Django Story Documentation

Release 1.0

Rémy HUBSCHER

Apr 11, 2017

Contents

CHAPTER 1

Introduction

Introduction de Django Story

date 2012-04-30 10:07

tags django, python

category Django

author Rémy Hubscher

Présentation

Je suis le premier membre inscrit sur le Siteduzero.com après l'ouverture de la v3.

Cela fait déjà quelques années que je suis membre de cette communauté. C'est donc tout naturellement que j'ai commencé, en novembre 2011, un tutoriel pour parler de Django.

Cependant, il est assez difficile pour moi de rédiger les tutos dans le navigateur faire la mise en forme avec le zcode et tout.

J'ai donc décidé de le reprendre ici en écrivant avec du RST présenté dans un pelican comme une histoire racontée à un ami sans détails inutiles.

Si vous souhaitez une explication, il vous suffit de la demander dans les commentaires.

Il est sous Licence Art Libre.

Vous pouvez trouver la source ici : http://bitbucket.org/natim/django-story

Vous pouvez le lire en mode blog ici : http://django-story.ionyse.com/

Vous pouvez le lire en mode documentation ici : http://django-story.rtfd.org/

Django, le framework Web-Python

date 2012-04-30 10:33

tags django, python

category Django

author Rémy Hubscher

Qu'est-ce que Django ?

Django est un framework développé en Python, initialement pour un journal local dans le Kansas : World Online.

Un peu d'histoire

En 2003, deux développeurs (Adrian Holovaty et Simon Willison) ont décidé d'abandonner le langage PHP pour se mettre au Python afin de développer leur site dédié aux faits actuels. Pour améliorer le temps de développement, ils ont décidé de mettre en place un framework, une structure simple permettant de réduire considérablement le temps de développement d'un site. En deux ans, ce moteur a beaucoup changé, et change encore aujourd'hui, avec des ajouts et corrections.

C'est en 2005 que World Online décide d'ouvrir les portes de son framework : Django. Depuis ce framework a rencontré un franc succès. Il est utilisé par de nombreux développeurs ayant besoin de développer des sites de grande qualité, très rapidement.

Pourquoi "Django" ?

Le nom « Django » a été donné en souvenir de Django Reinhardt, guitariste jazz ayant vécu de 1910 à 1953. Il est considéré comme le meilleur guitariste manouche de tous les temps. Cet homme a beaucoup changé notre vision du jazz, et nous ne pouvons que l'en remercier. :)

Pour plus d'informations sur Django et ses origines, je vous redirige sur la FAQ officielle du projet [en].

Pourquoi utiliser Django ?

Pourquoi utiliser un Framework ?

Lorsque l'on réalise un site Internet, on en revient toujours aux même étapes :

1. réalisation et codage du design ;
2. réalisation des modules :
   (a) réalisation du modèle de données concernant le module,
   (b) réalisation des formulaires d'ajout, modification et suppression des données :
      i. vérification des données des formulaires,
      ii. affichage des erreurs,
      iii. réalisation et affichage des formulaires,
   (c) réalisation des pages d'affichage du contenu du site ;
3. réalisation d'une administration pour gérer les modules ;

4. réalisation d'un espace utilisateur avec des droits sur l'accès aux données ;
5. mise en place de flux RSS/ATOM ;
6. mise en place d'un plan du site ;
7. ...

Tout cela est relativement répétitif, et si, la première fois, ça peut paraître très amusant, on en arrive rapidement à faire des copier/coller, assez mauvaise méthode car source de nombreuses erreurs. Finalement on regroupe des morceaux de code en fonctions réutilisables.

À ce moment, on se rapproche de plus en plus de la notion de framework ci-dessus. L'avantage d'utiliser un framework existant et surtout Open Source tel que Django, c'est que nous ne sommes pas les seuls à l'utiliser, et que les bugs sont donc corrigés plus rapidement, les améliorations sont exécutées par plusieurs personnes et de manière bien mieux réfléchie.

C'est d'ailleurs tout l'intérêt d'utiliser un framework. En faire moins, pour en faire plus dans le même temps.

Pourquoi Django ?

Il existe de nombreux framework web, dans différents langages de programmation. Pourquoi utiliser spécifiquement Django et pas un autre ?

Voici une question à laquelle chacun a sa réponse ; d'ailleurs, tout le monde n'utilise pas Django. Vous êtes complètement libre de votre choix. Nous sommes nombreux à avoir choisi Django pour plusieurs raisons.

- La simplicité d'apprentissage.
- La qualité des applications réalisées.
- La rapidité de développement.
- La sécurité du site Internet final.
- La facilité de maintenance des applications sur la durée.

