commande de la console pour exécuter le serveur:

```
python manage.py runserver
```

Il est évident que Django `ZeroDivisionError` une `ZeroDivisionError` lorsque vous essayez de charger une page d'index, mais si nous prétendons que le bogue est très profond dans le code, cela peut devenir vraiment désagréable.

Définition d'un point d'arrêt

Heureusement, nous pouvons définir un *point* d' *arrêt* pour tracer ce bogue:

```
from django.http import HttpResponse

# Pdb import
import pdb


def index(request):
```

```
    foo = 1
    bar = 0

    # This is our new breakpoint
    pdb.set_trace()

    bug = foo/bar

    return HttpResponse("%d goes here." % bug)
```

Commande de console pour exécuter le serveur avec pdb:

```
python -m pdb manage.py runserver
```

Désormais, le point d'arrêt de chargement de la page déclenchera l'invite (Pdb) dans le shell, ce qui bloquera également votre navigateur en attente.

Débogage avec le shell pdb

Il est temps de déboguer cette vue en interagissant avec le script via le shell:

```
> ../views.py(12)index()
-> bug = foo/bar
# input 'foo/bar' expression to see division results:
(Pdb) foo/bar
*** ZeroDivisionError: division by zero
# input variables names to check their values:
(Pdb) foo
1
(Pdb) bar
0
# 'bar' is a source of the problem, so if we set it's value > 0...
(Pdb) bar = 1
(Pdb) foo/bar
1.0
# exception gone, ask pdb to continue execution by typing 'c':
(Pdb) c
[03/Aug/2016 10:50:45] "GET / HTTP/1.1" 200 111
```

Dans la dernière ligne, nous voyons que notre vue a retourné une réponse `OK` et qu’elle s’exécutait comme il se doit.

Pour arrêter la boucle pdb, entrez simplement `q` dans un shell.

Utiliser la barre d'outils de débogage de Django

Tout d'abord, vous devez installer django-debug-toolbar :

```
pip install django-debug-toolbar
```

settings.py :

Ensuite, incluez-le aux applications installées du projet, mais faites attention: il est toujours

recommandé d'utiliser un fichier `settings.py` différent pour les applications et les middlewares de développement uniquement comme barre d'outils de débogage:

```
# If environment is dev...
DEBUG = True

INSTALLED_APPS += [
    'debug_toolbar',
]

MIDDLEWARE += ['debug_toolbar.middleware.DebugToolbarMiddleware']
```

La barre d'outils de débogage repose également sur des fichiers statiques, de sorte que l'application appropriée devrait également être incluse:

```
INSTALLED_APPS = [
    # ...
    'django.contrib.staticfiles',
    # ...
]

STATIC_URL = '/static/'

# If environment is dev...
DEBUG = True

INSTALLED_APPS += [
    'debug_toolbar',
]
```

Dans certains cas, il est également nécessaire de définir `INTERNAL_IPS` dans `settings.py` :

```
INTERNAL_IPS = ('127.0.0.1', )
```

urls.py :

Dans `urls.py` , comme le suggère la documentation officielle, l'extrait suivant doit permettre le routage de la barre d'outils de débogage:

```
if settings.DEBUG and 'debug_toolbar' in settings.INSTALLED_APPS:
    import debug_toolbar
    urlpatterns += [
        url(r'^__debug__/', include(debug_toolbar.urls)),
    ]
```

Recueillir statique de la barre d'outils après l'installation:

```
python manage.py collectstatic
```

Ça y est, la barre d'outils de débogage apparaîtra sur les pages de votre projet, fournissant diverses informations utiles sur le temps d'exécution, SQL, les fichiers statiques, les signaux, etc.

HTML:

De plus, `django-debug-toolbar` nécessite un rendu correct des balises `text/html` , `<html>` et `<body>` de *type Content* .

Dans le cas où vous êtes sûr d'avoir tout configuré correctement, mais que la barre d'outils de débogage n'est toujours pas rendue: utilisez cette solution "nucléaire" pour essayer de la comprendre.

Utiliser "assert False"

En développant, en insérant la ligne suivante dans votre code:

```
assert False, value
```

django déclenchera une `AssertionError` avec la valeur fournie en tant que message d'erreur lorsque cette ligne est exécutée.

Si cela se produit dans une vue, ou dans tout code appelé depuis une vue et que `DEBUG=True` est défini, une pile complète et détaillée contenant beaucoup d'informations de débogage sera affichée dans le navigateur.

N'oubliez pas de supprimer la ligne lorsque vous avez terminé!

Envisagez d'écrire plus de documentation, de tests, de journalisation et d'assertions au lieu d'utiliser un débogueur

Le débogage demande du temps et des efforts.

Au lieu de chasser les bogues avec un débogueur, envisagez de passer plus de temps à améliorer votre code en:

- **Écrire et exécuter des tests .** Python et Django ont d'excellentes structures de test intégrées - qui peuvent être utilisées pour tester votre code beaucoup plus rapidement qu'avec un débogueur manuel.
- **Rédaction de la documentation appropriée** pour vos fonctions, classes et modules. PEP 257 et le guide de style Python de Google fournissent de bonnes pratiques pour écrire de bons documents.
- **Utilisez la journalisation** pour générer une sortie de votre programme - pendant le développement et après le déploiement.
- **Ajouter `assert` ions** à votre code dans des lieux importants: Réduire l' ambiguïté, attraper des problèmes car ils sont créés.

Bonus: Ecrivez des certificats pour combiner documentation et test!

Lire Le débogage en ligne: https://riptutorial.com/fr/django/topic/5072/le-debogage

Chapitre 31: Les signaux

Paramètres

Classe / méthode	Le pourquoi
Classe UserProfile ()	La classe UserProfile étend le modèle d'utilisateur par défaut de Django .
Méthode create_profile ()	La méthode create_profile () est exécutée chaque fois qu'un signal `post_save` modèle d'utilisateur Django est libéré .

Remarques

Maintenant, les détails.

Les signaux de Django sont un moyen d'informer votre application de certaines tâches (telles qu'un modèle avant ou après la sauvegarde ou la suppression) lorsqu'elles ont lieu.

Ces signaux vous permettent d'effectuer des actions de votre choix immédiatement après la sortie du signal.

Par exemple, **chaque fois** qu'un nouvel utilisateur Django est créé, le modèle utilisateur libère un signal, en associant des paramètres tels que `sender=User` vous permettant de cibler spécifiquement votre écoute de signaux vers une activité spécifique, dans ce cas, une nouvelle création d'utilisateur. .

Dans l'exemple ci-dessus, l'intention est de créer un objet UserProfile *immédiatement* après la création d'un objet User. Par conséquent, en écoutant un signal `post_save` à partir du modèle `User` (le modèle utilisateur Django par défaut), nous créons un objet `UserProfile` juste après la création d'un nouvel `User` .

La documentation de Django fournit une documentation complète sur tous les signaux possibles disponibles .

Toutefois, l’exemple ci-dessus explique en termes pratiques un cas d’utilisation typique lorsqu’on utilise les signaux peut être un ajout utile.

"Un grand pouvoir implique de grandes responsabilités". Il peut être tentant d'avoir des signaux dispersés dans toute votre application ou projet, simplement parce qu'ils sont géniaux. Eh bien non. Parce qu'ils sont cool, ils ne sont pas la solution idéale pour toutes les situations simples qui leur viennent à l'esprit.

Les signaux sont parfaits pour, comme d'habitude, pas tout. Login / Logouts, les signaux sont excellents. Les modèles clés libèrent des signes, comme le modèle utilisateur, si cela convient.

La création de signaux pour chaque modèle de votre application peut être écrasante à un moment donné et faire échec à l'idée générale de l'utilisation des signaux Django.

N'utilisez pas de signaux lorsque (basé sur le livre Two Scoops of Django):

- Le signal se rapporte à un modèle particulier et peut être déplacé dans l'une des méthodes de ce modèle, éventuellement appelée par `save()` .
- Le signal peut être remplacé par une méthode de gestionnaire de modèles personnalisée.
- Le signal se rapporte à une vue particulière et peut être déplacé dans cette vue

Il peut être correct d'utiliser des signaux lorsque:

- Votre récepteur de signal doit apporter des modifications à plusieurs modèles.
- Vous souhaitez envoyer le même signal depuis plusieurs applications et les faire traiter de la même manière par un récepteur commun.
- Vous souhaitez invalider un cache après une sauvegarde de modèle.
- Vous avez un scénario inhabituel qui nécessite un rappel, et il n'y a pas d'autre moyen de le gérer que d'utiliser un signal. Par exemple, vous voulez déclencher quelque chose en fonction de `save()` ou `init()` du modèle d'une application tierce. Vous ne pouvez pas modifier le code tiers et l'étendre peut être impossible. Un signal déclenche donc un rappel.

Examples

Extension de l'exemple de profil utilisateur

Cet exemple est un extrait tiré du profil utilisateur Extending Django comme un pro

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save

class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)

def create_profile(sender, **kwargs):
    user = kwargs["instance"]
    if kwargs["created"]:
        user_profile = UserProfile(user=user)
        user_profile.save()
post_save.connect(create_profile, sender=User)
```

Syntaxe différente pour poster / pré-signaler

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

class UserProfile(models.Model):
```

```
    user = models.OneToOneField(User, related_name='user')
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)

@receiver(post_save, sender=UserProfile)
def post_save_user(sender, **kwargs):
    user = kwargs.get('instance')
    if kwargs.get('created'):
        ...
```

Comment trouver s'il s'agit d'une insertion ou d'une mise à jour dans le signal pre_save

En utilisant `pre_save` nous pouvons déterminer si une action de `save` sur notre base de données consistait à mettre à jour un objet existant ou à en créer un nouveau.

Pour ce faire, vous pouvez vérifier l'état de l'objet du modèle:

```
    @receiver(pre_save, sender=User)
    def pre_save_user(sender, instance, **kwargs):
        if not instance._state.adding:
            print ('this is an update')
        else:
            print ('this is an insert')
```

Maintenant, chaque fois qu'une action de `save` a lieu, le signal `pre_save` sera exécuté et imprimera:

- `this is an update` si l'action dérive d'une action de mise à jour.
- `this is an insert` si l'action dérive d'une action d'insertion.

Notez que cette méthode ne nécessite aucune requête de base de données supplémentaire.

Héritage des signaux sur les modèles étendus

Les signaux de Django sont limités à des signatures de classe précises lors de l'enregistrement, et les modèles sous-classés ne sont donc pas immédiatement enregistrés sur le même signal.

Prenez ce modèle et signalez par exemple

```
class Event(models.Model):
    user = models.ForeignKey(User)


class StatusChange(Event):
    ...


class Comment(Event):
    ...


def send_activity_notification(sender, instance: Event, raw: bool, **kwargs):
```

```
    """
    Fire a notification upon saving an event
    """

    if not raw:
        msg_factory = MessageFactory(instance.id)
        msg_factory.on_activity(str(instance))
post_save.connect(send_activity_notification, Event)
```

Avec les modèles étendus, vous devez attacher manuellement le signal sur chaque sous-classe, sinon ils ne seront pas effectués.

```
post_save.connect(send_activity_notification, StatusChange)
post_save.connect(send_activity_notification, Comment)
```

Avec Python 3.6, vous pouvez tirer parti de certaines méthodes de classe supplémentaires intégrées aux classes pour automatiser cette liaison.

```
class Event(models.Model):

    @classmethod
    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        post_save.connect(send_activity_notification, cls)
```

Lire Les signaux en ligne: https://riptutorial.com/fr/django/topic/2555/les-signaux

Chapitre 32: Mapper des chaînes à des chaînes avec HStoreField - un champ spécifique à PostgreSQL

Syntaxe

- FooModel.objects.filter (nom_zone__nom_key = 'valeur à interroger')

Examples

Configuration de HStoreField

Tout d'abord, nous devrons faire quelques réglages pour que `HStoreField` fonctionne.

1. assurez-vous que `django.contrib.postgres` est dans votre `INSTALLED_APPS
2. Ajoutez `HStoreExtension` à vos migrations. N'oubliez pas de placer `HStoreExtension` avant toute migration `CreateModel` ou `AddField` .

```
from django.contrib.postgres.operations import HStoreExtension
from django.db import migrations

class FooMigration(migrations.Migration):
    # put your other migration stuff here
    operations = [
        HStoreExtension(),
        ...
    ]
```

Ajout de HStoreField à votre modèle

> `->` Remarque: assurez-vous de configurer `HStoreField` avant de continuer avec cet exemple. (au dessus)

Aucun paramètre n'est requis pour initialiser un `HStoreField` .

```
from django.contrib.postgres.fields import HStoreField
from django.db import models

class Catalog(models.model):
    name = models.CharField(max_length=200)
    titles_to_authors = HStoreField()
```

Créer une nouvelle instance de modèle

Passer un dictionnaire Python natif mappant des chaînes de caractères à `create()` .

```
Catalog.objects.create(name='Library of Congress', titles_to_authors={
    'Using HStoreField with Django': 'CrazyPython and la communidad',
    'Flabbergeists and thingamajigs': 'La Artista Fooista',
    'Pro Git': 'Scott Chacon and Ben Straub',
})
```

Effectuer des recherches de clés

```
Catalog.objects.filter(titles__Pro_Git='Scott Chacon and Ben Straub')
```

Utiliser contient

`field_name__contains` un objet `dict` à `field_name__contains` comme argument de mot clé.

```
Catalog.objects.filter(titles__contains={
        'Pro Git': 'Scott Chacon and Ben Straub'})
```

Équivalent à l'opérateur SQL `@>`.

Lire Mapper des chaînes à des chaînes avec HStoreField - un champ spécifique à PostgreSQL en ligne: https://riptutorial.com/fr/django/topic/2670/mapper-des-chaines-a-des-chaines-avec-hstorefield---un-champ-specifique-a-postgresql

Chapitre 33: Meta: Guide de documentation

Remarques

Ceci est une extension des "Meta: Documentation Guidelines" de Python pour Django.

Ce ne sont que des propositions et non des recommandations. N'hésitez pas à modifier n'importe quoi ici si vous n'êtes pas d'accord ou si vous avez quelque chose à mentionner.

Examples

Les versions non prises en charge ne nécessitent pas de mention spéciale

Il est peu probable que quelqu'un utilise une version non prise en charge de Django, et à ses risques et périls. Si jamais quelqu'un le fait, ce doit être son souci de savoir si une fonctionnalité existe dans la version donnée.

Compte tenu de ce qui précède, il est inutile de mentionner les spécificités d'une version non prise en charge.

1.6

Ce type de bloc est inutile car aucune personne sensée n'utilise Django <1.6.

1.8

Ce type de bloc est inutile car aucune personne sensée n'utilise Django <1.8.

Cela vaut également pour les sujets. Au moment de la rédaction de cet exemple, les vues basées sur la classe indiquent que les versions prises en charge sont `1.3-1.9` . Nous pouvons supposer que cela est en fait équivalent à `All versions` . Cela évite également de mettre à niveau toutes les versions prises en charge à chaque nouvelle version.

> Les versions actuelles prises en charge sont: `1.8` [1] `1.9` [2] `1.10` [1]

1. Les correctifs de sécurité, les bogues de perte de données, les bogues en panne, les bogues majeurs des fonctionnalités dans les nouvelles fonctionnalités et les régressions des anciennes versions de Django.
2. Correctifs de sécurité et bogues de perte de données.

Lire Meta: Guide de documentation en ligne: https://riptutorial.com/fr/django/topic/5243/meta--guide-de-documentation

Chapitre 34: Middleware

Introduction

Middleware dans Django est un framework qui permet au code d’accéder au traitement des réponses / requêtes et de modifier l’entrée ou la sortie de Django.

Remarques

Le middleware doit être ajouté à votre liste settings.py `MIDDLEWARE_CLASSES` avant d'être inclus dans l'exécution. La liste par défaut fournie par Django lors de la création d'un nouveau projet est la suivante:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Ce sont toutes les fonctions qui s'exécuteront dans l' **ordre** à chaque demande (une fois avant d'atteindre votre code de vue dans `views.py` et une fois dans l'ordre inverse pour le rappel `process_response` , avant la version 1.10). Ils font une variété de choses telles que l'injection du jeton Cross Site Request Forgery (csrf) .

L'ordre est important car si certains middleware font une redirection, alors tous les middleware suivants ne seront jamais exécutés. Ou si un middleware s'attend à ce que le jeton csrf soit présent, il doit être exécuté après le `CsrfViewMiddleware` .

Examples

Ajouter des données aux requêtes

Django facilite l'ajout de données supplémentaires sur les demandes d'utilisation dans la vue. Par exemple, nous pouvons analyser le sous-domaine sur le META de la demande et le joindre en tant que propriété distincte sur la demande en utilisant un middleware.

```
class SubdomainMiddleware:
    def process_request(self, request):
        """
        Parse out the subdomain from the request
        """
        host = request.META.get('HTTP_HOST', '')
        host_s = host.replace('www.', '').split('.')
```

```
        request.subdomain = None
        if len(host_s) > 2:
            request.subdomain = host_s[0]
```

Si vous ajoutez des données avec un middleware à votre demande, vous pouvez accéder aux nouvelles données ajoutées ultérieurement. Ici, nous utiliserons le sous-domaine analysé pour déterminer quelque chose comme l'organisation accédant à votre application. Cette approche est utile pour les applications déployées avec une configuration DNS avec des sous-domaines génériques qui pointent tous vers une seule instance et la personne qui accède à l'application souhaite qu'une version avec habillage dépend du point d'accès.

```
class OrganizationMiddleware:
    def process_request(self, request):
        """
        Determine the organization based on the subdomain
        """
        try:
            request.org = Organization.objects.get(domain=request.subdomain)
        except Organization.DoesNotExist:
            request.org = None
```

Rappelez-vous que l'ordre est important lorsque le middleware dépend l'un de l'autre. Pour les requêtes, vous voudrez que le middleware dépendant soit placé après la dépendance.

```
MIDDLEWARE_CLASSES = [
    ...
    'myapp.middleware.SubdomainMiddleware',
    'myapp.middleware.OrganizationMiddleware',
    ...
]
```

Middleware pour filtrer par adresse IP

Premier: la structure du chemin

Si vous ne l'avez pas, vous devez créer le dossier du **middleware** dans votre application en suivant la structure:

```
yourproject/yourapp/middleware
```

Le middleware de dossier doit être placé dans le même dossier que settings.py, urls, templates ...