On bénéficie de la clarté de Python, qui permet à plusieurs développeurs de travailler sur le même projet. Le style est imposé, donc tout le monde suit les mêmes règles, ce qui facilite les travaux en équipe et la clarté du code.

En comparaison avec le PHP, on se rend compte qu'il existe de nombreuses manières de faire. On peut placer des morceaux de codes PHP au milieu de pages HTML (une solution assez mal organisée), ou encore utiliser un moteur de templates pour séparer le code du HTML. En Python/Django, tout est mis en place pour ne pouvoir faire que ce qui est bien, et ce dès le début de l'apprentissage.

Mais encore ?

Voyons concrètement ce que Django apporte et profitons-en pour définir quelques termes. Pour commencer, reprenons notre code CGI (listing 1.1) :

```
#!/usr/bin/python

import MySQLdb

print "Content-Type: text/html"
print
print "<html><head><title>Livres</title></head>"
print "<body>"
print "<h1>Livres</h1>"
print "<ul>"
```

```
connection = MySQLdb.connect(user='moi', passwd='laissezmoientrer', db='ma_base')
cursor = connection.cursor()
cursor.execute("SELECT nom FROM livres ORDER BY pub_date DESC LIMIT 10")
for row in cursor.fetchall():
    print "<li>%s</li>" % row[0]

print "</ul>"
print "</body></html>"

connection.close()
```

On définit, dans un premier temps, le type de fichier généré, puis on affiche du code HTML, on récupère ensuite des informations sur des livres contenus dans une base de données, on ré-affiche du HTML, et on ferme notre connexion à la base de données.

Pour une page simple comme celle-ci, cette approche aisée peut convenir, mais lorsque l'application grossit, il devient de plus en plus compliqué de la maintenir.

Voyons comment nous aurions écrit cette page en utilisant Django. Il faut noter que nous avons séparé notre fichier en trois fichiers Python (*models.py*, *views.py* et *urls.py*) et un gabarit HTML (*derniers_livres.html*).

models.py

```
# models.py (les tables de la base de données)

from django.db import models

class Livre(models.Model):
    nom = models.CharField(maxlength=50)
    date_publication = models.DateField()
```

views.py

```
# views.py (la logique métier)

from django.shortcuts import render_to_response
from models import Livre

def derniers_livres(request):
    liste_livres = Livre.objects.order_by('-date_publication')[:10]
    return render_to_response('derniers_livres.html', {'liste_livres': liste_livres})
```

urls.py

```
# urls.py (la configuration de l'URL)

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'derniers-livres/$', app.views.derniers_livres),
)
```

templates/derniers_livres.html

```
<!-- derniers_livres.html (le gabarit) -->

<ul>
    {% for livre in liste_livres %}
```

```
            <li>{{ livre.nom }}</li>
    {% endfor %}
</ul>
```

Ne vous préoccupez pas pour le moment de savoir comment cela fonctionne, je prendrai bien soin de vous l'expliquer.

Ce qu'il faut noter ici, c'est la répartition du code selon son objet :

- Le fichier *models.py* décrit la table pour stocker les données sous la forme d'une classe Python. Cette classe est appelée **modèle**.
- Le fichier *views.py* contient la logique de la page, sous la forme de la fonction Python *derniers_livres*. Cette fonction est appelée **vue**.
- Le fichier *urls.py* définit quelle vue sera appelée pour un modèle d'URL donné. Dans notre cas, *derniers-livres/* sera traité par la fonction *derniers_livres*.
- Le fichier *derniers_livres.html* est un gabarit HTML définissant l'aspect de la page. On l'appellera un **template**.

On nommera cette organisation de projet le **MTV** (Modèle Template Vue), proche parent du **MVC** (Modèle Vue Controleur).

CHAPTER 2

Installation

Installation et pré-requis

date 2012-04-30 10:52

tags django, python

category Django

author Rémy Hubscher

Pré-requis

Je ne vais pas m'attarder sur un troll, sachez juste que pour moi être développeur Django c'est peu compatible avec démarrer son ordinateur sous Windows.

Débrouillez-vous comme vous le souhaitez, (Dual-boot, VM, ..) mais démarrer sous un système unix (OS X, Ubuntu, ...)

Installer Django

Pour commencer on va installer les dépendances

```
$ easy_install pip
$ pip install MySQL-python PIL Django
```

Automatiquement, pip va vous installer la dernière version de Django et les dépendances.

Vous pouvez prendre la bonne habitude d'utiliser les virtualenv.