Important: n'oubliez pas de créer le fichier vide init .py dans le dossier du middleware afin que votre application reconnaisse ce dossier

Au lieu d'avoir un dossier séparé contenant vos classes de middleware, il est également possible de placer vos fonctions dans un seul fichier, `yourproject/yourapp/middleware.py` .

Deuxièmement: créer le middleware

Nous devons maintenant créer un fichier pour notre middleware personnalisé. Dans cet exemple, supposons que nous voulons un middleware qui filtre les utilisateurs en fonction de leur adresse IP, nous créons un fichier appelé **filter_ip_middleware.py** :

```
#yourproject/yourapp/middleware/filter_ip_middleware.py
from django.core.exceptions import PermissionDenied

class FilterIPMiddleware(object):
    # Check if client IP address is allowed
    def process_request(self, request):
        allowed_ips = ['192.168.1.1', '123.123.123.123', etc...] # Authorized ip's
        ip = request.META.get('REMOTE_ADDR') # Get client IP address
        if ip not in allowed_ips:
            raise PermissionDenied # If user is not allowed raise Error

       # If IP address is allowed we don't do anything
       return None
```

Troisièmement: ajoutez le middleware dans nos "settings.py"

Nous devons rechercher `MIDDLEWARE_CLASSES` dans le `MIDDLEWARE_CLASSES` settings.py et ajouter notre middleware (*ajoutez-le à la dernière position*). Cela devrait être comme:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
     # Above are Django standard middlewares

     # Now we add here our custom middleware
     'yourapp.middleware.filter_ip_middleware.FilterIPMiddleware'
)
```

Terminé! Maintenant, chaque demande de chaque client appelle votre middleware personnalisé et traite votre code personnalisé!

Exception de traitement global

Supposons que vous ayez implémenté une logique pour détecter les tentatives de modification d'un objet dans la base de données alors que le client qui a soumis les modifications n'a pas les dernières modifications. Dans ce cas, vous `ConfictError(detailed_message)` une exception personnalisée `ConfictError(detailed_message)` .

Maintenant, vous voulez retourner un code d'état HTTP 409 (Confict) lorsque cette erreur se produit. Vous pouvez généralement utiliser comme middleware pour cela au lieu de le gérer dans chaque vue susceptible de générer cette exception.

```
class ConfictErrorHandlingMiddleware:
    def process_exception(self, request, exception):
        if not isinstance(exception, ConflictError):
            return  # Propagate other exceptions, we only handle ConflictError
```

```
        context = dict(confict_details=str(exception))
        return TemplateResponse(request, '409.html', context, status=409)
```

Comprendre le nouveau style du middleware Django 1.10

Django 1.10 a introduit un nouveau style de middleware où `process_request` et `process_response` sont fusionnés.

Dans ce nouveau style, *un middleware est un appelant qui renvoie un autre appelable* . Eh bien, en réalité, le **premier est une usine de middleware** et **le second est le véritable middleware** .

La *fabrique de middleware* prend comme argument unique le *middleware* suivant dans la pile des middlewares, ou la vue elle-même lorsque le bas de la pile est atteint.

Le *middleware* prend la requête en argument unique et **retourne toujours un `HttpResponse`** .

Le meilleur exemple pour illustrer le fonctionnement de nouveaux *middlewares* est probablement de montrer comment créer un *middleware à* compatibilité descendante:

```
class MyMiddleware:

    def __init__(self, next_layer=None):
        """We allow next_layer to be None because old-style middlewares
        won't accept any argument.
        """
        self.get_response = next_layer

    def process_request(self, request):
        """Let's handle old-style request processing here, as usual."""
        # Do something with request
        # Probably return None
        # Or return an HttpResponse in some cases

    def process_response(self, request, response):
        """Let's handle old-style response processing here, as usual."""
        # Do something with response, possibly using request.
        return response

    def __call__(self, request):
        """Handle new-style middleware here."""
        response = self.process_request(request)
        if response is None:
            # If process_request returned None, we must call the next middleware or
            # the view. Note that here, we are sure that self.get_response is not
            # None because this method is executed only in new-style middlewares.
            response = self.get_response(request)
        response = self.process_response(request, response)
        return response
```

Lire Middleware en ligne: https://riptutorial.com/fr/django/topic/1721/middleware

Chapitre 35: Migrations

Paramètres

django-admin	Détails
makemigrations <my_app>	Générer des migrations pour my_app
makemigrations	Générer des migrations pour toutes les applications
makemigrations --merge	Résoudre les conflits de migration pour toutes les applications
makemigrations --merge <my_app>	Résoudre les conflits de migration pour my_app
makemigrations --name <migration_name> <my_app>	Générer une migration pour my_app avec le nom migration_name
migrate <my_app>	Appliquer les migrations en attente de my_app à la base de données
migrate	Appliquer toutes les migrations en attente à la base de données
migrate <my_app> <migration_name>	Appliquer ou non à la migration_name
migrate <my_app> zero	Inapplicable toutes les migrations dans my_app
sqlmigrate <my_app> <migration_name>	Imprime le SQL pour la migration nommée
showmigrations	Affiche toutes les migrations pour toutes les applications
showmigrations <my_app>	Affiche toutes les migrations dans my_app

Examples

Travailler avec des migrations

Django utilise les migrations pour propager les modifications apportées à vos modèles dans votre base de données. La plupart du temps, Django peut les générer pour vous.

Pour créer une migration, exécutez:

```
$ django-admin makemigrations <app_name>
```

Cela créera un fichier de migration dans le sous-module de `migration app_name` . La première migration s'appellera `0001_initial.py` , l'autre commencera par `0002_` , puis `0003` , ...

Si vous omettez `<app_name>` cela créera des migrations pour tous vos `INSTALLED_APPS` .

Pour propager les migrations vers votre base de données, exécutez:

```
$ django-admin migrate <app_name>
```

Pour afficher toutes vos migrations, exécutez:

```
$ django-admin showmigrations app_name
app_name
  [X] 0001_initial
  [X] 0002_auto_20160115_1027
  [X] 0003_somemodel
  [ ] 0004_auto_20160323_1826
```

- `[X]` signifie que la migration a été propagée vers votre base de données
- `[ ]` signifie que la migration n'a pas été propagée à votre base de données. Utilisez `django-admin migrate` pour le propager

Vous appelez également les migrations de retour, cela peut être fait en transmettant le nom de la migration à la `migrate command` . Compte tenu de la liste ci-dessus des migrations (montrée par `django-admin showmigrations`):

```
$ django-admin migrate app_name 0002  # Roll back to migration 0002
$ django-admin showmigrations app_name
app_name
  [X] 0001_initial
  [X] 0002_auto_20160115_1027
  [ ] 0003_somemodel
  [ ] 0004_auto_20160323_1826
```

Migrations manuelles

Parfois, les migrations générées par Django ne sont pas suffisantes. Cela est particulièrement vrai lorsque vous souhaitez effectuer **des migrations de données** .

Par exemple, vous avez un tel modèle:

```
class Article(models.Model):
    title = models.CharField(max_length=70)
```

Ce modèle possède déjà des données existantes et vous souhaitez maintenant ajouter un `SlugField` :

```
class Article(models.Model):
    title = models.CharField(max_length=70)
    slug = models.SlugField(max_length=70)
```

Vous avez créé les migrations pour ajouter le champ, mais vous souhaitez maintenant définir le slug pour tous les articles existants, en fonction de leur `title` .

Bien sûr, vous pourriez faire quelque chose comme ça dans le terminal:

```
$ django-admin shell
>>> from my_app.models import Article
>>> from django.utils.text import slugify
>>> for article in Article.objects.all():
...     article.slug = slugify(article.title)
...     article.save()
...
>>>
```

Mais vous devrez le faire dans tous vos environnements (c.-à-d. Votre ordinateur de bureau, votre ordinateur portable, etc.), tous vos collègues devront le faire également, et vous devrez y réfléchir lors de la mise en scène et de la poussée. vivre.

Pour le faire une fois pour toutes, nous le ferons dans une migration. Commencez par créer une migration vide:

```
$ django-admin makemigrations --empty app_name
```

Cela créera un fichier de migration vide. Ouvrez-le, il contient un squelette de base. Disons que votre migration précédente s'appelait `0023_article_slug` et que celle-ci s'appelle `0024_auto_20160719_1734` . Voici ce que nous allons écrire dans notre fichier de migration:

```
# -*- coding: utf-8 -*-
# Generated by Django 1.9.7 on 2016-07-19 15:34
from __future__ import unicode_literals

from django.db import migrations
from django.utils.text import slugify


def gen_slug(apps, schema_editor):
    # We can't import the Article model directly as it may be a newer
    # version than this migration expects. We use the historical version.
    Article = apps.get_model('app_name', 'Article')
    for row in Article.objects.all():
        row.slug = slugify(row.name)
        row.save()


class Migration(migrations.Migration):

    dependencies = [
        ('hosting', '0023_article_slug'),
    ]

    operations = [
        migrations.RunPython(gen_slug, reverse_code=migrations.RunPython.noop),
        # We set `reverse_code` to `noop` because we cannot revert the migration
        # to get it back in the previous state.
        # If `reverse_code` is not given, the migration will not be reversible,
```

```
        # which is not the behaviour we expect here.
    ]
```

Fausses migrations

Lorsqu'une migration est exécutée, Django stocke le nom de la migration dans une table django_migrations.

Créer et simuler des migrations initiales pour un schéma existant

Si votre application a déjà des modèles et des tables de base de données et ne dispose pas de migrations. Commencez par créer des migrations initiales pour votre application.

```
python manage.py makemigrations your_app_label
```

Faux migrations initiales maintenant appliquées

```
python manage.py migrate --fake-initial
```

Faux toutes les migrations dans toutes les applications

```
python manage.py migrate --fake
```

Fausses migrations d'applications uniques

```
python manage.py migrate --fake core
```

Faux fichier de migration unique

```
python manage.py migrate myapp migration_name
```

Noms personnalisés pour les fichiers de migration

Utilisez l' `makemigrations --name <your_migration_name>` pour autoriser la dénomination des migrations au lieu d'utiliser un nom généré.

```
python manage.py makemigrations --name <your_migration_name> <app_name>
```

Résoudre les conflits de migration

introduction

Parfois, les migrations sont en conflit, ce qui rend la migration infructueuse. Cela peut se produire dans de nombreux scénarios, mais cela peut se produire régulièrement lors du développement d'une application avec une équipe.

Les conflits de migration courants se produisent lors de l'utilisation du contrôle de source, en particulier lorsque la méthode de fonctionnalité par branche est utilisée. Pour ce scénario, nous utiliserons un modèle appelé `Reporter` avec le `name` et l' `address` attributs.

Deux développeurs à ce stade vont développer une fonctionnalité, ils obtiennent donc cette copie initiale du modèle `Reporter` . Le développeur A ajoute un `age` qui entraîne le fichier `0002_reporter_age.py` . Le développeur B ajoute un champ `bank_account` qui `0002_reporter_bank_account` . Une fois que ces développeurs fusionnent leur code et tentent de migrer les migrations, un conflit de migration s'est produit.

Ce conflit se produit car ces migrations modifient le même modèle, `Reporter` . De plus, les nouveaux fichiers commencent tous deux par 0002.

Fusion de migrations

Il y a plusieurs façons de le faire. Ce qui suit est dans l'ordre recommandé:

1. La solution la plus simple consiste à exécuter la commande makemigrations avec un indicateur --merge.

   ```
   python manage.py makemigrations --merge <my_app>
   ```

   Cela créera une nouvelle migration pour résoudre le conflit précédent.

2. Lorsque ce fichier supplémentaire n'est pas le bienvenu dans l'environnement de développement pour des raisons personnelles, vous pouvez supprimer les migrations en conflit. Ensuite, une nouvelle migration peut être effectuée à l'aide de la commande `makemigrations` . Lorsque des migrations personnalisées sont écrites, telles que `migrations.RunPython` , vous devez tenir compte de cette méthode.

Changer un CharField en un ForeignKey

Tout d'abord, supposons qu'il s'agisse de votre modèle initial, à l'intérieur d'une application appelée `discography` :

```
from django.db import models

class Album(models.Model):
    name = models.CharField(max_length=255)
    artist = models.CharField(max_length=255)
```

Maintenant, vous vous rendez compte que vous voulez utiliser un ForeignKey pour l'artiste à la place. C'est un processus assez complexe, qui doit être fait en plusieurs étapes.

Étape 1, ajoutez un nouveau champ pour ForeignKey, en veillant à le marquer comme nul (notez que le modèle auquel nous sommes liés est également maintenant inclus):

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

```
class Album(models.Model):
    name = models.CharField(max_length=255)
    artist = models.CharField(max_length=255)
    artist_link = models.ForeignKey('Artist', null=True)

class Artist(models.Model):
    name = models.CharField(max_length=255)
```

... et créer une migration pour ce changement.

```
./manage.py makemigrations discography
```

Étape 2, remplissez votre nouveau champ. Pour ce faire, vous devez créer une migration vide.

```
./manage.py makemigrations --empty --name transfer_artists discography
```

Une fois que vous avez cette migration vide, vous souhaitez y ajouter une seule opération `RunPython` afin de lier vos enregistrements. Dans ce cas, cela pourrait ressembler à ceci:

```
def link_artists(apps, schema_editor):
    Album = apps.get_model('discography', 'Album')
    Artist = apps.get_model('discography', 'Artist')
    for album in Album.objects.all():
        artist, created = Artist.objects.get_or_create(name=album.artist)
        album.artist_link = artist
        album.save()
```

Maintenant que vos données sont transférées dans le nouveau champ, vous pouvez réellement le faire et tout laisser tel `artist_link` , en utilisant le nouveau champ `artist_link` pour tout. Ou, si vous souhaitez effectuer un peu de nettoyage, vous souhaitez créer deux autres migrations.

Pour votre première migration, vous souhaitez supprimer votre champ d'origine, `artist` . Pour votre deuxième migration, renommez le nouveau champ `artist_link` en `artist` .

Cela se fait en plusieurs étapes pour s'assurer que Django reconnaît correctement les opérations.

Lire Migrations en ligne: https://riptutorial.com/fr/django/topic/1200/migrations

Chapitre 36: Paramètres

Examples

Définition du fuseau horaire

Vous pouvez définir le fuseau horaire qui sera utilisé par Django dans le fichier `settings.py` . Exemples:

```
TIME_ZONE = 'UTC'  # use this, whenever possible
TIME_ZONE = 'Europe/Berlin'
TIME_ZONE = 'Etc/GMT+1'
```

Voici la liste des fuseaux horaires valables

*Lors de l'exécution dans un environnement **Windows** , cette zone doit être identique à celle du **fuseau horaire de** votre **système** .*

Si vous ne voulez pas que Django utilise des dates-heures sensibles au fuseau horaire:

```
USE_TZ = False
```

Les meilleures pratiques de Django demandent d'utiliser `UTC` pour stocker des informations dans la base de données:

> Même si votre site Web est disponible dans un seul fuseau horaire, il est toujours recommandé de stocker les données en UTC dans votre base de données. La raison principale est l'heure d'été (DST). De nombreux pays ont un système de DST, où les horloges sont avancées au printemps et en arrière en automne. Si vous travaillez à l'heure locale, vous risquez de rencontrer des erreurs deux fois par an, lorsque les transitions se produisent.

https://docs.djangoproject.com/en/stable/topics/i18n/timezones/

Accéder aux paramètres

Une fois que vous avez tous vos paramètres, vous voudrez les utiliser dans votre code. Pour cela, ajoutez l'importation suivante à votre fichier:

```
from django.conf import settings
```

Vous pouvez ensuite accéder à vos paramètres en tant qu'attributs du module de `settings` , par exemple:

```
if not settings.DEBUG:
    email_user(user, message)
```

Utiliser BASE_DIR pour assurer la portabilité des applications

Il est déconseillé de coder des chemins dans votre application. Il faut toujours utiliser des URL relatives pour que votre code puisse fonctionner de manière transparente sur différentes machines. La meilleure façon de configurer cela est de définir une variable comme celle-ci.

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(__file__))
```

Ensuite, utilisez cette variable `BASE_DIR` pour définir tous vos autres paramètres.

```
TEMPLATE_PATH = os.path.join(BASE_DIR, "templates")
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),

]
```

Etc. Cela garantit que vous pouvez porter votre code sur différentes machines sans aucun souci.

Cependant, `os.path` est un peu verbeux. Par exemple, si votre module de paramètres est `project.settings.dev` , vous devrez écrire:

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.dirname(__file__)))
```

Une alternative consiste à utiliser le module `unipath` (que vous pouvez installer avec `pip install unipath`).