Si MySQL-python vous dit qu'il manque mysql_config, installez *libmysqlclient-dev*

```
$ sudo apt-get install libmysqlclient-dev
```

Si PIL vous mets que vous n'avez aucun support, installez certaines dépendances

```
sudo apt-get install libjpeg8 libjpeg8-dev
sudo apt-get install zlib1g-dev
sudo apt-get install libfreetype6 libfreetype6-dev
```

Si vous êtes sous x64 ajoutez

```
sudo ln -s /usr/lib/*/libjpeg.so /usr/lib
sudo ln -s /usr/lib/*/libz.so /usr/lib
sudo ln -s /usr/lib/*/libfreetype.so /usr/lib
```

Vérification de l'installation

```
$ python
>>> import django
>>> print django.VERSION
(1, 4, 0, 'final', 0)
>>> import Image
>>> print Image.VERSION
1.1.7
>>> import MySQLdb
>>> print MySQLdb.__version__
1.2.3
```

Lancer son premier projet django

Lancer un projet django

```
$ django-admin.py startproject tuto_django
$ cd tuto_django
$ python manage.py runserver
```

Ensuite cliquez ici : http://127.0.0.1:8000/

Installer PhpMyAdmin

Pour la suite, il va vous falloir une base de données MySQL.

Vous pouvez installer *phpmyadmin* pour gérer cette partie

```
$ sudo apt-get install phpmyadmin mysql-server apache2
```

Configuration du projet

date 2012-04-30 11:20

tags django, python

category Django

author Rémy Hubscher

Configuration

Maintenant que vous avez tout installé, il faut configurer son projet.

Commencez par créer une base de données MySQL avec un utilisateur:

```
user : tuto_django_user
passwd : tuto_django_pwd
bdd : tuto_django_bdd
```

Ensuite modifiez votre fichier *settings.py* comme ceci

```
# -*- coding: utf-8 -*-
# Django settings for tuto_django project.
import os

ABSOLUTE_PATH = os.path.dirname(__file__)

DEBUG = True
TEMPLATE_DEBUG = DEBUG

ADMINS = (
    ('Natim', 'natim@siteduzero.com'),
)

MANAGERS = ADMINS

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql', # Add 'postgresql_psycopg2', 'mysql',
↪'sqlite3' or 'oracle'.
        'NAME': 'tuto_django_bdd',                      # Or path to database file if␣
↪using sqlite3.
        'USER': 'tuto_django_user',                      # Not used with sqlite3.
        'PASSWORD': 'tuto_django_pwd',                  # Not used with sqlite3.
        'HOST': '',                      # Set to empty string for localhost. Not␣
↪used with sqlite3.
        'PORT': '',                      # Set to empty string for default. Not used␣
↪with sqlite3.
    }
}

# Local time zone for this installation. Choices can be found here:
# http://en.wikipedia.org/wiki/List_of_tz_zones_by_name
# although not all choices may be available on all operating systems.
# On Unix systems, a value of None will cause Django to use the same
# timezone as the operating system.
# If running in a Windows environment this must be set to the same as your
# system time zone.
TIME_ZONE = 'Europe/Paris'

# Language code for this installation. All choices can be found here:
# http://www.i18nguy.com/unicode/language-identifiers.html
LANGUAGE_CODE = 'fr-fr'

SITE_ID = 1

# If you set this to False, Django will make some optimizations so as not
# to load the internationalization machinery.
```

```
USE_I18N = True

# If you set this to False, Django will not format dates, numbers and
# calendars according to the current locale.
USE_L10N = True

# If you set this to False, Django will not use timezone-aware datetimes.
USE_TZ = True

# Absolute filesystem path to the directory that will hold user-uploaded files.
# Example: "/home/media/media.lawrence.com/media/"
MEDIA_ROOT = os.path.join(ABSOLUTE_PATH, 'medias')

# URL that handles the media served from MEDIA_ROOT. Make sure to use a
# trailing slash.
# Examples: "http://media.lawrence.com/media/", "http://example.com/media/"
MEDIA_URL = '/medias/'

# Absolute path to the directory static files should be collected to.
# Don't put anything in this directory yourself; store your static files
# in apps' "static/" subdirectories and in STATICFILES_DIRS.
# Example: "/home/media/media.lawrence.com/static/"
STATIC_ROOT = os.path.join(ABSOLUTE_PATH, 'collected_static')

# URL prefix for static files.
# Example: "http://media.lawrence.com/static/"
STATIC_URL = '/static/'

# Additional locations of static files
STATICFILES_DIRS = (
    # Put strings here, like "/home/html/static" or "C:/www/django/static".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
    # os.path.join(ABSOLUTE_PATH, 'static'),
)

# List of finder classes that know how to find static files in
# various locations.
STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
#    'django.contrib.staticfiles.finders.DefaultStorageFinder',
)

# Make this unique, and don't share it with anybody.
SECRET_KEY = '_$6a=hh50yz!o@(oks0+#6hx+8tmm3^ga#5_9%)xw0hrda%l^b'

# List of callables that know how to import templates from various sources.
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
#     'django.template.loaders.eggs.Loader',
)

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',
    # Uncomment the next line for simple clickjacking protection:
    # 'django.middleware.clickjacking.XFrameOptionsMiddleware',
)

ROOT_URLCONF = 'tuto_django.urls'

# Python dotted path to the WSGI application used by Django's runserver.
WSGI_APPLICATION = 'tuto_django.wsgi.application'

TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/django/templates
↪".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
    # os.path.join(ABSOLUTE_PATH, 'templates'),
)

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
)

# A sample logging configuration. The only tangible logging
# performed by this configuration is to send an email to
# the site admins on every HTTP 500 error when DEBUG=False.
# See http://docs.djangoproject.com/en/dev/topics/logging for
# more details on how to customize your logging configuration.
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'filters': {
        'require_debug_false': {
            '()': 'django.utils.log.RequireDebugFalse'
        }
    },
    'handlers': {
        'mail_admins': {
            'level': 'ERROR',
            'filters': ['require_debug_false'],
            'class': 'django.utils.log.AdminEmailHandler'
        }
    },
    'loggers': {
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': True,
        },
```

```
    }
}
```

Nous avons définit la configuration minimale pour notre projet. Nous allons maintenant pouvoir créer notre première app.

Configuration bonus

Un petit bonus, lorsque vous allez mettre en production votre serveur, il faudra bien penser à lui mettre une *SECRET_KEY* spécifique.

Pour se faire, vous pouvez génerer la nouvelle en faisant :

```
from random import choice
print ''.join([choice('abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*(-_=+)') for i in␣
↪range(50)])
```

N'oubliez pas de configurer votre VCS afin qu'il ne stocke pas votre fichier *settings.py* qui peut contenir des informations confidentielle telle qu'un mot de passe.

CHAPTER 3

TP : Une Todo list

TP : Gestion d'une liste de tâches - Models + Admin (1/3)

date 2012-04-30 12:51

tags django, python

category Django

author Rémy Hubscher

Énoncé

Pour découvrir Django en douceur, je vous propose une première petite app.

Il s'agit de gérer une liste de tâche.

Dans un premier temps, nous avons une unique liste.

1. Nous voulons ajouter des tâches dans la liste
2. Nous voulons dire que la tâche est réalisée (la barrer)
3. Nous voulons pouvoir vider la liste en fin de journée
4. Nous voulons pouvoir marquer toutes les tâches comme terminée
5. Nous voulons pouvoir supprimer une tâche

Créer une nouvelle app

Pour créer une app, vous devez entrer la commande suivante

```
$ python manage.py startapp todo
```

L'organisation des app se fait de la manière suivante :

1. Si votre app est spécifique à votre projet, vous devez la mettre dans le répertoire du projet

```
tuto_django/
- manage.py
- tuto_django
    - __init__.py
    - settings.py
    - todo
    |   - __init__.py
    |   - models.py
    |   - tests.py
    |   - views.py
    - urls.py
    - wsgi.py
```

2. Si votre app peut-être ensuite réutilisée, vous devez la mettre à côté du manage.py

```
tuto_django/
- manage.py
- todo
|   - __init__.py
|   - models.py
|   - tests.py
|   - views.py
- tuto_django
    - __init__.py
    - settings.py
    - urls.py
    - wsgi.py
```

Il ne faut pas oublier d'ajouter l'app dans le fichier de *settings.py* du projet à la variable *INSTALLED_APP*.

Soit *'tuto_django.todo'* pour 1. soit *'todo'* pour 2.

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.admin',
    'todo',
)
```

Ici on en profite aussi pour activer l'administration automatique de Django.

Models

Première chose à faire notre modèle de données pour stocker notre liste.

```
# -*- coding: utf-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _

class Task(models.Model):
    content = models.CharField(_(u'task'), max_length=255)
    is_resolved = models.BooleanField(_(u'Resolved?'))
```

```
    def __unicode__(self):
        return u'Task %d : %s' % (self.id, self.content)
```

Rien de bien compliquer pour commencer :

1. On créé le modèle d'une tâche
2. On définit qu'elle a un champ *content* qui va contenir l'énoncé de la tâche
3. On code toujours tout en anglais et on verra comment traduire ensuite
4. Dans les models on utilise *ugettext_lazy* pour ensuite pouvoir traduire notre app en français

Pour avoir la liste des fields disponibles c'est dans la documentation

Création de la BDD

On va maintenant créer la base de données avec notre projet

```
$ python manage.py syncdb
Creating tables ...
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_user_permissions
Creating table auth_user_groups
Creating table auth_user
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table todo_task

You just installed Django's auth system, which means you don't have any superusers␣
↪defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'natim'): admin
E-mail address: natim@siteduzero.com
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
```

Pour finir on va créer un super-utilisateur qui nous permettra d'administrer notre liste de tâche.

Création des urls

On va activer l'URL pour l'admin dans le fichier *urls.py*

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()
```

```
urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'tuto_django.views.home', name='home'),
    # url(r'^tuto_django/', include('tuto_django.foo.urls')),

    url(r'^admin/', include(admin.site.urls)),
)
```

Je laisse les deux examples car ils seront intéressant pour la suite.