```
from unipath import Path

BASE_DIR = Path(__file__).ancestor(2)  # or ancestor(3) if using a submodule

TEMPLATE_PATH = BASE_DIR.child('templates')
STATICFILES_DIRS = [
    BASE_DIR.child('static'),
]
```

Utilisation de variables d'environnement pour gérer les paramètres sur les serveurs

L'utilisation de variables d'environnement est un moyen largement utilisé pour définir la configuration d'une application en fonction de son environnement, comme indiqué dans l'application Twelve-Factor .

Comme les configurations sont susceptibles de changer entre les environnements de déploiement, c'est une manière très intéressante de modifier la configuration sans avoir à creuser le code source de l'application, ainsi qu'à garder des secrets en dehors des fichiers d'application et du référentiel de code source.

Dans Django, les paramètres principaux se trouvent sous le nom `settings.py` dans le dossier de

votre projet. Comme il s'agit d'un simple fichier Python, vous pouvez utiliser le module `os` de Python depuis la bibliothèque standard pour accéder à l'environnement (et même disposer des paramètres par défaut appropriés).

settings.py

```
import os

SECRET_KEY = os.environ.get('APP_SECRET_KEY', 'unsafe-secret-key')

DEBUG = bool(os.environ.get('DJANGO_DEBUG', True) == 'False')

ALLOWED_HOSTS = os.environ.get('DJANGO_ALLOWED_HOSTS', '').split()

DATABASES = {
    'default': {
        'ENGINE': os.environ.get('APP_DB_ENGINE', 'django.db.backends.sqlite3'),
        'NAME': os.environ.get('DB_NAME', 'db.sqlite'),
        'USER': os.environ.get('DB_USER', ''),
        'PASSWORD': os.environ.get('DB_PASSWORD', ''),
        'HOST': os.environ.get('DB_HOST', None),
        'PORT': os.environ.get('DB_PORT', None),
        'CONN_MAX_AGE': 600,
    }
}
```

Avec Django, vous pouvez modifier la technologie de votre base de données, de sorte que vous puissiez utiliser sqlite3 sur votre machine de développement (ce qui devrait être une option saine pour s’engager dans un système de contrôle de source). Bien que cela soit possible, il est déconseillé:

> Les services de support, tels que la base de données de l'application, le système de mise en file d'attente ou le cache, constituent un domaine où la parité dev / prod est importante. (The Twelve-Factor App - Parité Dev / prod)

Pour utiliser un paramètre DATABASE_URL pour la connexion à la base de données, consultez l' exemple associé .

Utiliser plusieurs paramètres

La structure de projet par défaut de Django crée un seul `settings.py` . Ceci est souvent utile pour le diviser comme ceci:

```
myprojectroot/
    myproject/
        __init__.py
        settings/
            __init__.py
            base.py
            dev.py
            prod.py
            tests.py
```

Cela vous permet de travailler avec différents paramètres selon que vous soyez en développement, en production, en tests ou autre.

Lorsque vous passez de la présentation par défaut à cette présentation, le `settings/base.py` `settings.py` origine devient `settings/base.py` Lorsque tous les autres sous-modules "sous `settings/base.py` " `settings/base.py` en commençant par à partir `from .base import *` . Par exemple, voici à quoi peuvent ressembler `settings/dev.py` :

```
# -*- coding: utf-8 -*-
from .base import *  # noqa

DEBUG = True
INSTALLED_APPS.extend([
    'debug_toolbar',
])
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
INTERNAL_IPS = ['192.168.0.51', '192.168.0.69']
```

Alternative n ° 1

Pour `django-admin` commandes `django-admin` fonctionnent correctement, vous devrez définir la variable d'environnement `DJANGO_SETTINGS_MODULE` (qui par défaut est `myproject.settings`). En développement, vous allez le définir sur `myproject.settings.dev` . En production, vous le définissez sur `myproject.settings.prod` . Si vous utilisez un virtualenv, le mieux est de le définir dans votre script `postactivate` :

```
#!/bin/sh
export PYTHONPATH="/home/me/django_projects/myproject:$VIRTUAL_ENV/lib/python3.4"
export DJANGO_SETTINGS_MODULE="myproject.settings.dev"
```

Si vous souhaitez utiliser un module de paramètres qui ne sont pas fait par `DJANGO_SETTINGS_MODULE` pour une fois, vous pouvez utiliser la `--settings` option `django-admin` :

```
django-admin test --settings=myproject.settings.tests
```

Alternative n ° 2

Si vous souhaitez laisser `DJANGO_SETTINGS_MODULE` à sa configuration par défaut (`myproject.settings`), vous pouvez simplement indiquer au module de `settings` quelle configuration charger en plaçant l'importation dans votre fichier `__init__.py` .

Dans l'exemple ci-dessus, le même résultat peut être obtenu en `__init__.py` un `__init__.py` sur:

```
from .dev import *
```

Utiliser plusieurs fichiers de besoins

Chaque fichier de configuration doit correspondre au nom d'un fichier de paramètres. Lire Utilisation de plusieurs paramètres pour plus d'informations.

Structure

```
djangoproject
├── config
│   ├── __init__.py
│   ├── requirements
│   │   ├── base.txt
│   │   ├── dev.txt
│   │   ├── test.txt
│   │   └── prod.txt
│   └── settings
└── manage.py
```

Dans le fichier `base.txt` , placez les dépendances utilisées dans tous les environnements.

```
# base.txt
Django==1.8.0
psycopg2==2.6.1
jinja2==2.8
```

Et dans tous les autres fichiers, incluez les dépendances de base avec `-r base.txt` , et ajoutez des dépendances spécifiques nécessaires pour l'environnement actuel.

```
# dev.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in dev env
django-queryinspect==0.1.0
```

```
# test.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in test env
nose==1.3.7
django-nose==1.4
```

```
# prod.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in production env
django-queryinspect==0.1.0
gunicorn==19.3.0
django-storages-redux==1.3
boto==2.38.0
```

Enfin, pour installer des dépendances. Exemple, sur dev env: `pip install -r config/requirements/dev.txt`

Masquage de données secrètes à l'aide d'un fichier JSON

Lorsque vous utilisez un VCS tel que Git ou SVN, il existe des données secrètes qui ne doivent jamais être versionnées (que le référentiel soit public ou privé).

Parmi ces données, vous trouvez le paramètre `SECRET_KEY` et le mot de passe de la base de données.

Une pratique courante pour masquer ces paramètres du contrôle de version consiste à créer un fichier `secrets.json` à la racine de votre projet (merci " *Two Scoops of Django* " pour l'idée):

```
{
    "SECRET_KEY": "N4HE:AMk:.Ader5354DR453TH8SHTQr",
    "DB_PASSWORD": "v3ry53cr3t"
}
```

Et ajoutez-le à votre liste des `.gitignore` (`.gitignore` pour git):

```
*.py[co]
*.sw[po]
*~
/secrets.json
```

Ajoutez ensuite la fonction suivante à votre module de `settings` :

```
import json
import os
from django.core.exceptions import ImproperlyConfigured

with open(os.path.join(BASE_DIR, 'secrets.json')) as secrets_file:
    secrets = json.load(secrets_file)

def get_secret(setting, secrets=secrets):
    """Get secret setting or fail with ImproperlyConfigured"""
    try:
        return secrets[setting]
    except KeyError:
        raise ImproperlyConfigured("Set the {} setting".format(setting))
```

Remplissez ensuite les paramètres de la manière suivante:

```
SECRET_KEY = get_secret('SECRET_KEY')
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgres',
        'NAME': 'db_name',
        'USER': 'username',
        'PASSWORD': get_secret('DB_PASSWORD'),
    },
}
```

Crédits: Deux Scoops de Django: Meilleures pratiques pour Django 1.8, par Daniel Roy Greenfeld et Audrey RoyGreenfeld. Copyright 2015 Two Scoops Press (ISBN 978-0981467344)

Utiliser un DATABASE_URL de l'environnement

Dans les sites PaaS tels que Heroku, il est courant de recevoir les informations de la base de données sous la forme d'une seule variable d'environnement URL, au lieu de plusieurs paramètres (hôte, port, utilisateur, mot de passe ...).

Il existe un module, `dj_database_url` qui extrait automatiquement la variable d'environnement DATABASE_URL dans un dictionnaire Python approprié pour injecter les paramètres de base de données dans Django.

Usage:

```
import dj_database_url

if os.environ.get('DATABASE_URL'):
    DATABASES['default'] =
        dj_database_url.config(default=os.environ['DATABASE_URL'])
```

Lire Paramètres en ligne: https://riptutorial.com/fr/django/topic/942/parametres

Chapitre 37: Processeurs de contexte

Remarques

Utilisez des processeurs de contexte pour ajouter des variables accessibles partout dans vos modèles.

Spécifiez une fonction ou des fonctions qui renvoient `dict` s des variables que vous voulez, puis ajouter ces fonctions à `TEMPLATE_CONTEXT_PROCESSORS` .

Examples

Utiliser un processeur de contexte pour accéder aux paramètres.DEBUG dans les modèles

dans `myapp/context_processors.py` :

```
from django.conf import settings

def debug(request):
  return {'DEBUG': settings.DEBUG}
```

dans `settings.py` :

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'myapp.context_processors.debug',
            ],
        },
    },
]
```

ou, pour les versions <1.9:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'myapp.context_processors.debug',
)
```

Ensuite, dans mes modèles, simplement:

```
 {% if DEBUG %} .header { background:#f00; } {% endif %}
 {{ DEBUG }}
```

Utiliser un processeur de contexte pour accéder à vos entrées de blog les plus récentes dans tous les modèles

En supposant que vous ayez un modèle appelé `Post` défini dans votre fichier `models.py` qui contient des billets de blog et un champ `date_published` .

Etape 1: Ecrivez le processeur de contexte

Créez (ou ajoutez) un fichier dans le répertoire de votre application appelé `context_processors.py` :

```
from myapp.models import Post

def recent_blog_posts(request):
    return {'recent_posts':Post.objects.order_by('-date_published')[0:3],}  # Can change
numbers for more/fewer posts
```

Étape 2: Ajouter le processeur de contexte à votre fichier de paramètres

Veillez à ajouter votre nouveau processeur de contexte à votre fichier `settings.py` dans la variable `TEMPLATES` :

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'myapp.context_processors.recent_blog_posts',
            ],
        },
    },
]
```

(Dans les versions de Django antérieures à la version 1.9, cela était défini directement dans `settings.py` utilisant une variable `TEMPLATE_CONTEXT_PROCESSORS` .)

Étape 3: Utilisez le processeur de contexte dans vos modèles

Plus besoin de passer des entrées de blog récentes via des vues individuelles! Utilisez simplement `recent_blog_posts` dans n'importe quel modèle.

Par exemple, dans `home.html` vous pouvez créer une barre latérale avec des liens vers des publications récentes:

```
<div class="blog_post_sidebar">
    {% for post in recent_blog_posts %}
        <div class="post">
            <a href="{{post.get_absolute_url}}">{{post.title}}</a>
        </div>
    {% endfor %}
```

```
</div>
```

Ou dans `blog.html` vous pouvez créer un affichage plus détaillé de chaque article:

```
<div class="content">
    {% for post in recent_blog_posts %}
        <div class="post_detail">
            <h2>{{post.title}}</h2>
            <p>Published on {{post.date_published}}</p>
            <p class="author">Written by: {{post.author}}</p>
            <p><a href="{{post.get_absolute_url}}">Permalink</a></p>
            <p class="post_body">{{post.body}}</p>
        </div>
    {% endfor %}
</div>
```

Extension de vos modèles

Processeur de contexte pour déterminer le modèle en fonction de l'appartenance à un groupe (ou de toute requête / logique). Cela permet à nos utilisateurs publics / réguliers d’obtenir un modèle et notre groupe spécial pour en obtenir un autre.

myapp / context_processors.py

```
def template_selection(request):
    site_template = 'template_public.html'
    if request.user.is_authenticated():
        if request.user.groups.filter(name="some_group_name").exists():
            site_template = 'template_new.html'

    return {
        'site_template': site_template,
    }
```

Ajoutez le processeur de contexte à vos paramètres.

Dans vos modèles, utilisez la variable définie dans le processeur de contexte.

```
{% extends site_template %}
```

Lire Processeurs de contexte en ligne: https://riptutorial.com/fr/django/topic/491/processeurs-de-contexte

Chapitre 38: Querysets

Introduction

Un `Queryset` est fondamentalement une liste d'objets dérivés d'un `Model` , par une compilation de requêtes de base de données.

Examples

Requêtes simples sur un modèle autonome

Voici un modèle simple que nous utiliserons pour exécuter quelques requêtes de test:

```
class MyModel(models.Model):
    name = models.CharField(max_length=10)
    model_num = models.IntegerField()
    flag = models.NullBooleanField(default=False)
```

Obtenez un objet de modèle unique où l'id / pk est 4:
(S'il n'y a pas d'éléments avec l'id de 4 ou s'il y en a plus d'un, cela lancera une exception.)

```
MyModel.objects.get(pk=4)
```

Tous les objets du modèle:

```
MyModel.objects.all()
```

Les objets de modèle dont l' `flag` défini sur `True` :

```
MyModel.objects.filter(flag=True)
```

Modéliser des objets avec un `model_num` supérieur à 25:

```
MyModel.objects.filter(model_num__gt=25)
```

Les objets de modèle portant le `name` "Élément bon marché" et le `flag` défini sur `False` :

```
MyModel.objects.filter(name="Cheap Item", flag=False)
```

Modélise un `name` recherche simple pour une chaîne spécifique (sensible à la casse):

```
MyModel.objects.filter(name__contains="ch")
```

Modèles Recherche simple `name` de chaîne spécifique (insensible à la casse):

```
MyModel.objects.filter(name__icontains="ch")
```

Requêtes avancées avec des objets Q

Vu le modèle:

```
class MyModel(models.Model):
    name = models.CharField(max_length=10)
    model_num = models.IntegerField()
    flag = models.NullBooleanField(default=False)
```

Nous pouvons utiliser des objets `Q` pour créer des conditions `AND` , `OR` dans votre requête de recherche. Par exemple, supposons que tous les objets ayant `flag=True` **OR** `model_num>15` .

```
from django.db.models import Q
MyModel.objects.filter(Q(flag=True) | Q(model_num__gt=15))
```

Ce qui précède se traduit par `WHERE flag=True OR model_num > 15` même pour un **ET que** vous feriez.

```
MyModel.objects.filter(Q(flag=True) & Q(model_num__gt=15))
```

`Q` objets `Q` nous permettent également de faire des requêtes **NOT** avec l'utilisation de `~` . Disons que nous voulions obtenir tous les objets qui ont `flag=False` **AND** `model_num!=15` , nous ferions:

```
MyModel.objects.filter(Q(flag=True) & ~Q(model_num=15))
```

Si vous utilisez des objets Q et des paramètres "normaux" dans `filter()` , les objets `Q` doivent être placés en *premier* . La requête suivante recherche les modèles avec (`flag` défini sur `True` ou un numéro de modèle supérieur à `15`) et un nom commençant par "H".

```
from django.db.models import Q
MyModel.objects.filter(Q(flag=True) | Q(model_num__gt=15), name__startswith="H")
```

Remarque: les objets `Q` peuvent être utilisés avec toute fonction de recherche prenant en compte les arguments de mots clés tels que `filter` , `exclude` , `get` . Assurez-vous que lorsque vous utilisez avec `get` , vous ne retournerez qu'un objet ou que l'exception `MultipleObjectsReturned` sera déclenchée.

Réduire le nombre de requêtes sur ManyToManyField (problème n + 1)

Problème

```
# models.py:
class Library(models.Model):
    name = models.CharField(max_length=100)
    books = models.ManyToManyField(Book)
```

```
class Book(models.Model):
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.all()

    # Query the database on each iteration (len(author) times)
    # if there is 100 librairies, there will have 100 queries plus the initial query
    for library in libraries:
        books = library.books.all()
        books[0].title
        # ...

    # total : 101 queries
```

Solution

Utilisez `prefetch_related` sur `ManyToManyField` si vous savez que vous devrez accéder ultérieurement à un champ qui est un champ `ManyToManyField` .

```
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books').all()

    # Does not query the database again, since `books` is pre-populated
    for library in libraries:
        books = library.books.all()
        books[0].title
        # ...

    # total : 2 queries - 1 for libraries, 1 for books
```

`prefetch_related` peut également être utilisé sur les champs de recherche:

```
# models.py:
class User(models.Model):
    name = models.CharField(max_length=100)

class Library(models.Model):
    name = models.CharField(max_length=100)
    books = models.ManyToManyField(Book)

class Book(models.Model):
    title = models.CharField(max_length=100)
    readers = models.ManyToManyField(User)
```

```
 # views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books', 'books__readers').all()
```

```
    # Does not query the database again, since `books` and `readers` is pre-populated
    for library in libraries:
        for book in library.books.all():
            for user in book.readers.all():
                user.name
                # ...

    # total : 3 queries - 1 for libraries, 1 for books, 1 for readers
```

Cependant, une fois que le jeu de requête a été exécuté, les données extraites ne peuvent pas être modifiées sans toucher à nouveau la base de données. Les éléments suivants exécuteraient des requêtes supplémentaires, par exemple:

```
 # views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books').all()
    for library in libraries:
        for book in library.books.filter(title__contains="Django"):
            print(book.name)
```

Les éléments suivants peuvent être optimisés en utilisant un objet `Prefetch` , introduit dans Django 1.7:

```
from django.db.models import Prefetch
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related(
        Prefetch('books', queryset=Book.objects.filter(title__contains="Django")
    ).all()
    for library in libraries:
        for book in library.books.all():
            print(book.name)  # Will print only books containing Django for each library
```

Réduire le nombre de requêtes sur le champ ForeignKey (problème n + 1)

Problème

Les groupes de requêtes Django sont évalués paresseusement. Par exemple:

```
# models.py:
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author, related_name='books')
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
```

```
    # Query the database
    books = Book.objects.all()

    for book in books:
        # Query the database on each iteration to get author (len(books) times)
        # if there is 100 books, there will have 100 queries plus the initial query
        book.author
        # ...

    # total : 101 queries
```

Le code ci-dessus fait que django interroge la base de données pour trouver l'auteur de chaque livre. Ceci est inefficace et il est préférable de ne disposer que d'une seule requête.

Solution

Utilisez `select_related` sur `ForeignKey` si vous savez que vous devrez ultérieurement accéder à un champ `ForeignKey` .

```
# views.py
def myview(request):
    # Query the database.
    books = Books.objects.select_related('author').all()

    for book in books:
        # Does not query the database again, since `author` is pre-populated
        book.author
        # ...

    # total : 1 query
```

`select_related` peut également être utilisé dans les champs de recherche:

```
# models.py:
class AuthorProfile(models.Model):
    city = models.CharField(max_length=100)

class Author(models.Model):
    name = models.CharField(max_length=100)
    profile = models.OneToOneField(AuthorProfile)

class Book(models.Model):
    author = models.ForeignKey(Author, related_name='books')
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    books = Book.objects.select_related('author')\
                        .select_related('author__profile').all()

    for book in books:
        # Does not query database
        book.author.name
        # or
```

```
        book.author.profile.city
        # ...

    # total : 1 query
```

Obtenir le jeu de requête SQL pour Django

L'attribut `query` sur queryset vous donne une syntaxe équivalente SQL pour votre requête.

```
>>> queryset = MyModel.objects.all()
>>> print(queryset.query)
SELECT "myapp_mymodel"."id", ... FROM "myapp_mymodel"
```

> **Attention:**
>
> Cette sortie ne doit être utilisée qu'à des fins de débogage. La requête générée n'est pas spécifique au backend. En tant que tels, les paramètres ne sont pas correctement cités, ce qui les rend vulnérables à l'injection SQL, et la requête peut même ne pas être exécutable sur votre backend de base de données.