Activation de l'admin de Todo

On va créer le fichier *admin.py* dans l'app *todo*

```
# -*- coding: utf-8 -*-
from django.contrib import admin
from todo.models import Task

admin.site.register(Task)
```

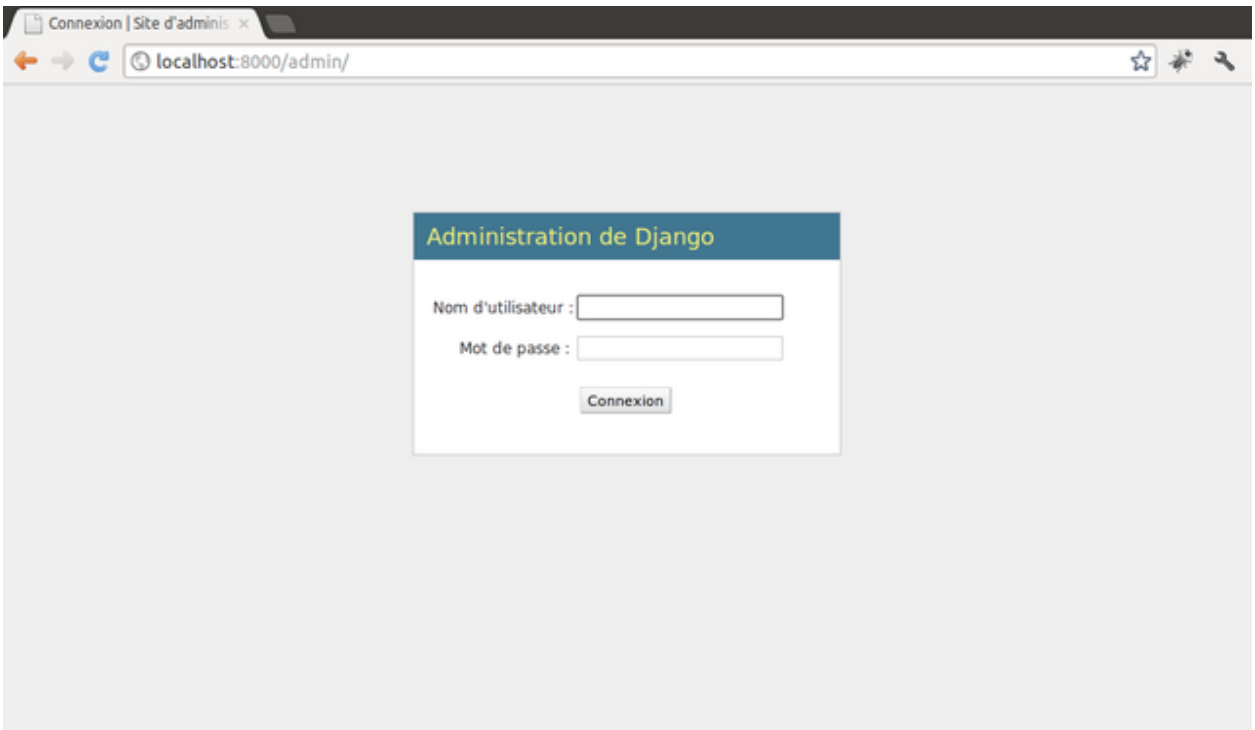
Lancement du serveur

Comme précédement on va lancer notre serveur

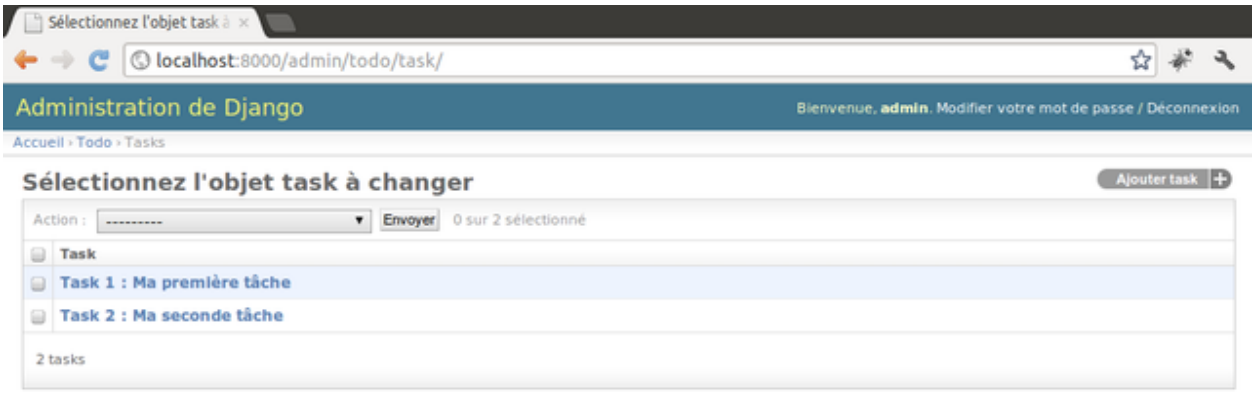
```
$ python manage.py runserver
```

On se rends ensuite ici : http://localhost:8000/admin/

On se connecte avec l'utilisateur créé lors du syncdb et on peut maintenant créer ses tâches.




Administration du site | Si
localhost:8000/admin/
Voulez-vous que Chromium enregistre votre mot de passe ?
Jamais pour ce site
Enregistrer le mot de passe
Administration de Django
Bienvenue, admin. Modifier votre mot de passe / Déconnexion
Administration du site
Auth
Groupes
Ajouter
Modifier
Utilisateurs
Ajouter
Modifier
Sites
Sites
Ajouter
Modifier
Todo
Tasks
Ajouter
Modifier
Actions récentes
Mes actions
Aucun(e) disponible


Sélectionnez l'objet task à
localhost:8000/admin/todo/task/
Administration de Django
Bienvenue, admin. Modifier votre mot de passe / Déconnexion
Accueil › Todo › Tasks
Sélectionnez l'objet task à changer
Ajouter task
Action :
Envoyer
0 sur 2 sélectionné
Task
Task 1 : Ma première tâche
Task 2 : Ma seconde tâche
2 tasks

TP : Gestion d'une liste de tâches - Templates (2/3)

date 2012-04-30 13:34

tags django, python

category Django

author Rémy Hubscher

Récapitulatif

Nous avons vu dans le chapitre précédent comment créer le models et l'administrer dans l'admin Django.

Je vous rappelle que, dans un premier temps, nous avons une unique liste :

1. Nous voulons ajouter des tâches dans la liste
2. Nous voulons dire que la tâche est réalisée (la barrer)
3. Nous voulons pouvoir vider la liste en fin de journée
4. Nous voulons pouvoir marquer toutes les tâches comme terminée
5. Nous voulons pouvoir supprimer une tâche

Les views

Pour commencer nous souhaitons afficher notre liste de tâche.

Nous commençons donc par faire le HTML/CSS/JS :

templates/todo/tasks-list.html

```
{% load i18n staticfiles %}
<!DOCTYPE html>
<html>
<head>
        <meta charset="utf-8">
        <title>{% trans 'Todo list' %}</title>
        <link rel="stylesheet" href="{% static 'todo/base.css' %}">
</head>

<body>

        <section id="todoapp">
                <header id="header">
                        <h1>{% trans 'Todos' %}</h1>
                        <input id="new-todo" placeholder="{% trans 'What needs to be
↪done?' %}" autofocus>
                </header>

                <section id="main">
                        <input id="toggle-all" type="checkbox" >
                        <label for="toggle-all">{% trans 'Mark all as complete' %}</
↪label>
                        <ul id="todo-list">

                                                <!-- C'est ici que ça se passe -->

                        </ul>
                </section>

                <footer id="footer">
                        <span id="todo-count"></span>
                        <button id="clear-completed">{% trans 'Clear completed' %}</
↪button>
                </footer>
        </section>

        <footer id="info">
                <p>{% trans 'Double-click to edit a todo.' %}</p>
        </footer>

</body>
</html>
```

static/todo/base.css

À récupérer ici : https://bitbucket.org/natim/django-story/raw/tip/demos/tuto_django/todo/static/todo/base.css

static/todo/bg.png

À récupérer ici : https://bitbucket.org/natim/django-story/raw/tip/demos/tuto_django/todo/static/todo/bg.png

Arborescence

Vous devez normalement maintenant avoir l'arborescence suivante

```
todo/
- admin.py
- __init__.py
- models.py
- static
|   - todo
|       - app.js
|       - base.css
|       - bg.png
- templates
|   - todo
|       - tasks-list.html
- tests.py
- views.py
```

Views

Nous allons ensuite écrire notre première views, qui va demander d'afficher la liste des tâches.

```
# -*- coding: utf-8 -*-

from django.views.generic import TemplateView

class TasksView(TemplateView):

    template_name="todo/tasks-list.html"
```

Cette views va pour l'instant simplement charger notre template *todo/tasks-list.html* et l'afficher.