Obtenir le premier et le dernier enregistrement de QuerySet

Pour obtenir le premier objet:

```
MyModel.objects.first()
```

Pour obtenir les derniers objets:

```
MyModel.objects.last()
```

Utilisation du filtre Premier objet:

```
MyModel.objects.filter(name='simple').first()
```

Utilisation du filtre Dernier objet:

```
MyModel.objects.filter(name='simple').last()
```

Requêtes avancées avec des objets F

> Un objet F () représente la valeur d'un champ de modèle ou d'une colonne annotée. Il permet de faire référence aux valeurs de champs de modèle et d'effectuer des opérations de base de données en les utilisant sans avoir à les extraire de la base de données en mémoire Python. - Expressions F ()

Il est approprié d'utiliser des objets `F()` chaque fois que vous devez référencer la valeur d'un autre champ dans votre requête. En soi, les objets `F()` ne veulent rien dire et ils ne peuvent et ne doivent pas être appelés en dehors d'un jeu de requête. Ils sont utilisés pour référencer la valeur

d'un champ sur le même jeu de requête.

Par exemple, donné un modèle ...

```
SomeModel(models.Model):
    ...
    some_field = models.IntegerField()
```

... un utilisateur peut interroger des objets dont la valeur `some_field` correspond à deux fois son `id` en référençant la valeur du champ `id` lors du filtrage à l'aide de `F()` comme ceci:

```
SomeModel.objects.filter(some_field=F('id') * 2)
```

`F('id')` référence simplement la valeur de l' `id` pour cette même instance. Django l'utilise pour créer l'instruction SQL correspondante. Dans ce cas, quelque chose qui ressemble beaucoup à ceci:

```
SELECT * FROM some_app_some_model
WHERE some_field = ((id * 2))
```

Sans les expressions `F()` , cela se ferait avec du SQL brut ou du filtrage en Python (ce qui réduit les performances, surtout quand il y a beaucoup d'objets).

Les références:

- Les filtres peuvent référencer des champs sur le modèle
- F expressions
- Réponse de TinyInstance

De la définition de la classe `F()` :

> Un objet capable de résoudre les références aux objets de requête existants. - source F

Remarque: Cet exemple publié provient de la réponse répertoriée ci-dessus avec le consentement de TinyInstance.

Lire Querysets en ligne: https://riptutorial.com/fr/django/topic/1235/querysets

Chapitre 39: RangeFields - un groupe de champs spécifiques à PostgreSQL

Syntaxe

- à partir de django.contrib.postgres.fields import * RangeField
- IntegerRangeField (** options)
- BigIntegerRangeField (options **)
- FloatRangeField (** options)
- DateTimeRangeField (options **)
- DateRangeField (** options)

Examples

Inclure des champs de plage numériques dans votre modèle

Il existe trois types de `RangeField` numériques dans Python. `IntegerField` , `BigIntegerField` et `FloatField` . Ils convertissent en `psycopg2` `NumericRange` s, mais acceptent les entrées en tant que tuples Python natifs. **La limite inférieure est incluse et la limite supérieure est exclue.**

```
class Book(models.Model):
    name = CharField(max_length=200)
    ratings_range = IntegerRange()
```

Mise en place pour RangeField

1. Ajouter `'django.contrib.postgres'` à votre `INSTALLED_APPS`
2. installez `psycopg2`

Création de modèles avec des champs de plage numériques

Il est plus simple et facile de saisir des valeurs sous la forme d'un tuple Python au lieu d'un `NumericRange` .

```
Book.objects.create(name='Pro Git', ratings_range=(5, 5))
```

Méthode alternative avec `NumericRange` :

```
Book.objects.create(name='Pro Git', ratings_range=NumericRange(5, 5))
```

Utiliser contient

Cette requête sélectionne tous les livres avec une note inférieure à trois.

```
bad_books = Books.objects.filter(ratings_range__contains=(1, 3))
```

Utiliser contain_by

Cette requête obtient tous les livres avec des évaluations supérieures ou égales à zéro et inférieures à six.

```
all_books = Book.objects.filter(ratings_range_contained_by=(0, 6))
```

Utiliser le chevauchement

Cette requête obtient tous les rendez-vous qui se chevauchent de six à dix.

```
Appointment.objects.filter(time_span__overlap=(6, 10))
```

Utiliser None pour signifier aucune limite supérieure

Cette requête sélectionne tous les livres dont l'évaluation est supérieure ou égale à quatre.

```
maybe_good_books = Books.objects.filter(ratings_range__contains=(4, None))
```

Opérations de gammes

```
from datetime import timedelta

from django.utils import timezone
from psycopg2.extras import DateTimeTZRange

# To create a "period" object we will use psycopg2's DateTimeTZRange
# which takes the two datetime bounds as arguments
period_start = timezone.now()
period_end = period_start + timedelta(days=1, hours=3)
period = DateTimeTZRange(start, end)

# Say Event.timeslot is a DateTimeRangeField

# Events which cover at least the whole selected period,
Event.objects.filter(timeslot__contains=period)

# Events which start and end within selected period,
Event.objects.filter(timeslot__contained_by=period)

# Events which, at least partially, take place during the selected period.
Event.objects.filter(timeslot__overlap=period)
```

Lire RangeFields - un groupe de champs spécifiques à PostgreSQL en ligne: https://riptutorial.com/fr/django/topic/2630/rangefields---un-groupe-de-champs-specifiques-a-postgresql

Chapitre 40: Référence du champ du modèle

Paramètres

Paramètre	Détails
nul	Si true, les valeurs vides peuvent être stockées comme `null` dans la base de données
blanc	Si cela est vrai, le champ ne sera pas requis dans les formulaires. Si les champs sont laissés en blanc, Django utilisera la valeur du champ par défaut.
les choix	Une itération d'itérables à 2 éléments à utiliser comme choix pour ce champ. Si défini, le champ est rendu sous la forme d'une liste déroulante dans l'administrateur. `[('m', 'Male'),('f','Female'),('z','Prefer Not to Disclose')]` . Pour regrouper les options, imbriquer simplement les valeurs: `[('Video Source',((1,'YouTube'),(2,'Facebook')),('Audio Source',((3, 'Soundcloud'),(4, 'Spotify'))]`
db_column	Par défaut, django utilise le nom du champ pour la colonne de la base de données. Utilisez ceci pour fournir un nom personnalisé
db_index	Si `True` , un index sera créé sur ce champ dans la base de données
db_tablespace	Le tablespace à utiliser pour l'index de ce champ. *Ce champ est utilisé uniquement si le moteur de base de données le prend en charge, sinon il est ignoré* .
défaut	La valeur par défaut pour ce champ. Peut être une valeur ou un objet appelable. Pour les valeurs par défaut mutables (une liste, un ensemble, un dictionnaire), vous **devez** utiliser un appelable. En raison de la compatibilité avec les migrations, vous ne pouvez pas utiliser lambdas.
modifiable	Si la valeur est `False` , le champ n'est pas affiché dans l'administrateur du modèle ni dans `ModelForm` . La valeur par défaut est `True` .
messages d'erreur	Utilisé pour personnaliser les messages d'erreur par défaut affichés pour ce champ. La valeur est un dictionnaire, les clés représentant l'erreur et la valeur étant le message. Les clés par défaut (pour les messages d'erreur) sont `null` , `blank` , `invalid` , `invalid_choice` , `unique` et `unique_for_date` ; des messages d'erreur supplémentaires peuvent être définis par des champs personnalisés.
Texte d'aide	Texte à afficher avec le champ, pour aider les utilisateurs. HTML est

Paramètre	Détails
	autorisé.
on_delete	Lorsqu'un objet référencé par une ForeignKey est supprimé, Django émule le comportement de la contrainte SQL spécifiée par l'argument on_delete. C'est le deuxième argument de position pour les champs `ForeignKey` et `OneToOneField` . D'autres champs n'ont pas cet argument.
clé primaire	Si `True` , ce champ sera la clé primaire. Django ajoute automatiquement une clé primaire; il n'est donc nécessaire que si vous souhaitez créer une clé primaire personnalisée. Vous ne pouvez avoir qu'une seule clé primaire par modèle.
unique	Si la valeur est `True` , des erreurs sont générées si des valeurs en double sont entrées pour ce champ. Ceci est une restriction au niveau de la base de données et pas simplement un bloc d'interface utilisateur.
unique_pour_date	Définissez la valeur sur un `DateField` ou un `DateTimeField` , et des erreurs seront `DateField` s'il existe des valeurs en double *pour la même date ou l'heure de la date* .
unique_pour_mois	Semblable à `unique_for_date` , sauf que les chèques sont limités pour le mois.
unique_pour_année	Similaire à `unique_for_date` , sauf que les chèques sont limités à l'année.
verbose_name	Un nom convivial pour le champ, utilisé par django à divers endroits (comme la création d'étiquettes dans les formulaires d'administration et de modèle).
validateurs	Une liste de validateurs pour ce champ.

Remarques

- Vous pouvez écrire vos propres champs si vous le trouvez nécessaire
- Vous pouvez remplacer les fonctions de la classe du modèle de base, le plus souvent la fonction `save()`

Examples

Champs numériques

Des exemples de champs numériques sont donnés:

AutoField

Un entier auto-incrémenté généralement utilisé pour les clés primaires.

```
from django.db import models

class MyModel(models.Model):
    pk = models.AutoField()
```

> Chaque modèle obtient un champ de clé primaire (appelé `id`) par défaut. Par conséquent, il n'est pas nécessaire de dupliquer un champ id dans le modèle pour les besoins d'une clé primaire.

BigIntegerField

Un nombre entier convenable de `-9223372036854775808` à `9223372036854775807` (`8 Bytes`).

```
from django.db import models

class MyModel(models.Model):
    number_of_seconds = models.BigIntegerField()
```

IntegerField

IntegerField permet de stocker des valeurs entières comprises entre -2147483648 et 2147483647 (`4 Bytes`).

```
from django.db import models

class Food(models.Model):
    name = models.CharField(max_length=255)
    calorie = models.IntegerField(default=0)
```

> `default` paramètre `default` n'est pas obligatoire. Mais il est utile de définir une valeur par défaut.

PositiveIntegerField

Comme un IntegerField, mais doit être positif ou nul (0). Le PositiveIntegerField est utilisé pour stocker des valeurs entières comprises entre 0 et 2147483647 (`4 Bytes`). Cela peut être utile sur le terrain, ce qui devrait être sémantiquement positif. Par exemple, si vous enregistrez des aliments avec ses calories, elles ne devraient pas être négatives. Ce champ empêchera les valeurs négatives via ses validations.

```
from django.db import models

class Food(models.Model):
    name = models.CharField(max_length=255)
    calorie = models.PositiveIntegerField(default=0)
```

`default` paramètre `default` n'est pas obligatoire. Mais il est utile de définir une valeur par défaut.

SmallIntegerField

SmallIntegerField permet de stocker des valeurs entières comprises entre -32768 et 32767 (`2 Bytes`). Ce champ est utile pour les valeurs qui ne sont pas extrêmes.

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=255)
    temperature = models.SmallIntegerField(null=True)
```

PositiveSmallIntegerField

SmallIntegerField permet de stocker des valeurs entières comprises entre 0 et 32767 (`2 Bytes`). Tout comme SmallIntegerField, ce champ est utile pour les valeurs qui ne vont pas aussi haut et devrait être sémantiquement positif. Par exemple, il peut stocker l'âge qui ne peut être négatif.

```
from django.db import models

class Staff(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    age = models.PositiveSmallIntegerField(null=True)
```

Outre que PositiveSmallIntegerField est utile pour les choix, il s’agit de la façon dont Django a implémenté Enum:

```
from django.db import models
from django.utils.translation import gettext as _

APPLICATION_NEW = 1
APPLICATION_RECEIVED = 2
APPLICATION_APPROVED = 3
APPLICATION_REJECTED = 4

APLICATION_CHOICES = (
    (APPLICATION_NEW, _('New')),
    (APPLICATION_RECEIVED, _('Received')),
    (APPLICATION_APPROVED, _('Approved')),
    (APPLICATION_REJECTED, _('Rejected')),
)

class JobApplication(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    status = models.PositiveSmallIntegerField(
        choices=APLICATION_CHOICES,
        default=APPLICATION_NEW
    )
    ...
```

La définition des choix en tant que variables de classe ou variables de module en fonction de la situation est un bon moyen de les utiliser. Si des choix sont passés au champ sans noms conviviaux, cela créera une confusion.

DecimalField

Un nombre décimal à précision fixe, représenté dans Python par une instance Decimal. Contrairement à IntegerField et ses dérivés, ce champ a 2 arguments obligatoires:

1. *DecimalField.max_digits* : nombre maximal de chiffres autorisé dans le nombre. Notez que ce nombre doit être supérieur ou égal à decimal_places.
2. *DecimalField.decimal_places* : nombre de décimales à stocker avec le nombre.

Si vous voulez stocker des nombres jusqu'à 99 avec 3 décimales, vous devez utiliser `max_digits=5` et `decimal_places=3` :

```
class Place(models.Model):
    name = models.CharField(max_length=255)
    atmospheric_pressure = models.DecimalField(max_digits=5, decimal_places=3)
```

BinaryField

Ceci est un champ spécialisé, utilisé pour stocker des données binaires. Il accepte uniquement les **octets** . Les données sont sérialisées en base64 lors du stockage.

Comme cela stocke des données binaires, ce champ ne peut pas être utilisé dans un filtre.

```
from django.db import models

class MyModel(models.Model):
    my_binary_data = models.BinaryField()
```

CharField

Le CharField est utilisé pour stocker des longueurs de texte définies. Dans l'exemple ci-dessous, jusqu'à 128 caractères de texte peuvent être stockés dans le champ. L'entrée d'une chaîne plus longue entraînera une erreur de validation.

```
from django.db import models

class MyModel(models.Model):
    name = models.CharField(max_length=128, blank=True)
```

DateTimeField

DateTimeField est utilisé pour stocker les valeurs de date et heure.

```
class MyModel(models.Model):
```

```
    start_time = models.DateFimeField(null=True, blank=True)
    created_on = models.DateTimeField(auto_now_add=True)
    updated_on = models.DateTimeField(auto_now=True)
```

Un `DateTimeField` a deux paramètres facultatifs:

- `auto_now_add` définit la valeur du champ sur la date / heure actuelle lors de la création de l'objet.
- `auto_now` définit la valeur du champ sur la date / heure actuelle chaque fois que le champ est enregistré.

Ces options et le paramètre `default` sont mutuellement exclusifs.

Clé étrangère

Le champ ForeignKey est utilisé pour créer une relation `many-to-one` entre les modèles. Pas comme la plupart des autres champs nécessite des arguments de position. L'exemple suivant illustre la relation entre la voiture et le propriétaire:

```
from django.db import models

class Person(models.Model):
    GENDER_FEMALE = 'F'
    GENDER_MALE = 'M'

    GENDER_CHOICES = (
        (GENDER_FEMALE, 'Female'),
        (GENDER_MALE, 'Male'),
    )

    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
    age = models.SmallIntegerField()


class Car(model.Model)
    owner = models.ForeignKey('Person')
    plate = models.CharField(max_length=15)
    brand = models.CharField(max_length=50)
    model = models.CharField(max_length=50)
    color = models.CharField(max_length=50)
```

Le premier argument du champ est la classe à laquelle le modèle est lié. Le deuxième argument positionnel est l'argument `on_delete` . Dans les versions actuelles, cet argument n'est pas requis, mais il sera requis dans Django 2.0. La fonctionnalité par défaut de l'argument est affichée comme suit:

```
class Car(model.Model)
    owner = models.ForeignKey('Person', on_delete=models.CASCADE)
    ...
```

Cela entraînera la suppression des objets Car du modèle lorsque son propriétaire a supprimé du modèle Personne. C'est la fonctionnalité par défaut.

```
class Car(model.Model)
    owner = models.ForeignKey('Person', on_delete=models.PROTECT)
    ...
```

Cela empêchera la suppression des objets Personne s'ils sont liés à au moins un objet Car. Tous les objets Car qui font référence à un objet Personne doivent être supprimés en premier. Et puis l'objet Personne peut être supprimé.

Lire Référence du champ du modèle en ligne:
https://riptutorial.com/fr/django/topic/3686/reference-du-champ-du-modele

Chapitre 41: Relations plusieurs-à-plusieurs

Examples

Avec un modèle traversant

```
class Skill(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Developer(models.Model):
    name = models.CharField(max_length=50)
    skills = models.ManyToManyField(Skill, through='DeveloperSkill')

class DeveloperSkill(models.Model):
    """Developer skills with respective ability and experience."""

    class Meta:
        order_with_respect_to = 'developer'
        """Sort skills per developer so that he can choose which
        skills to display on top for instance.
        """
        unique_together = [
            ('developer', 'skill'),
        ]
        """It's recommended that a together unique index be created on
        `(developer,skill)`. This is especially useful if your database is
        being access/modified from outside django. You will find that such an
        index is created by django when an explicit through model is not
        being used.
        """


    ABILITY_CHOICES = [
        (1, "Beginner"),
        (2, "Accustomed"),
        (3, "Intermediate"),
        (4, "Strong knowledge"),
        (5, "Expert"),
    ]

    developer = models.ForeignKey(Developer, models.CASCADE)
    skill = models.ForeignKey(Skill, models.CASCADE)
    """The many-to-many relation between both models is made by the
    above two foreign keys.

    Other fields (below) store information about the relation itself.
    """

    ability = models.PositiveSmallIntegerField(choices=ABILITY_CHOICES)
    experience = models.PositiveSmallIntegerField(help_text="Years of experience.")
```

Il est recommandé de créer un index unique `(developer,skill)` . Ceci est particulièrement utile si votre base de données est en cours d'accès / modifiée depuis l'extérieur de django. Vous constaterez qu'un tel index est créé par django lorsqu'un modèle explicite n'est pas utilisé.

Simple plusieurs à plusieurs relations.

```
class Person(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Club(models.Model):
    name = models.CharField(max_length=50)
    members = models.ManyToManyField(Person)
```

Nous définissons ici une relation dans laquelle un club a plusieurs `Person` et membres et une personne peut être membre de plusieurs `Club` différents.

Bien que nous ne définissions que deux modèles, django crée en réalité trois tables dans la base de données pour nous. Ce sont `myapp_person` , `myapp_club` et myapp_club_members. Django crée automatiquement un index unique sur les `myapp_club_members(club_id,person_id)` .

Utiliser les champs ManyToMany

Nous utilisons ce modèle du premier exemple:

```
class Person(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Club(models.Model):
    name = models.CharField(max_length=50)
    members = models.ManyToManyField(Person)
```

Ajouter Tom et Bill à la boîte de nuit:

```
tom = Person.objects.create(name="Tom", description="A nice guy")
bill = Person.objects.create(name="Bill", description="Good dancer")

nightclub = Club.objects.create(name="The Saturday Night Club")
nightclub.members.add(tom, bill)
```

Qui est dans le club?

```
for person in nightclub.members.all():
    print(person.name)
```

Te donnera

```
Tom
Bill
```

Lire Relations plusieurs-à-plusieurs en ligne: https://riptutorial.com/fr/django/topic/2379/relations-plusieurs-a-plusieurs

Chapitre 42: Routage d'URL

Examples

Comment Django gère une requête

Django gère une requête en acheminant le chemin d’URL entrant vers une fonction de vue. La fonction de visualisation est chargée de renvoyer une réponse au client effectuant la demande. Différentes URL sont généralement traitées par différentes fonctions d'affichage. Pour acheminer la requête vers une fonction de vue spécifique, Django examine votre configuration d'URL (ou URLconf en abrégé). Le modèle de projet par défaut définit l'URLconf dans `<myproject>/urls.py` .

Votre URLconf devrait être un module python qui définit un attribut nommé `urlpatterns` , qui est une liste d' `django.conf.urls.url()` de `django.conf.urls.url()` . Chaque instance `url()` doit au minimum définir une expression régulière (une regex) à faire correspondre à l'URL et à une cible, qui est soit une fonction de vue, soit un URLconf différent. Si un modèle d'URL cible une fonction de vue, il est judicieux de lui attribuer un nom afin de pouvoir facilement s'y référer ultérieurement.

Jetons un coup d'oeil à un exemple de base:

```
# In <myproject>/urls.py

from django.conf.urls import url

from myapp.views import home, about, blog_detail

urlpatterns = [
    url(r'^$', home, name='home'),
    url(r'^about/$', about, name='about'),
    url(r'^blog/(?P<id>\d+)/$', blog_detail, name='blog-detail'),
]
```

Cet URLconf définit trois modèles d'URL, tous ciblant une vue: `home` , `about` et `blog-detail` .

- `url(r'^$', home, name='home'),`

Le regex contient une ancre de début '^', immédiatement suivie d'une ancre de fin '$'. Ce modèle correspondra aux demandes pour lesquelles le chemin de l'URL est une chaîne vide et les achemine vers la vue d' `home` définie dans `myapp.views` .

- `url(r'^about/$', about, name='about'),`

Cette expression régulière contient une ancre de début, suivie de la chaîne littérale à `about/` et de l'ancre de fin. Cela correspondra à l'URL `/about/` et l'acheminera vers la vue à `about` . Puisque chaque URL non vide commence par un `/` , Django coupe commodément la première barre oblique pour vous.

- `url(r'^blog/(?P<id>\d+)/$', blog_detail, name='blog-detail'),`

Cette regex est un peu plus complexe. Il définit l’ancre de départ et la chaîne littérale `blog/` , comme le modèle précédent. La partie suivante, `(?P<id>\d+)` , s'appelle un groupe de capture. Un groupe de capture, comme son nom l'indique, capture une partie de la chaîne et Django transmet la chaîne capturée en tant qu'argument à la fonction de vue.

La syntaxe d'un groupe de capture est la suivante: `(?P<name>pattern)` . `name` définit le nom du groupe, qui est également le nom que Django utilise pour transmettre l'argument à la vue. Le modèle définit les caractères qui correspondent au groupe.

Dans ce cas, le nom est `id` , donc la fonction `blog_detail` doit accepter un paramètre nommé `id` . Le motif est `\d+` . `\d` signifie que le motif ne correspond qu'aux caractères numériques. `+` signifie que le motif doit correspondre à un ou plusieurs caractères.

Quelques modèles communs:

Modèle	Utilisé pour	Allumettes
`\d+`	id	Un ou plusieurs caractères numériques
`[\w-]+`	limace	Un ou plusieurs caractères alphanumériques, traits de soulignement ou tirets
`[0-9]{4}`	année (long)	Quatre chiffres, zéro à neuf
`[0-9]{2}`	année courte mois jour du mois	Deux chiffres, zéro à neuf
`[^/]+`	segment de trajectoire	Tout sauf une barre oblique

Le groupe de capture dans le modèle de `blog-detail` est suivi d'un littéral `/` et de l'ancre de fin.

Les URL valides incluent:

- `/blog/1/ # passes id='1'`
- `/blog/42/ # passes id='42'`

Les URL non valides sont par exemple:

- `/blog/a/ # 'a' does not match '\d'`
- `/blog// # no characters in the capturing group does not match '+'`

Django traite chaque modèle d’URL dans le même ordre qu’ils sont définis dans `urlpatterns` . Ceci est important si plusieurs modèles peuvent correspondre à la même URL. Par exemple:

```
urlpatterns = [
    url(r'blog/(?P<slug>[\w-]+)/$', blog_detail, name='blog-detail'),
    url(r'blog/overview/$', blog_overview, name='blog-overview'),
```

```
]
```

Dans le URLconf ci-dessus, le deuxième modèle n'est pas accessible. Le modèle correspond à l'URL `/blog/overview/` , mais au lieu d'appeler la vue `blog_overview` , l'URL correspondra d'abord au modèle de `blog-detail` du `blog-detail` et appellera la vue `blog_detail` avec un argument `slug='overview'` .

Pour vous assurer que l'URL `/blog/overview/` est routé vers la vue `blog_overview` , le modèle doit être placé au-dessus du modèle de `blog-detail` du `blog-detail` :

```
urlpatterns = [
    url(r'blog/overview/$', blog_overview, name='blog-overview'),
    url(r'blog/(?P<slug>[\w-]+)/$', blog_detail, name='blog-detail'),
]
```

Définir l'espace de noms URL pour une application réutilisable (Django 1.9+)

Configurez l'URLconf de votre application pour utiliser automatiquement un espace de noms URL en définissant l'attribut `app_name` :

```
# In <myapp>/urls.py
from django.conf.urls import url

from .views import overview

app_name = 'myapp'
urlpatterns = [
    url(r'^$', overview, name='overview'),
]
```

Cela définira l' espace de noms de l' application sur `'myapp'` lorsqu'il est inclus dans l'URLconf racine>. L'utilisateur de votre application réutilisable n'a pas besoin d'effectuer de configuration autre que l'inclusion de vos URL:

```
# In <myproject>/urls.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^myapp/', include('myapp.urls')),
]
```

Votre application réutilisable peut maintenant inverser les URL à l'aide de l'espace de noms de l'application:

```
>>> from django.urls import reverse
>>> reverse('myapp:overview')
'/myapp/overview/'
```

La URLconf racine peut toujours définir un espace de noms d'instance avec le paramètre `namespace` :

```
# In <myproject>/urls.py
urlpatterns = [
    url(r'^myapp/', include('myapp.urls', namespace='mynamespace')),
]
```

L'espace de noms de l'application et l'espace de noms de l'instance peuvent être utilisés pour inverser les URL:

```
>>> from django.urls import reverse
>>> reverse('myapp:overview')
'/myapp/overview/'
>>> reverse('mynamespace:overview')
'/myapp/overview/'
```

L'espace de noms de l'instance utilise par défaut l'espace de noms de l'application s'il n'est pas défini explicitement.

Lire Routage d'URL en ligne: https://riptutorial.com/fr/django/topic/3299/routage-d-url

Chapitre 43: Routeurs de base de données

Examples

Ajout d'un fichier de routage de base de données

Pour utiliser plusieurs bases de données dans Django, il suffit de spécifier chacune d'elles dans `settings.py` :

```
DATABASES = {
    'default': {
        'NAME': 'app_data',
        'ENGINE': 'django.db.backends.postgresql',
        'USER': 'django_db_user',
        'PASSWORD': os.environ['LOCAL_DB_PASSWORD']
    },
    'users': {
        'NAME': 'remote_data',
        'ENGINE': 'django.db.backends.mysql',
        'HOST': 'remote.host.db',
        'USER': 'remote_user',
        'PASSWORD': os.environ['REMOTE_DB_PASSWORD']
    }
}
```

Utilisez un fichier `dbrouters.py` pour indiquer quels modèles doivent fonctionner sur quelles bases de données pour chaque classe d'opération de base de données. Par exemple, pour les données distantes stockées dans `remote_data` , vous pouvez vouloir:

```
class DbRouter(object):
    """
    A router to control all database operations on models in the
    auth application.
    """
    def db_for_read(self, model, **hints):
        """
        Attempts to read remote models go to remote database.
        """
        if model._meta.app_label == 'remote':
            return 'remote_data'
        return 'app_data'

    def db_for_write(self, model, **hints):
        """
        Attempts to write remote models go to the remote database.
        """
        if model._meta.app_label == 'remote':
            return 'remote_data'
        return 'app_data'

    def allow_relation(self, obj1, obj2, **hints):
        """
        Do not allow relations involving the remote database
        """
```

```
        if obj1._meta.app_label == 'remote' or \
           obj2._meta.app_label == 'remote':
           return False
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        """
        Do not allow migrations on the remote database
        """
        if model._meta.app_label == 'remote':
            return False
        return True
```

Enfin, ajoutez votre `dbrouter.py` à `settings.py` :

```
DATABASE_ROUTERS = ['path.to.DbRouter', ]
```

Spécifier différentes bases de données dans le code

La `obj.save()` normale `obj.save()` utilisera la base de données par défaut ou, si un routeur de base de données est utilisé, elle utilisera la base de données spécifiée dans `db_for_write` . Vous pouvez le remplacer en utilisant:

```
obj.save(using='other_db')
obj.delete(using='other_db')
```

De même, pour lire:

```
MyModel.objects.using('other_db').all()
```

Lire Routeurs de base de données en ligne: https://riptutorial.com/fr/django/topic/3395/routeurs-de-base-de-donnees

Chapitre 44: Sécurité

Examples

Protection XSS (Cross Site Scripting)

Les attaques XSS consistent à injecter du code HTML (ou JS) dans une page. Voir Qu'est-ce qu'un script intersite pour plus d'informations.

Pour éviter cette attaque, par défaut, Django échappe aux chaînes passées par une variable de modèle.

Compte tenu du contexte suivant:

```
context = {
    'class_name': 'large" style="font-size:4000px',
    'paragraph': (
        "<script type=\"text/javascript\">alert('hello world!');</script>"),
}
```

```
<p class="{{ class_name }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large&quot; style=&quot;font-size: 4000px">&lt;script&gt;alert(&#39;hello
world!&#39;);&lt;/script&gt;</p>
```

Si vous avez des variables contenant du code HTML que vous avez confiance et que vous voulez réellement rendre, vous devez explicitement le déclarer sûr:

```
<p class="{{ class_name|safe }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large" style="font-size: 4000px">&lt;script&gt;alert(&#39;hello
world!&#39;);&lt;/script&gt;</p>
```

Si vous avez un bloc contenant plusieurs variables qui sont toutes sûres, vous pouvez désactiver localement l'échappement automatique:

```
{% autoescape off %}
<p class="{{ class_name }}">{{ paragraph }}</p>
{% endautoescape %}
<!-- Will be rendered as: -->
<p class="large" style="font-size: 4000px"><script>alert('hello world!');</script></p>
```

Vous pouvez également marquer une chaîne comme sûre en dehors du modèle:

```
from django.utils.safestring import mark_safe

context = {
    'class_name': 'large" style="font-size:4000px',
    'paragraph': mark_safe(
        "<script type=\"text/javascript\">alert('hello world!');</script>"),
```

```
}
```

```
<p class="{{ class_name }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large&quot; style=&quot;font-size: 4000px"><script>alert('hello
world!');</script></p>
```

Certains utilitaires Django tels que `format_html` déjà des chaînes marquées comme sûres:

```
from django.utils.html import format_html

context = {
    'var': format_html('<b>{}</b> {}', 'hello', '<i>world!</i>'),
}
```

```
<p>{{ var }}</p>
<!-- Will be rendered as -->
<p><b>hello</b> &lt;i&gt;world!&lt;/i&gt;</p>
```

Protection de clickjacking

> Clickjacking est une technique malveillante qui consiste à inciter un utilisateur Web à cliquer sur quelque chose de différent de ce que l'utilisateur perçoit en cliquant dessus. Apprendre encore plus

Pour activer la protection contre le `XFrameOptionsMiddleware` de `XFrameOptionsMiddleware` , ajoutez le `XFrameOptionsMiddleware` à vos classes de middleware. Cela devrait déjà être là si vous ne l'avez pas supprimé.

```
# settings.py
MIDDLEWARE_CLASSES = [
    ...
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ...
]
```

Ce middleware définit l'en-tête 'X-Frame-Options' sur toutes vos réponses, à moins d'une exemption explicite ou déjà définie (non remplacée si elle est déjà définie dans la réponse). Par défaut, il est défini sur "SAMEORIGIN". Pour changer cela, utilisez le paramètre `X_FRAME_OPTIONS` :

```
X_FRAME_OPTIONS = 'DENY'
```

Vous pouvez remplacer le comportement par défaut pour chaque vue.

```
from django.utils.decorators import method_decorator
from django.views.decorators.clickjacking import (
    xframe_options_exempt, xframe_options_deny, xframe_options_sameorigin,
)

xframe_options_exempt_m = method_decorator(xframe_options_exempt, name='dispatch')
```

```
@xframe_options_sameorigin
def my_view(request, *args, **kwargs):
    """Forces 'X-Frame-Options: SAMEORIGIN'."""
    return HttpResponse(...)

@method_decorator(xframe_options_deny, name='dispatch')
class MyView(View):
    """Forces 'X-Frame-Options: DENY'."""

@xframe_options_exempt_m
class MyView(View):
    """Does not set 'X-Frame-Options' header when passing through the
    XFrameOptionsMiddleware.
    """
```

Protection contre la falsification de sites inter-sites (CSRF)

> La falsification de requêtes intersites, également connue sous le nom d'attaque en un clic ou d'équitation de session et abrégée CSRF ou XSRF, est un type d'exploitation malveillante d'un site Web sur lequel des commandes non autorisées sont transmises par un utilisateur du site. Apprendre encore plus

Pour activer la protection CSRF, ajoutez le `CsrfViewMiddleware` à vos classes de middleware. Ce middleware est activé par défaut.

```
# settings.py
MIDDLEWARE_CLASSES = [
    ...
    'django.middleware.csrf.CsrfViewMiddleware',
    ...
]
```

Ce middleware définira un jeton dans un cookie sur la réponse sortante. Chaque fois qu'une requête entrante utilise une méthode non sécurisée (n'importe quelle méthode sauf `GET` , `HEAD` , `OPTIONS` et `TRACE`), le cookie doit correspondre à un jeton envoyé en tant que données de formulaire `csrfmiddlewaretoken` ou en `X-CsrfToken` tête `X-CsrfToken` . Cela garantit que le client à l'origine de la demande est également le propriétaire du cookie et, par extension, la session (authentifiée).

Si une demande est faite via `HTTPS` , la vérification stricte du référent est activée. Si l'en `HTTP_REFERER` tête `HTTP_REFERER` ne correspond pas à l'hôte de la demande en cours ou à un hôte dans `CSRF_TRUSTED_ORIGINS` (nouveau dans 1.9), la demande est refusée.

Les formulaires qui utilisent la méthode `POST` doivent inclure le jeton CSRF dans le modèle. La balise de modèle `{% csrf_token %}` affichera un champ masqué et garantira que le cookie est défini sur la réponse:

```
<form method='POST'>
{% csrf_token %}
...
</form>
```

Les vues individuelles qui ne sont pas vulnérables aux attaques CSRF peuvent être exemptées à l'aide du décorateur `@csrf_exempt` :

```
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def my_view(request, *args, **kwargs):
    """Allows unsafe methods without CSRF protection"""
    return HttpResponse(...)
```

Bien que cela ne soit pas recommandé, vous pouvez désactiver le `CsrfViewMiddleware` si beaucoup de vos vues ne sont pas vulnérables aux attaques CSRF. Dans ce cas, vous pouvez utiliser le décorateur `@csrf_protect` pour protéger les vues individuelles:

```
from django.views.decorators.csrf import csrf_protect

@csrf_protect
def my_view(request, *args, **kwargs):
    """This view is protected against CSRF attacks if the middleware is disabled"""
    return HttpResponse(...)
```

Lire Sécurité en ligne: https://riptutorial.com/fr/django/topic/2957/securite

Chapitre 45: Structure du projet

Examples

Repository> Project> Site / Conf

Pour un projet Django avec des `requirements` et des `deployment tools` sous contrôle de source. Cet exemple s'appuie sur les concepts des deux scoops de Django . Ils ont publié un modèle :

```
repository/
    docs/
    .gitignore
    project/
        apps/
            blog/
                migrations/
                static/ #( optional )
                    blog/
                        some.css
                templates/ #( optional )
                    blog/
                        some.html
                models.py
                tests.py
                admin.py
                apps.py #( django 1.9 and later )
                views.py
            accounts/
                #... ( same as blog )
            search/
                #... ( same as blog )
        conf/
            settings/
                local.py
                development.py
                production.py
            wsgi
            urls.py
        static/
        templates/
    deploy/
        fabfile.py
    requirements/
        base.txt
        local.txt
    README
    AUTHORS
    LICENSE
```

Ici, les `apps` et les dossiers de `conf` contiennent respectivement `user created applications` et le `core configuration folder` pour le projet.

`static` dossiers `static` et `templates` dans le répertoire du `project` contiennent respectivement des fichiers statiques et des fichiers de `html markup` qui sont utilisés globalement tout au long du projet.

De plus, tous les `blog` , `accounts` et `search` dossiers d'application peuvent également contenir (principalement) des dossiers `static` et des `templates` .

Noms de fichiers statiques et de fichiers dans les applications django

`static` dossier `static` et `templates` dans les applications peut également contenir un dossier portant le nom de l'application `ex. blog` Ceci est une convention utilisée pour empêcher la pollution des espaces de noms, donc nous référençons les fichiers tels que `/blog/base.html` plutôt que `/base.html` qui fournit plus de clarté sur le fichier que nous référençons et préserve l'espace de noms.

Exemple: le dossier de `templates` dans `search` applications de `blog` et de `search` contient un fichier nommé `base.html` , et lorsque vous référencez le fichier dans les `views` votre application devient confuse dans quel fichier elle doit être rendue.