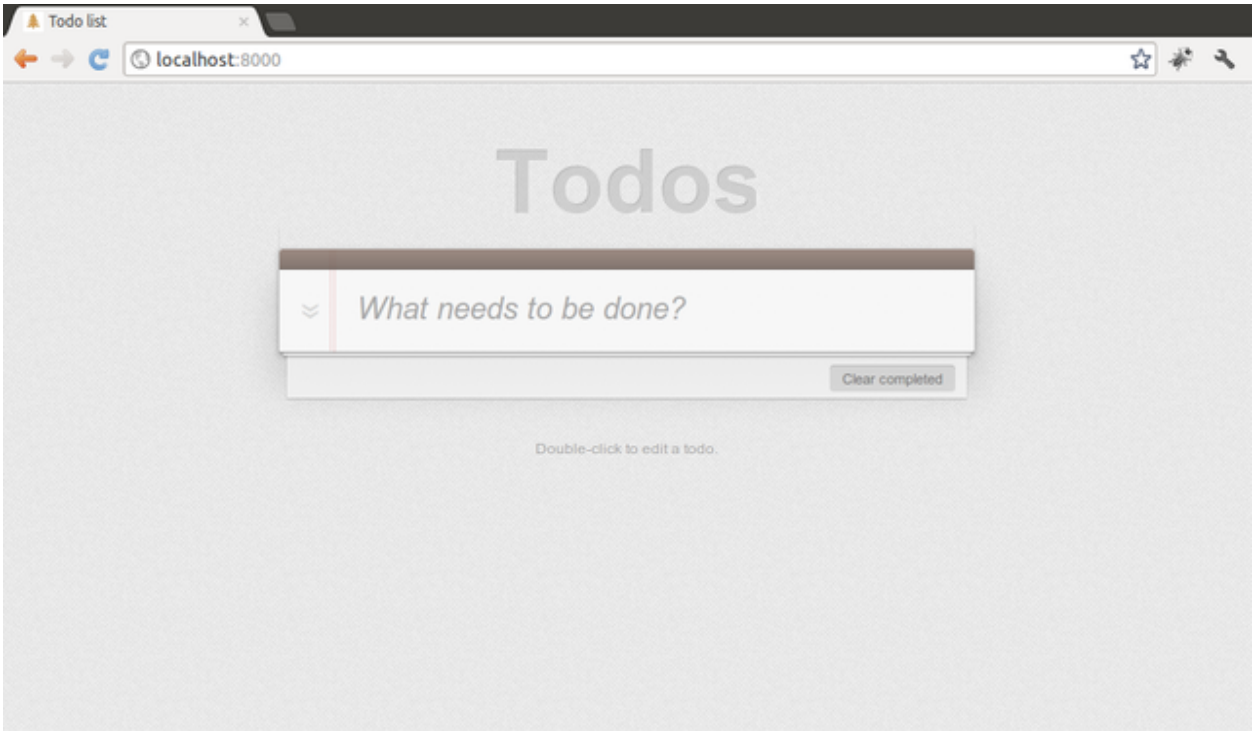
Url

Ne pas oublier de définir l'URL

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

from todo.views import TasksView

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^$', TasksView.as_view(), name='tasks-list'),
)
```

Le résultat

Connectez-vous à http://localhost:8000/ :



Afficher la liste des tâches

On va modifier le fichier *views.py* pour y ajouter des informations supplémentaires

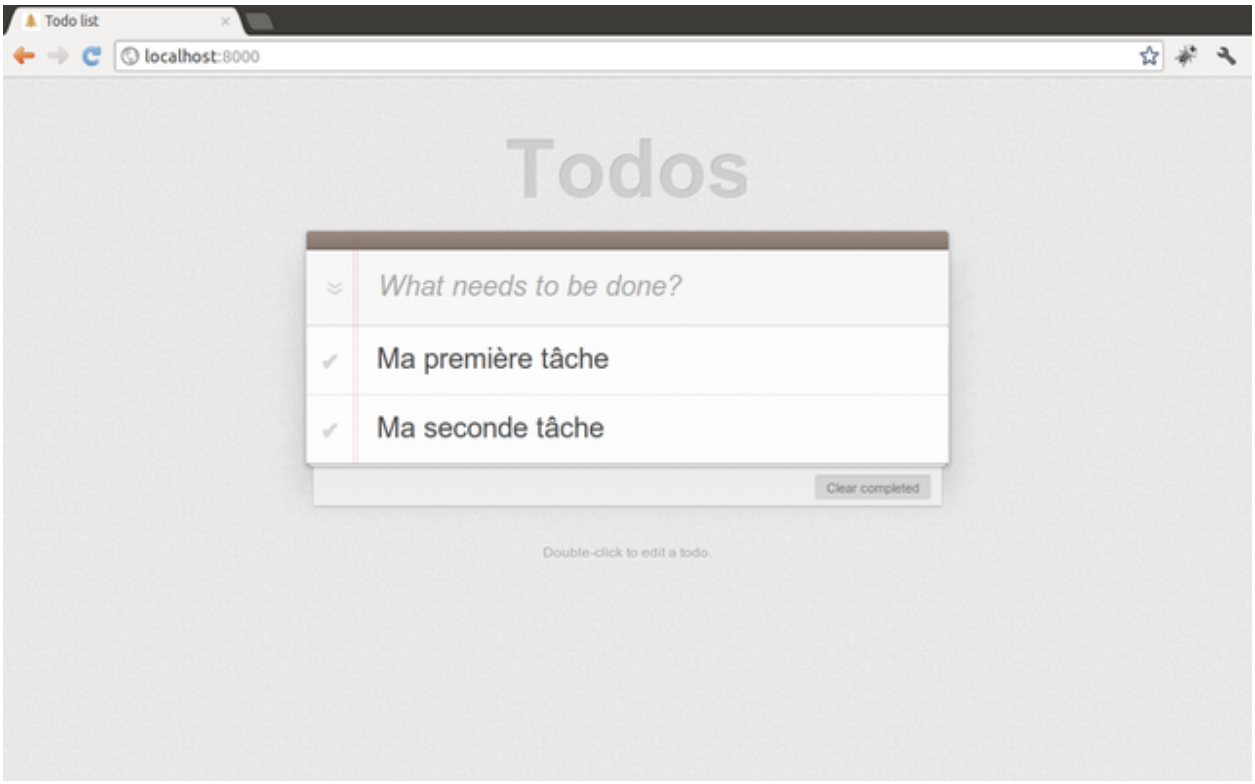
```
# -*- coding: utf-8 -*-

from django.views.generic import ListView
from todo.models import Task

class TasksView(ListView):

    template_name="todo/tasks-list.html"
    model = Task
```

On va modifier le fichier *tasks-list.html* pour y spécifier comment afficher les tâches

```
<ul id="todo-list">
      {% for task in object_list %}
      <li>
            <div class="view">
              <input class="toggle" type="checkbox">
              <label>{{ task.content }}</label>
              <button class="destroy"></button>
            </div>
            <input class="edit" value="{{ task.content }}">
      </li>
      {% endfor %}
</ul>
```

Le résultat

Connectez-vous à http://localhost:8000/ :



TP : Gestion d'une liste de tâches - Views et URLs (3/3)

date 2012-04-30 16:34

tags django, python

category Django

author Rémy Hubscher

Récapitulatif

Nous avons vu dans le chapitre précédent comment créer les views et afficher notre liste de tâches.

Je vous rappelle que, dans un premier temps, nous avons une unique liste :

1. Nous voulons ajouter des tâches dans la liste
2. Nous voulons pouvoir vider la liste en fin de journée