```
(Project Structure)
.../project/
    apps/
        blog/
            templates/
                base.html
        search/
            templates/
                base.html

(blog/views.py)
def some_func(request):
    return render(request, "/base.html")

(search/views.py)
def some_func(request):
    return render(request, "/base.html")

## After creating a folder inside /blog/templates/(blog) ##

(Project Structure)
.../project/
    apps/
        blog/
            templates/
                blog/
                    base.html
        search/
            templates/
                search/
                    base.html

(blog/views.py)
def some_func(request):
    return render(request, "/blog/base.html")

(search/views.py)
def some_func(request):
    return render(request, "/search/base.html")
```

Lire Structure du projet en ligne: https://riptutorial.com/fr/django/topic/4299/structure-du-projet

Chapitre 46: Tâches Async (Céleri)

Remarques

Celery est une file d'attente de tâches qui peut exécuter des tâches en arrière-plan ou planifiées et s'intègre assez bien à Django. Le céleri nécessite quelque chose connu sous le nom de **courtier** de messages pour transmettre les messages de l'invocation aux travailleurs. Ce courtier de messages peut être redis, rabbitmq ou même Django ORM / db, bien que cette approche ne soit pas recommandée.

Avant de commencer avec l'exemple, vous devrez configurer le céleri. Pour configurer le céleri, créez un fichier `celery_config.py` dans l'application principale, parallèle au fichier `settings.py` .

```
from __future__ import absolute_import
import os
from celery import Celery
from django.conf import settings

# broker url
BROKER_URL = 'redis://localhost:6379/0'

# Indicate Celery to use the default Django settings module
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'config.settings')

app = Celery('config')
app.config_from_object('django.conf:settings')
# if you do not need to keep track of results, this can be turned off
app.conf.update(
    CELERY_RESULT_BACKEND=BROKER_URL,
)

# This line will tell Celery to autodiscover all your tasks.py that are in your app folders
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

Et dans le fichier `__init__.py` l'application principale, importez l'application Céleri. comme ça

```
# -*- coding: utf-8 -*-
# Not required for Python 3.
from __future__ import absolute_import

from .celery_config import app as celery_app  # noqa
```

Pour exécuter un travailleur céleri, utilisez cette commande au niveau où manage.py est.

```
# pros is your django project,
celery -A proj worker -l info
```

Examples

Exemple simple pour ajouter 2 nombres

Pour commencer:

1. Installer le céleri en `pip install celery`
2. configurer le céleri (aller à la section des remarques)

```
from __future__ import absolute_import, unicode_literals

from celery.decorators import task


@task
def add_number(x, y):
    return x + y
```

Vous pouvez l'exécuter de manière asynchrone en utilisant la méthode `.delay()` .

`add_number.delay(5, 10)` , où 5 et 10 sont les arguments de la fonction `add_number`

Pour vérifier si la fonction asynchrone a terminé l'opération, vous pouvez utiliser la fonction `.ready()` sur l'objet asynchrone renvoyé par la méthode `delay` .

Pour récupérer le résultat du calcul, vous pouvez utiliser l'attribut `.result` sur l'objet asynchrone.

Exemple

```
async_result_object = add_number.delay(5, 10)
if async_result_object.ready():
    print(async_result_object.result)
```

Lire Tâches Async (Céleri) en ligne: https://riptutorial.com/fr/django/topic/5481/taches-async--celeri-

Chapitre 47: Templating

Examples

Les variables

Les variables que vous avez fournies dans votre contexte de vue peuvent être accédées en utilisant la notation à double accolade:

Dans vos `views.py` :

```
class UserView(TemplateView):
  """ Supply the request user object to the template """

  template_name = "user.html"

  def get_context_data(self, **kwargs):
    context = super(UserView, self).get_context_data(**kwargs)
    context.update(user=self.request.user)
    return context
```

Dans `user.html` :

```
<h1>{{ user.username }}</h1>

<div class="email">{{ user.email }}</div>
```

La notation par points aura accès à:

- propriétés de l'objet, par exemple `user.username` sera `{{ user.username }}`
- les recherches dans les dictionnaires, par exemple `request.GET["search"]` sera `{{ request.GET.search }}`
- les méthodes sans arguments, par exemple `users.count()` seront `{{ user.count }}`

Les variables de modèle ne peuvent pas accéder aux méthodes qui prennent des arguments.

Les variables peuvent également être testées et mises en boucle:

```
{% if user.is_authenticated %}
  {% for item in menu %}
    <li><a href="{{ item.url }}">{{ item.name }}</a></li>
  {% endfor %}
{% else %}
  <li><a href="{% url 'login' %}">Login</a>
{% endif %}
```

On accède aux URL en utilisant le format `{% url 'name' %}` , où les noms correspondent aux noms de vos `urls.py`

`{% url 'login' %}` - Rendra probablement sous `/accounts/login/`

`{% url 'user_profile' user.id %}` - Les arguments pour les URL sont fournis dans l'ordre
`{% url next %}` - Les URL peuvent être des variables

Templating in Class Views Vues

Vous pouvez transmettre des données à un modèle dans une variable personnalisée.

Dans vos `views.py` :

```
from django.views.generic import TemplateView
from MyProject.myapp.models import Item

class ItemView(TemplateView):
    template_name = "item.html"

    def items(self):
        """ Get all Items """
        return Item.objects.all()

    def certain_items(self):
        """ Get certain Items """
        return Item.objects.filter(model_field="certain")

    def categories(self):
        """ Get categories related to this Item """
        return Item.objects.get(slug=self.kwargs['slug']).categories.all()
```

Une simple liste dans votre `item.html` :

```
{% for item in view.items %}
<ul>
    <li>{{ item }}</li>
</ul>
{% endfor %}
```

Vous pouvez également récupérer des propriétés supplémentaires des données.

En supposant que votre `Item` modèle a un champ de `name` :

```
{% for item in view.certain_items %}
<ul>
    <li>{{ item.name }}</li>
</ul>
{% endfor %}
```

Création de modèles dans les vues basées sur les fonctions

Vous pouvez utiliser un modèle dans une vue basée sur les fonctions comme suit:

```
from django.shortcuts import render

def view(request):
    return render(request, "template.html")
```

Si vous souhaitez utiliser des variables de modèle, vous pouvez le faire comme suit:

```
from django.shortcuts import render

def view(request):
    context = {"var1": True, "var2": "foo"}
    return render(request, "template.html", context=context)
```

Ensuite, dans `template.html` , vous pouvez vous référer à vos variables comme suit:

```
<html>
{% if var1 %}
    <h1>{{ var2 }}</h1>
{% endif %}
</html>
```

Filtres de modèle

Le système de gabarit de Django comporte des *balises* et des *filtres* intégrés, qui sont des fonctions à l'intérieur d'un modèle pour rendre le contenu de manière spécifique. Plusieurs filtres peuvent être spécifiés avec des tubes et les filtres peuvent avoir des arguments, comme dans la syntaxe des variables.

```
{{ "MAINROAD 3222"|lower }}    # mainroad 3222
{{ 10|add:15}}                 # 25
{{ "super"|add:"glue" }}       # superglue
{{ "A7"|add:"00" }}            # A700
{{ myDate | date:"D d M Y"}}   # Wed 20 Jul 2016
```

Une liste des **filtres intégrés** disponibles est disponible à l' adresse https://docs.djangoproject.com/en/dev/ref/templates/builtins/#ref-templates-builtins-filters .

Création de filtres personnalisés

Pour ajouter vos propres filtres de modèle, créez un dossier nommé `templatetags` dans le dossier de votre application. Ajoutez ensuite un `__init__.py` et le fichier contenant les filtres:

```
#/myapp/templatetags/filters.py
from django import template

register = template.Library()

@register.filter(name='tostring')
def to_string(value):
    return str(value)
```

Pour utiliser le filtre, vous devez le charger dans votre modèle:

```
#templates/mytemplate.html
{% load filters %}
{% if customer_id|tostring = customer %} Welcome back {% endif%}
```

Des trucs

Même si les filtres semblent simples au début, cela permet de faire des choses astucieuses:

```
{% for x in ""|ljust:"20" %}Hello World!{% endfor %}    # Hello World!Hello World!Hel...
{{ user.name.split|join:"_" }} ## replaces whitespace with '_'
```

Voir aussi les balises de modèle pour plus d'informations.

Empêcher les méthodes sensibles d'être appelées dans des modèles

Lorsqu'un objet est exposé au contexte du modèle, ses méthodes sans arguments sont disponibles. Ceci est utile lorsque ces fonctions sont des "getters". Mais cela peut être dangereux si ces méthodes modifient certaines données ou ont des effets secondaires. Même si vous faites probablement confiance à l'auteur du modèle, il peut ne pas être conscient des effets secondaires d'une fonction ou penser à appeler le mauvais attribut par erreur.

Vu le modèle suivant:

```
class Foobar(models.Model):
    points_credit = models.IntegerField()

    def credit_points(self, nb_points=1):
        """Credit points and return the new points credit value."""
        self.points_credit = F('points_credit') + nb_points
        self.save(update_fields=['points_credit'])
        return self.points_credit
```

Si vous écrivez ceci, par erreur, dans un modèle:

```
 You have {{ foobar.credit_points }} points!
```

Cela augmentera le nombre de points à chaque appel du modèle. Et vous ne le remarquerez peut-être même pas.

Pour éviter cela, vous devez définir l'attribut `alters_data` sur `True` pour les méthodes qui ont des effets secondaires. Cela rendra impossible de les appeler à partir d'un modèle.

```
def credit_points(self, nb_points=1):
    """Credit points and return the new points credit value."""
    self.points_credit = F('points_credit') + nb_points
    self.save(update_fields=['points_credit'])
    return self.points_credit
credit_points.alters_data = True
```

Utilisation de {% extend%}, {% include%} et {% blocks%}

résumé

- **{% extend%}** : cela déclare le modèle donné comme argument en tant que parent du modèle en cours. Utilisation: `{% extends 'parent_template.html' %}` .
- **{% block%} {% endblock%}** : Ceci est utilisé pour définir des sections dans vos modèles, de sorte que si un autre modèle étend celui-ci, il pourra remplacer le code HTML qui a été écrit à l'intérieur. Les blocs sont identifiés par leur nom. Utilisation: `{% block content %} <html_code> {% endblock %}` .
- **{% include%}** : cela insérera un modèle dans le modèle actuel. Sachez que le modèle inclus recevra le contexte de la requête et vous pourrez également lui attribuer des variables personnalisées. Utilisation de base: `{% include 'template_name.html' %}` , utilisation avec des variables: `{% include 'template_name.html' with variable='value' variable2=8 %}`

Guider

Supposons que vous construisiez votre code côté frontal avec des dispositions communes pour tout le code et que vous ne souhaitiez pas répéter le code pour chaque modèle. Django vous offre des balises construites pour cela.
Supposons que nous ayons un site Web de blog ayant 3 modèles qui partagent la même mise en page:

```
project_directory
    ..
    templates
      front-page.html
      blogs.html
      blog-detail.html
```

1) Définir le fichier `base.html` ,

```
<html>
  <head>
  </head>

  <body>
        {% block content %}
        {% endblock %}
   </body>
</html>
```

2) `blog.html` dans `blog.html` comme,

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

{% block content %}
    # write your blog related code here
{% endblock %}

# None of the code written here will be added to the template
```

Ici, nous avons étendu la disposition de base afin que sa mise en page HTML soit maintenant disponible dans le `blog.html` blog.html. Le concept de `{ % block %}` est un héritage de modèle qui vous permet de créer un modèle de base contenant tous les éléments site et définit les blocs que les modèles enfants peuvent remplacer.

3) Supposons maintenant que tous vos 3 modèles aient le même code HTML, ce qui définit certains articles populaires. Au lieu d'être écrits, les 3 fois créent un nouveau modèle `posts.html` .

blog.html

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

{% block content %}
    # write your blog related code here
    {% include 'posts.html' %} # includes posts.html in blog.html file without passing any
data
    <!-- or -->
    {% include 'posts.html' with posts=postdata %} # includes posts.html in blog.html file
with passing posts data which is context of view function returns.
{% endblock %}
```

Lire Templating en ligne: https://riptutorial.com/fr/django/topic/588/templating

Chapitre 48: Test d'unité

Examples

Testing - un exemple complet

Cela suppose que vous ayez lu la documentation sur le démarrage d'un nouveau projet Django. Supposons que l'application principale de votre projet s'appelle td (abréviation de test). Pour créer votre premier test, créez un fichier nommé test_view.py et copiez-y le contenu suivant.

```
from django.test import Client, TestCase

class ViewTest(TestCase):

    def test_hello(self):
        c = Client()
        resp = c.get('/hello/')
        self.assertEqual(resp.status_code, 200)
```

Vous pouvez exécuter ce test en

```
 ./manage.py test
```

et il échouera tout naturellement! Vous verrez une erreur similaire à la suivante.

```
Traceback (most recent call last):
  File "/home/me/workspace/td/tests_view.py", line 9, in test_hello
    self.assertEqual(resp.status_code, 200)
AssertionError: 200 != 404
```

Pourquoi ça arrive? Parce que nous n'avons pas défini de vue pour cela! Alors faisons-le. Créez un fichier appelé views.py et placez-y le code suivant

```
from django.http import HttpResponse
def hello(request):
    return HttpResponse('hello')
```

Ensuite, associez-le à / hello / en éditant les URL py comme suit:

```
from td import views

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/', views.hello),
    ....
]
```

Maintenant, relancez le test `./manage.py test` nouveau et alto !!

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

Tester les modèles Django efficacement

En supposant une classe

```
from django.db import models

class Author(models.Model):
   name = models.CharField(max_length=50)

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('view_author', args=[str(self.id)])


class Book(models.Model):
    author = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    private = models.BooleanField(default=false)
    publish_date = models.DateField()

    def get_absolute_url(self):
        return reverse('view_book', args=[str(self.id)])

    def __str__(self):
        return self.name
```

Exemples de tests

```
from django.test import TestCase
from .models import Book, Author

class BaseModelTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(BaseModelTestCase, cls).setUpClass()
        cls.author = Author(name='hawking')
        cls.author.save()
        cls.first_book = Book(author=cls.author, name="short_history_of_time")
        cls.first_book.save()
        cls.second_book = Book(author=cls.author, name="long_history_of_time")
        cls.second_book.save()


class AuthorModelTestCase(BaseModelTestCase):
    def test_created_properly(self):
         self.assertEqual(self.author.name, 'hawking')
         self.assertEqual(True, self.first_book in self.author.book_set.all())

    def test_absolute_url(self):
```

```
        self.assertEqual(self.author.get_absolute_url(), reverse('view_author',
args=[str(self.author.id)]))

class BookModelTestCase(BaseModelTestCase):

    def test_created_properly(self:
        ...
        self.assertEqual(1, len(Book.objects.filter(name__startswith='long'))

    def test_absolute_url(self):
        ...
```

Des points

- `created_properly` tests `created_properly` sont utilisés pour vérifier les propriétés d'état des modèles django. Ils aident à prendre des précautions lorsque nous avons modifié les valeurs par défaut, file_upload_paths, etc.
- `absolute_url` peut sembler trivial mais j'ai trouvé que cela m'a aidé à éviter certains bogues lors de la modification des chemins d'url
- De même, j'écris des cas de test pour toutes les méthodes implémentées dans un modèle (en utilisant `mock` objets `mock` , etc.)
- En définissant un `BaseModelTestCase` commun, nous pouvons définir les relations nécessaires entre les modèles pour garantir des tests corrects.

Enfin, en cas de doute, écrivez un test. Les changements de comportement triviaux se font en prêtant attention aux détails et les bouts de code oubliés ne finissent pas par causer des problèmes inutiles.

Tester le contrôle d'accès dans les vues Django

tl; dr : crée une classe de base qui définit deux objets utilisateur (disons `user` et `another_user`). Créez vos autres modèles et définissez trois instances `Client` .

- `self.client` : Représentant l' `user` connecté au navigateur
- `self.another_client` : Représentant `another_user` client de »
- `self.unlogged_client` : Représenter une personne non associée

Accédez maintenant à toutes vos URL publiques et privées à partir de ces trois objets client et dictez la réponse attendue. Ci-dessous, j'ai présenté la stratégie d'un objet `Book` qui peut être `private` (appartenant à quelques utilisateurs privilégiés) ou `public` (visible par tous).

```
from django.test import TestCase, RequestFactory, Client
from django.core.urlresolvers import reverse

class BaseViewTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(BaseViewTestCase, cls).setUpClass()
        cls.client = Client()
        cls.another_client = Client()
        cls.unlogged_client = Client()
```

```
        cls.user = User.objects.create_user(
                'dummy',password='dummy'
                )
        cls.user.save()
        cls.another_user = User.objects.create_user(
                'dummy2', password='dummy2'
                )
        cls.another_user.save()
        cls.first_book = Book.objects.create(
                name='first',
                private = true
        )
        cls.first_book.readers.add(cls.user)
        cls.first_book.save()
        cls.public_book = Template.objects.create(
                name='public',
                private=False
        )
        cls.public_book.save()


    def setUp(self):
        self.client.login(username=self.user.username, password=self.user.username)
        self.another_client.login(username=self.another_user.username,
password=self.another_user.username)


"""
   Only cls.user owns the first_book and thus only he should be able to see it.
   Others get 403(Forbidden) error
"""
class PrivateBookAccessTestCase(BaseViewTestCase):

    def setUp(self):
        super(PrivateBookAccessTestCase, self).setUp()
        self.url = reverse('view_book',kwargs={'book_id':str(self.first_book.id)})

    def test_user_sees_own_book(self):
        response = self.client.get(self.url)
        self.assertEqual(200, response.status_code)
        self.assertEqual(self.first_book.name,response.context['book'].name)
        self.assertTemplateUsed('myapp/book/view_template.html')

    def test_user_cant_see_others_books(self):
        response = self.another_client.get(self.url)
        self.assertEqual(403, response.status_code)

    def test_unlogged_user_cant_see_private_books(self):
        response = self.unlogged_client.get(self.url)
        self.assertEqual(403, response.status_code)

"""
    Since book is public all three clients should be able to see the book
"""
 class PublicBookAccessTestCase(BaseViewTestCase):

    def setUp(self):
        super(PublicBookAccessTestCase, self).setUp()
        self.url = reverse('view_book',kwargs={'book_id':str(self.public_book.id)})

    def test_user_sees_book(self):
```

```
        response = self.client.get(self.url)
        self.assertEqual(200, response.status_code)
        self.assertEqual(self.public_book.name,response.context['book'].name)
        self.assertTemplateUsed('myapp/book/view_template.html')

    def test_another_user_sees_public_books(self):
        response = self.another_client.get(self.url)
        self.assertEqual(200, response.status_code)

    def test_unlogged_user_sees_public_books(self):
        response = self.unlogged_client.get(self.url)
        self.assertEqual(200, response.status_code)
```

La base de données et les tests

Django utilise des paramètres de base de données spéciaux lors des tests afin que les tests puissent utiliser la base de données normalement mais par défaut, ils s'exécutent sur une base de données vide. Les modifications de base de données dans un test ne seront pas vues par un autre. Par exemple, les deux tests suivants réussiront:

```
from django.test import TestCase
from myapp.models import Thing

class MyTest(TestCase):

    def test_1(self):
        self.assertEqual(Thing.objects.count(), 0)
        Thing.objects.create()
        self.assertEqual(Thing.objects.count(), 1)

    def test_2(self):
        self.assertEqual(Thing.objects.count(), 0)
        Thing.objects.create(attr1="value")
        self.assertEqual(Thing.objects.count(), 1)
```

Agencements

Si vous souhaitez que des objets de base de données soient utilisés par plusieurs tests, créez-les dans la méthode `setUp` du `setUp` de test. De plus, si vous avez défini des appareils dans votre projet Django, ils peuvent être inclus comme suit:

```
class MyTest(TestCase):
    fixtures = ["fixture1.json", "fixture2.json"]
```

Par défaut, django recherche des appareils dans le répertoire des `fixtures` de chaque application. D'autres répertoires peuvent être définis à l'aide du paramètre `FIXTURE_DIRS` :

```
# myapp/settings.py
FIXTURE_DIRS = [
    os.path.join(BASE_DIR, 'path', 'to', 'directory'),
]
```

Supposons que vous ayez créé un modèle comme suit:

```
# models.py
from django.db import models


class Person(models.Model):
    """A person defined by his/her first- and lastname."""
    firstname = models.CharField(max_length=255)
    lastname = models.CharField(max_length=255)
```

Alors vos appareils .json pourraient ressembler à ça:

```
# fixture1.json
[
    { "model": "myapp.person",
        "pk": 1,
        "fields": {
            "firstname": "Peter",
            "lastname": "Griffin"
        }
    },
    { "model": "myapp.person",
        "pk": 2,
        "fields": {
            "firstname": "Louis",
            "lastname": "Griffin"
        }
    },
]
```

Réutiliser la base de données de test

Pour accélérer vos tests, vous pouvez demander à la commande-management de réutiliser la base de données de test (et d'empêcher sa création avant et sa suppression après chaque test). Cela peut être fait en utilisant le flag keepdb (ou sténographie `-k`) comme ceci:

```
# Reuse the test-database (since django version 1.8)
$ python manage.py test --keepdb
```

Limiter le nombre de tests exécutés

Il est possible de limiter les tests exécutés par le `manage.py test` en spécifiant les modules à découvrir par le runner de test:

```
# Run only tests for the app names "app1"
$ python manage.py test app1

# If you split the tests file into a module with several tests files for an app
$ python manage.py test app1.tests.test_models

# it's possible to dig down to individual test methods.
$ python manage.py test app1.tests.test_models.MyTestCase.test_something
```

Si vous voulez exécuter un tas de tests, vous pouvez passer un modèle de noms de fichiers. Par exemple, vous pouvez exécuter uniquement des tests impliquant vos modèles:

```
$ python manage.py test -p test_models*
Creating test database for alias 'default'...
.................................................
----------------------------------------------------------------------
Ran 115 tests in 3.869s

OK
```

Enfin, il est possible d'arrêter la suite de tests dès le premier échec, en utilisant `--failfast` . Cet argument permet d'obtenir rapidement l'erreur potentielle rencontrée dans la suite:

```
$ python manage.py test app1
...F..
----------------------------------------------------------------------
Ran 6 tests in 0.977s

FAILED (failures=1)


$ python manage.py test app1 --failfast
...F
======================================================================
[Traceback of the failing test]
----------------------------------------------------------------------
Ran 4 tests in 0.372s

FAILED (failures=1)
```

Lire Test d'unité en ligne: https://riptutorial.com/fr/django/topic/1232/test-d-unite

Chapitre 49: Transactions de base de données

Examples

Transactions atomiques

Problème

Par défaut, Django valide immédiatement les modifications apportées à la base de données. Lorsque des exceptions se produisent pendant une série de validations, cela peut laisser votre base de données dans un état indésirable:

```
def create_category(name, products):
    category = Category.objects.create(name=name)
    product_api.add_products_to_category(category, products)
    activate_category(category)
```

Dans le scénario suivant:

```
>>> create_category('clothing', ['shirt', 'trousers', 'tie'])
---------------------------------------------------------------------------
ValueError: Product 'trousers' already exists
```

Une exception se produit lors de la tentative d’ajout du produit pantalon à la catégorie des vêtements. À ce stade, la catégorie elle-même a déjà été ajoutée et le produit de la chemise a été ajouté.

La catégorie incomplète et le produit contenant doivent être supprimés manuellement avant de corriger le code et d'appeler à nouveau la méthode `create_category()` , sinon une catégorie de doublons serait créée.

Solution

Le module `django.db.transaction` vous permet de combiner plusieurs modifications de base de données en une transaction atomique :

> [une] série d'opérations de base de données telles que toutes se produisent ou rien ne se produit.

Appliqué au scénario ci-dessus, il peut être appliqué en tant que décorateur :

```
from django.db import transaction

@transaction.atomic
def create_category(name, products):
    category = Category.objects.create(name=name)
    product_api.add_products_to_category(category, products)
    activate_category(category)
```

Ou en utilisant un gestionnaire de contexte :

```
def create_category(name, products):
    with transaction.atomic():
        category = Category.objects.create(name=name)
        product_api.add_products_to_category(category, products)
        activate_category(category)
```

Maintenant, si une exception se produit à n'importe quelle étape de la transaction, aucune modification de base de données ne sera validée.

Lire Transactions de base de données en ligne: https://riptutorial.com/fr/django/topic/5555/transactions-de-base-de-donnees

Chapitre 50: Utiliser Redis avec Django - Caching Backend

Remarques

Utiliser django-redis-cache ou django-redis sont deux solutions efficaces pour stocker tous les éléments mis en cache. Bien qu'il soit certainement possible de configurer directement Redis en tant que `SESSION_ENGINE` , une stratégie efficace consiste à configurer la mise en cache (comme ci-dessus) et à déclarer votre cache par défaut en tant que `SESSION_ENGINE` . Bien que ce soit vraiment le sujet d'un autre article sur la documentation, sa pertinence mène à l'inclusion.

Ajoutez simplement les éléments suivants à `settings.py` :

```
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
```

Examples

Utiliser django-redis-cache

Une implémentation potentielle de Redis comme utilitaire de mise en cache backend est le package django-redis-cache .

Cet exemple suppose que vous avez déjà un serveur Redis en fonctionnement .

```
$ pip install django-redis-cache
```

Modifiez vos `settings.py` pour inclure un objet `CACHES` (voir la documentation de Django sur la mise en cache).

```
CACHES = {
    'default': {
        'BACKEND': 'redis_cache.RedisCache',
        'LOCATION': 'localhost:6379',
        'OPTIONS': {
            'DB': 0,
        }
    }
}
```

Utiliser django-redis

Une implémentation potentielle de Redis comme utilitaire de mise en cache backend est le package django-redis .

Cet exemple suppose que vous avez déjà un serveur Redis en fonctionnement .

```
$ pip install django-redis
```

Modifiez vos `settings.py` pour inclure un objet `CACHES` (voir la documentation de Django sur la mise en cache).

```
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        }
    }
}
```

Lire Utiliser Redis avec Django - Caching Backend en ligne:
https://riptutorial.com/fr/django/topic/4085/utiliser-redis-avec-django---caching-backend

Chapitre 51: Vues basées sur la classe

Remarques

Lorsque vous utilisez CBV, nous avons souvent besoin de savoir exactement quelles méthodes nous pouvons écraser pour chaque classe générique. Cette page de la documentation de django répertorie toutes les classes génériques avec toutes leurs méthodes aplaties et les attributs de classe que nous pouvons utiliser.

De plus, le site Web Classy Class Based View fournit les mêmes informations avec une interface interactive agréable.

Examples

Vues basées sur la classe

Les vues en classe vous permettent de vous concentrer sur ce qui rend vos vues spéciales.

Une page statique à propos de la page peut n'avoir rien de spécial, à l'exception du modèle utilisé. Utilisez un TemplateView ! Tout ce que vous avez à faire est de définir un nom de modèle. Travail terminé. Prochain.

views.py

```
from django.views.generic import TemplateView


class AboutView(TemplateView):
    template_name = "about.html"
```

urls.py

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url('^about/', views.AboutView.as_view(), name='about'),
]
```

Remarquez comment nous n'utilisons pas directement `AboutView` dans l'URL. C'est parce qu'un appelable est attendu et c'est exactement ce que `as_view()` .

Données de contexte

Parfois, votre modèle nécessite un peu plus d'informations. Par exemple, nous aimerions que l'utilisateur figure dans l'en-tête de la page, avec un lien vers son profil à côté du lien de déconnexion. Dans ces cas, utilisez la méthode `get_context_data` .

views.py

```
class BookView(DetailView):
    template_name = "book.html"

    def get_context_data(self, **kwargs)
        """ get_context_data let you fill the template context """
        context = super(BookView, self).get_context_data(**kwargs)
        # Get Related publishers
        context['publishers'] = self.object.publishers.filter(is_active=True)
        return context
```

Vous devez appeler la méthode get_context_data sur la super classe et elle renverra l'instance de contexte par défaut. Tout élément que vous ajoutez à ce dictionnaire sera disponible pour le modèle.

book.html

```
<h3>Active publishers</h3>
<ul>
  {% for publisher in publishers %}
    <li>{{ publisher.name }}</li>
  {% endfor %}
</ul>
```

Vues de liste et de détails

Les vues de modèle conviennent bien à la page statique et vous pouvez les utiliser pour tout avec `get_context_data` mais cela ne serait guère mieux que d'utiliser la fonction comme vue.

Entrez ListView et DetailView

app / models.py

```
from django.db import models

class Pokemon(models.Model):
    name = models.CharField(max_length=24)
    species = models.CharField(max_length=48)
    slug = models.CharField(max_length=48)
```

app / views.py

```
from django.views.generic import ListView, DetailView
from .models import Pokemon


class PokedexView(ListView):
    """ Provide a list of Pokemon objects """
    model = Pokemon
    paginate_by = 25

class PokemonView(DetailView):
    model = Pokemon
```

C'est tout ce dont vous avez besoin pour générer une vue répertoriant tous vos objets d'un modèle et les vues d'un élément singulier. La liste est même paginée. Vous pouvez fournir `template_name` si vous voulez quelque chose de spécifique. Par défaut, il est généré à partir du nom du modèle.

app / templates / app / pokemon_list.html

```
<!DOCTYPE html>
<title>Pokedex</title>
<ul>{% for pokemon in pokemon_list %}
    <li><a href="{% url "app:pokemon" pokemon.pk %}">{{ pokemon.name }}</a>
        &ndash; {{ pokemon.species }}
</ul>
```

Le contexte est rempli avec la liste des objets sous deux noms, `object_list` et un deuxième build à partir du nom du modèle, ici `pokemon_list` . Si vous avez paginé la liste, vous devez également vous occuper du lien suivant et précédent. L'objet Paginator peut vous aider, il est également disponible dans les données de contexte.

app / templates / app / pokemon_detail.html

```
<!DOCTYPE html>
<title>Pokemon {{ pokemon.name }}</title>
<h1>{{ pokemon.name }}</h1>
<h2>{{ pokemon.species }} </h2>
```

Comme précédemment, le contexte est rempli avec votre objet modèle sous l' `object` name et `pokemon` , le second étant dérivé du nom du modèle.

app / urls.py

```
from django.conf.urls import url
from . import views

app_name = 'app'
urlpatterns = [
    url(r'^pokemon/$', views.PokedexView.as_view(), name='pokedex'),
    url(r'^pokemon/(?P<pk>\d+)/$', views.PokemonView.as_view(), name='pokemon'),
]
```

Dans cet extrait de code, l'URL de la vue de détail est générée à l'aide de la clé primaire. Il est également possible d'utiliser un slug comme argument. Cela donne une URL plus belle, plus facile à retenir. Cependant, il nécessite la présence d'un champ nommé slug dans votre modèle.

```
url(r'^pokemon/(?P<slug>[A-Za-z0-9_-]+)/$', views.PokemonView.as_view(), name='pokemon'),
```

Si un champ appelé `slug` n'est pas présent, vous pouvez utiliser la `slug_field` mise en `DetailView` pour pointer vers un autre domaine.

Pour la pagination, utilisez une page get parameters ou placez une page directement dans l'URL.

Création de forme et d'objet

Ecrire une vue pour créer un objet peut être assez ennuyeux. Vous devez afficher un formulaire, vous devez le valider, vous devez enregistrer l'élément ou renvoyer le formulaire avec une erreur. Sauf si vous utilisez l'une des vues d'édition génériques .

app / views.py

```
from django.core.urlresolvers import reverse_lazy
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from .models import Pokemon


class PokemonCreate(CreateView):
    model = Pokemon
    fields = ['name', 'species']


class PokemonUpdate(UpdateView):
    model = Pokemon
    fields = ['name', 'species']


class PokemonDelete(DeleteView):
    model = Pokemon
    success_url = reverse_lazy('pokedex')
```

`CreateView` et `UpdateView` ont deux attributs requis, le `model` et les `fields` . Par défaut, les deux utilisent un nom de modèle basé sur le nom de modèle suffixé par «_form». Vous ne pouvez modifier que le suffixe avec l'attribut template_name_suffix. DeleteView affiche un message de confirmation avant de supprimer l'objet.

`UpdateView` et `DeleteView` doivent être `DeleteView` sur l'objet. Ils utilisent la même méthode que `DetailView` , en extrayant une variable de l'URL et en faisant correspondre les champs d'objet.

app / templates / app / pokemon_form.html (extrait)

```
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save" />
</form>
```

`form` contient le formulaire avec tous les champs nécessaires. Ici, il sera affiché avec un paragraphe pour chaque champ à cause de `as_p` .

app / templates / app / pokemon_confirm_delete.html (extrait)

```
<form action="" method="post">
    {% csrf_token %}
    <p>Are you sure you want to delete "{{ object }}"?</p>
    <input type="submit" value="Confirm" />
</form>
```

La balise `csrf_token` est requise en raison de la protection django contre la falsification de requêtes. L'action d'attribut est laissée vide car l'URL affichant le formulaire est la même que celle qui gère la suppression / sauvegarde.

Il reste deux problèmes avec le modèle, si vous utilisez la même chose que pour la liste et l'exemple de détail. Tout d'abord, créer et mettre à jour se plaindra d'une URL de redirection manquante. Cela peut être résolu en ajoutant un `get_absolute_url` au modèle pokemon. Le deuxième problème est la confirmation de suppression qui n’affiche aucune information significative. Pour résoudre ce problème, la solution la plus simple consiste à ajouter une représentation sous forme de chaîne.

app / models.py

```
from django.db import models
from django.urls import reverse
from django.utils.encoding import python_2_unicode_compatible


@python_2_unicode_compatible
class Pokemon(models.Model):
```

```
    name = models.CharField(max_length=24)
    species = models.CharField(max_length=48)

    def get_absolute_url(self):
        return reverse('app:pokemon', kwargs={'pk':self.pk})

    def __str__(self):
        return self.name
```

Le décorateur de classe s'assurera que tout fonctionne correctement sous Python 2.

Exemple minimal

views.py :

```
from django.http import HttpResponse
from django.views.generic import View

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

urls.py :

```
from django.conf.urls import url
from myapp.views import MyView

urlpatterns = [
    url(r'^about/$', MyView.as_view()),
]
```

En savoir plus sur la documentation de Django »

Vues basées sur la classe Django: exemple de CreateView

Avec les vues génériques basées sur les classes, il est très simple et facile de créer les vues CRUD à partir de nos modèles. Souvent, l'admin de Django intégré n'est pas suffisant ou pas préféré et nous devons déployer nos propres vues CRUD. Les CBV peuvent être très utiles dans de tels cas.

La classe `CreateView` besoin de 3 choses: un modèle, les champs à utiliser et l'URL de réussite.

Exemple:

```
from django.views.generic import CreateView
from .models import Campaign

class CampaignCreateView(CreateView):
    model = Campaign
    fields = ('title', 'description')

    success_url = "/campaigns/list"
```

```

```

Une fois la création réussie, l'utilisateur est redirigé vers `success_url` . Nous pouvons également définir une méthode `get_success_url` place et utiliser `reverse` ou `reverse_lazy` pour obtenir l'URL de succès.

Maintenant, nous devons créer un modèle pour cette vue. Le modèle doit être nommé au format `<app name>/<model name>_form.html` . Le nom du modèle doit être en lettres majuscules. Par exemple, si le nom de mon application est `dashboard` , pour la vue de création ci-dessus, je dois créer un modèle nommé `dashboard/campaign_form.html` .

Dans le modèle, une variable de `form` contiendrait le formulaire. Voici un exemple de code pour le modèle:

```
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save" />
</form>
```

Maintenant il est temps d'ajouter la vue à nos modèles d'URL.

```
url('^campaign/new/$', CampaignCreateView.as_view(), name='campaign_new'),
```

Si nous visitons l'URL, nous devrions voir un formulaire avec les champs que nous avons choisis. Lorsque nous soumettons, il essaiera de créer une nouvelle instance du modèle avec les données et de l'enregistrer. En cas de succès, l'utilisateur sera redirigé vers l'URL de succès. En cas d'erreur, le formulaire sera à nouveau affiché avec les messages d'erreur.

Une vue, plusieurs formulaires

Voici un exemple rapide d'utilisation de plusieurs formulaires dans une vue Django.

```
from django.contrib import messages
from django.views.generic import TemplateView

from .forms import AddPostForm, AddCommentForm
from .models import Comment

class AddCommentView(TemplateView):

    post_form_class = AddPostForm
    comment_form_class = AddCommentForm
    template_name = 'blog/post.html'

    def post(self, request):
        post_data = request.POST or None
        post_form = self.post_form_class(post_data, prefix='post')
        comment_form = self.comment_form_class(post_data, prefix='comment')

        context = self.get_context_data(post_form=post_form,
                                        comment_form=comment_form)
```

```
        if post_form.is_valid():
            self.form_save(post_form)
        if comment_form.is_valid():
            self.form_save(comment_form)

        return self.render_to_response(context)

    def form_save(self, form):
        obj = form.save()
        messages.success(self.request, "{} saved successfully".format(obj))
        return obj

    def get(self, request, *args, **kwargs):
        return self.post(request, *args, **kwargs)
```

Lire Vues basées sur la classe en ligne: https://riptutorial.com/fr/django/topic/1220/vues-basees-sur-la-classe

Chapitre 52: Vues génériques

Introduction

Les vues génériques sont des vues qui effectuent une action prédéfinie, telle que la création, la modification ou la suppression d'objets, ou simplement l'affichage d'un modèle.

Les vues génériques doivent être distinguées des vues fonctionnelles, qui sont toujours écrites à la main pour effectuer les tâches requises. En bref, on peut dire que les vues génériques doivent être configurées, alors que les vues fonctionnelles doivent être programmées.

Les vues génériques peuvent gagner beaucoup de temps, en particulier lorsque vous avez de nombreuses tâches normalisées à effectuer.

Remarques

Ces exemples montrent que les vues génériques simplifient généralement les tâches standardisées. Au lieu de tout programmer à partir de zéro, vous configurez ce que les autres personnes ont déjà programmé pour vous. Cela a du sens dans de nombreuses situations, car cela vous permet de vous concentrer davantage sur la conception de vos projets plutôt que sur les processus en arrière-plan.

Alors, devriez-vous *toujours les* utiliser? Non, ils n'ont de sens que si vos tâches sont assez standardisées (chargement, édition, suppression d'objets) et que vos tâches sont répétitives. Utiliser une seule vue générique spécifique une seule fois, puis remplacer toutes ses méthodes pour effectuer des tâches très spécifiques peut ne pas avoir de sens. Vous pouvez être mieux avec une vue fonctionnelle ici.

Cependant, si vous avez beaucoup de vues qui requièrent cette fonctionnalité ou si vos tâches correspondent exactement aux tâches définies pour une vue générique spécifique, alors les vues génériques sont exactement ce dont vous avez besoin pour vous simplifier la vie.

Examples

Exemple minimum: vues fonctionnelle vs vues génériques

Exemple de vue fonctionnelle pour créer un objet. En excluant les commentaires et les lignes vierges, nous avons besoin de 15 lignes de code:

```
# imports
from django.shortcuts import render_to_response
from django.http import HttpResponseRedirect

from .models import SampleObject
from .forms import SampleObjectForm
```

```
# view functioon
def create_object(request):

    # when request method is 'GET', show the template
    if request.method == GET:
        # perform actions, such as loading a model form
        form = SampleObjectForm()
        return render_to_response('template.html', locals())

    # if request method is 'POST', create the object and redirect
    if request.method == POST:
        form = SampleObjectForm(request.POST)

        # save object and redirect to success page if form is valid
        if form.is_valid:
            form.save()
            return HttpResponseRedirect('url_to_redirect_to')

        # load template with form and show errors
        else:
            return render_to_response('template.html', locals())
```

Exemple pour une "vue générique basée sur les classes" pour effectuer la même tâche. Nous avons seulement besoin de 7 lignes de code pour accomplir la même tâche:

```
from django.views.generic import CreateView

from .models import SampleObject
from .forms import SampleObjectForm

class CreateObject(CreateView):
    model = SampleObject
    form_class = SampleObjectForm
    success_url = 'url_to_redirect_to'
```

Personnalisation des vues génériques

L'exemple ci-dessus ne fonctionne que si vos tâches sont des tâches entièrement standard. Vous n’ajoutez pas de contexte supplémentaire ici, par exemple.

Faisons un exemple plus réaliste. Supposons que nous voulions ajouter un titre de page au modèle. Dans la vue fonctionnelle, cela fonctionnerait comme cela - avec une seule ligne supplémentaire:

```
def create_object(request):
    page_title = 'My Page Title'

    # ...

    return render_to_response('template.html', locals())
```

Ceci est plus difficile (ou: contre-intuitif) à réaliser avec des vues génériques. Comme ils sont basés sur des classes, vous devez remplacer une ou plusieurs méthodes de la classe pour obtenir le résultat souhaité. Dans notre exemple, nous devons remplacer la méthode

get_context_data de la classe comme *suit* :

```
class CreateObject(CreateView):
    model = SampleObject
    form_class = SampleObjectForm
    success_url = 'url_to_redirect_to'

    def get_context_data(self, **kwargs):

        # Call class's get_context_data method to retrieve context
        context = super().get_context_data(**kwargs)

        context['page_title'] = 'My page title'
        return context
```

Ici, nous avons besoin de coder quatre lignes supplémentaires au lieu d'une seule - du moins pour la *première* variable de contexte supplémentaire à ajouter.

Vues génériques avec des mixins

Le véritable pouvoir des vues génériques se manifeste lorsque vous les combinez avec des mixins. Un mixin est juste une autre classe définie par vous dont les méthodes peuvent être héritées par votre classe de vue.

Supposons que vous souhaitiez que chaque vue affiche la variable supplémentaire 'page_title' dans le modèle. Au lieu de remplacer la méthode get_context_data à chaque fois que vous définissez la vue, vous créez un mixin avec cette méthode et laissez vos vues hériter de ce mixin. Cela semble plus compliqué qu'il ne l'est en réalité:

```
# Your Mixin
class CustomMixin(object):

    def get_context_data(self, **kwargs):

        # Call class's get_context_data method to retrieve context
        context = super().get_context_data(**kwargs)

        context['page_title'] = 'My page title'
        return context

# Your view function now inherits from the Mixin
class CreateObject(CustomMixin, CreateView):
    model = SampleObject
    form_class = SampleObjectForm
    success_url = 'url_to_redirect_to'

# As all other view functions which need these methods
class EditObject(CustomMixin, EditView):
    model = SampleObject
    # ...
```

La beauté de ceci est que votre code devient beaucoup plus structuré que ce n'est le cas pour les vues fonctionnelles. Toute votre logique derrière des tâches spécifiques se trouve dans un seul endroit et un seul endroit. En outre, vous économiserez énormément de temps, en particulier

lorsque vous avez de nombreuses vues qui effectuent toujours les mêmes tâches, sauf avec des objets différents.

Lire Vues génériques en ligne: https://riptutorial.com/fr/django/topic/9452/vues-generiques

Chapitre 53: Widgets de formulaire

Examples

Widget de saisie de texte simple

L'exemple le plus simple de widget est la saisie de texte personnalisée. Par exemple, pour créer un `<input type="tel">` , vous devez sous- `input_type` `TextInput` et définir `input_type` sur `'tel'` .

```
from django.forms.widgets import TextInput

class PhoneInput(TextInput):
    input_type = 'tel'
```

Widget composite

Vous pouvez créer des widgets composés de plusieurs widgets à l'aide de `MultiWidget` .

```
from datetime import date

from django.forms.widgets import MultiWidget, Select
from django.utils.dates import MONTHS

class SelectMonthDateWidget(MultiWidget):
    """This widget allows the user to fill in a month and a year.

    This represents the first day of this month or, if `last_day=True`, the
    last day of this month.
    """

    default_nb_years = 10

    def __init__(self, attrs=None, years=None, months=None, last_day=False):
        self.last_day = last_day

        if not years:
            this_year = date.today().year
            years = range(this_year, this_year + self.default_nb_years)
        if not months:
            months = MONTHS

        # Here we will use two `Select` widgets, one for months and one for years
        widgets = (Select(attrs=attrs, choices=months.items()),
                   Select(attrs=attrs, choices=((y, y) for y in years)))
        super().__init__(widgets, attrs)

    def format_output(self, rendered_widgets):
        """Concatenates rendered sub-widgets as HTML"""
        return (
            '<div class="row">'
            '<div class="col-xs-6">{}</div>'
            '<div class="col-xs-6">{}</div>'
            '</div>'
        ).format(*rendered_widgets)
```

```
    def decompress(self, value):
        """Split the widget value into subwidgets values.
        We expect value to be a valid date formated as `%Y-%m-%d`.
        We extract month and year parts from this string.
        """
        if value:
            value = date(*map(int, value.split('-')))
            return [value.month, value.year]
        return [None, None]

    def value_from_datadict(self, data, files, name):
        """Get the value according to provided `data` (often from `request.POST`)
        and `files` (often from `request.FILES`, not used here)
        `name` is the name of the form field.

        As this is a composite widget, we will grab multiple keys from `data`.
        Namely: `field_name_0` (the month) and `field_name_1` (the year).
        """
        datalist = [
            widget.value_from_datadict(data, files, '{}_{}'.format(name, i))
            for i, widget in enumerate(self.widgets)]
        try:
            # Try to convert it as the first day of a month.
            d = date(day=1, month=int(datelist[0]), year=int(datelist[1]))
            if self.last_day:
                # Transform it to the last day of the month if needed
                if d.month == 12:
                    d = d.replace(day=31)
                else:
                    d = d.replace(month=d.month+1) - timedelta(days=1)
        except (ValueError, TypeError):
            # If we failed to recognize a valid date
            return ''
        else:
            # Convert it back to a string with format `%Y-%m-%d`
            return str(d)
```

Lire Widgets de formulaire en ligne: https://riptutorial.com/fr/django/topic/1230/widgets-de-formulaire

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Django	A. Raza, Abhishek Jain, Aidas Bendoraitis, Alexander Tyapkov, Ankur Gupta, Anthony Pham, Antoine Pinsard, arifin4web, Community, e4c5, elbear, ericdwang, ettanany, Franck Dernoncourt, greatwolf, ilse2005, Ivan Semochkin, J F, Jared Hooper, John, John Moutafis, JRodDynamite, Kid Binary, knbk, Louis, Luis Alberto Santana, lxer, maciek, McAbra, MiniGunnR, mnoronha, Nathan Osman, naveen.panwar, nhydock, Nikita Davidenko, noɥʇʎdʎzɐɹƆ, Rahul Gupta, rajarshig, Ron, ruddra, sarvajeetsuman, shacker, ssice, Stryker, techydesigner, The Brewmaster, Thereissoupinmyfly, Tom, WesleyJohnson, Zags
2	Administration	Antoine Pinsard, coffee-grinder, George H., Ivan Semochkin, noɥʇʎdʎzɐɹƆ, ssice
3	Agrégations de modèles	Ian Clark, John Moutafis, ravigadila
4	ArrayField - un champ spécifique à PostgreSQL	Antoine Pinsard, e4c5, noɥʇʎdʎzɐɹƆ
5	Backends d'authentification	knbk, Rahul Gupta
6	Balises de modèle et filtres	Antoine Pinsard, irakli khitarishvili, knbk, Medorator, naveen.panwar, The_Cthulhu_Kid
7	Céleri en cours d'exécution avec superviseur	RéÑjïth, sebb
8	Commandes de gestion	Antoine Pinsard, aquasan, Brian Artschwager, HorsePunchKid, Ivan Semochkin, John Moutafis, knbk, lxer, MarZab, Nikolay Fominyh, pbaranay, ptim, Rana Ahmed, techydesigner, Zags
9	Comment réinitialiser les migrations django	Cristus Cleetus
10	Comment utiliser Django avec Cookiecutter?	Atul Mishra, noɥʇʎdʎzɐɹƆ, OliPro007, RamenChef

11	Configuration de la base de données	Ahmad Anwar, Antoine Pinsard, Evans Murithi, Kid Binary, knbk , lxer, Majid, Peter Mortensen
12	CRUD à Django	aisflat439, George H.
13	Déploiement	Antoine Pinsard, Arpit Solanki, CodeFanatic23, I Am Batman, Ivan Semochkin, knbk, lxer, Maxime S., MaxLunar, Meska, no ɥʇdʎzɐɹϽ, rajarshig, Rishabh Agrahari, Roald Nefs, Rohini Choudhary, sebb
14	Des modèles	Aidas Bendoraitis, Alireza Aghamohammadi, alonisser, Antoine Pinsard, aquasan, Arpit Solanki, atomh33ls, coffee-grinder, DataSwede, ettanany, Gahan, George H., gkr, Ivan Semochkin, Jamie Cockburn, Joey Wilhelm, kcrk, knbk, Linville, lxer, maazza, Matt Seymour, MuYi, Navid777, nhydock, noɥʇdʎzɐɹϽ , pbaranay, PhoebeB, Rana Ahmed, Saksow, Sanyam Khurana, scriptmonster, Selcuk, SpiXel, sudshekhar, techydesigner, The_Cthulhu_Kid, Utsav T, waterproof, zurfyx
15	Des vues	ettanany, HorsePunchKid, John Moutafis
16	Django à partir de la ligne de commande.	e4c5, OliPro007
17	Django et les réseaux sociaux	Aidas Bendoraitis, aisflat439, Carlos Rojas, Ivan Semochkin, Rexford, Simplans
18	Django Rest Framework	The Brewmaster
19	django-filter	4444, Ahmed Atalla
20	Enregistrement	Antwane, Brian Artschwager, RamenChef
21	Expressions F ()	Antoine Pinsard, John Moutafis, Linville, Omar Shehata, RamenChef, Roald Nefs
22	Extension ou substitution de modèle d'utilisateur	Antoine Pinsard, Jon Clements, mnoronha, Raito, Rexford, rigdonmr, Rishabh Agrahari, Roald Nefs, techydesigner, The_Cthulhu_Kid
23	Formes	Aidas Bendoraitis, Antoine Pinsard, Daniel Rucci, ettanany, George H., knbk, NBajanca, nicorellius, RamenChef, rumman0786, sudshekhar, trpt4him
24	Formsets	naveen.panwar
25	Fuseaux horaires	William Reed
26	Gestionnaires et	abidibo, knbk, sudshekhar, Trivial

	ensembles de requêtes personnalisés	
27	Intégration continue avec Jenkins	pnovotnak
28	Internationalisation	Antoine Pinsard, dmvrtx
29	JSONField - un champ spécifique à PostgreSQL	Antoine Pinsard, Daniil Ryzhkov, Matthew Schinckel, uoɥʇʎԀʎzɐɹƆ, Omar Shehata, techydesigner
30	Le débogage	Antoine Pinsard, Ashutosh, e4c5, Kid Binary, knbk, Sayse, Udi
31	Les signaux	Antoine Pinsard, e4c5, Hetdev, John Moutafis, Majid, nhydock, Rexford
32	Mapper des chaînes à des chaînes avec HStoreField - un champ spécifique à PostgreSQL	uoɥʇʎԀʎzɐɹƆ
33	Meta: Guide de documentation	Antoine Pinsard
34	Middleware	AlvaroAV, Antoine Pinsard, George H., knbk, lxer, nhydock, Omar Shehata, Peter Mortensen, Trivial, William Reed
35	Migrations	Antoine Pinsard, engineercoding, Joey Wilhelm, knbk, MicroPyramid, ravigadila, Roald Nefs
36	Paramètres	allo, Antoine Pinsard, Brian Artschwager, fredley, J F, knbk, Louis, Louis Barranqueiro, lxer, Maxime Lorant, NBajanca, Nils Werner, ProfSmiles, RamenChef, Sanyam Khurana, Sayse, Selcuk, SpiXel, ssice, sudshekhar, Tema, The Brewmaster
37	Processeurs de contexte	Antoine Pinsard, Brian Artschwager, Dan Russell, Daniil Ryzhkov, fredley
38	Querysets	Antoine Pinsard, Brian Artschwager, Chalist, coffee-grinder, DataSwede, e4c5, Evans Murithi, George H., John Moutafis, Justin, knbk, Louis Barranqueiro, Maxime Lorant, MicroPyramid, nima, ravigadila, Sanyam Khurana, The Brewmaster
39	RangeFields - un groupe de champs spécifiques à PostgreSQL	Antoine Pinsard, uoɥʇʎԀʎzɐɹƆ

40	Référence du champ du modèle	Burhan Khalid, Husain Basrawala, knbk, Matt Seymour, Rod Xavier, scriptmonster, techydesigner, The_Cthulhu_Kid
41	Relations plusieurs-à-plusieurs	Antoine Pinsard, e4c5, knbk, Kostronor
42	Routage d'URL	knbk
43	Routeurs de base de données	fredley, knbk
44	Sécurité	Antoine Pinsard, knbk
45	Structure du projet	Antoine Pinsard, naveen.panwar, nicorellius
46	Tâches Async (Céleri)	iankit, Mevin Babu
47	Templating	Adam Starrh, Alasdair, Aniket, Antoine Pinsard, Brian H., coffee-grinder, doctorsherlock, fredley, George H., gkr, lxer, Stephen Leppik, Zags
48	Test d'unité	Adrian17, Antoine Pinsard, e4c5, Kim, Matthew Schinckel, Maxime Lorant, Patrik Stenmark, SandroM, sudshekhar, Zags
49	Transactions de base de données	Ian Clark
50	Utiliser Redis avec Django - Caching Backend	Majid, The Brewmaster
51	Vues basées sur la classe	Antoine Pinsard, Antwane, coffee-grinder, e4c5, gkr, knbk, maciek, masnun, Maxime Lorant, nicorellius, pleasedontbelong, Pureferret
52	Vues génériques	nikolas-berlin
53	Widgets de formulaire	Antoine Pinsard, ettanany