3. Nous voulons dire que la tâche est réalisée (la barrer)
4. Nous voulons pouvoir marquer toutes les tâches comme terminée
5. Nous voulons pouvoir supprimer une tâche

Nous allons maintenant en faire en sorte de pouvoir ajouter une tâche.

Ajouter une tâche

Nous allons mettre en place un form pour éditer notre modèle.

Créer un fichier *forms.py* et y ajouter ces informations.

forms.py

```
# -*- coding: utf-8 -*-
from django import forms
from todo.models import Task

class TaskForm(forms.ModelForm):
    class Meta:
        model = Task
        exclude = ('is_resolved',)
```

On va ensuite gérer ce formulaire dans une nouvelle vue

views.py

```
# -*- coding: utf-8 -*-

    # [...] On rajoute au document le code suivant

from django.views.generic import ListView, CreateView
from django.core.urlresolvers import reverse_lazy
from django.http import HttpResponseRedirect
    from todo.forms import TaskForm

class TaskCreateView(CreateView):
    form_class = TaskForm
    success_url  = reverse_lazy('tasks-list')

    def form_invalid(self, form):
        # Attention les erreurs du form ne seront pas affichées
        return HttpResponseRedirect(self.success_url)
```

Pour l'instant la seule erreur c'est que le champ soit vide. Dans notre cas, la tâche ne sera pas sauvegardée si on renvoit un *content* vide.

urls.py

On commence à avoir plusieurs urls pour notre module todo. Comme on souhaite qu'il soit réutilisable, on va mettre toutes les urls concernant les todos dans le fichier *todo/urls.py*

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url

from todo.views import *

urlpatterns = patterns('',
    url(r'^$', TasksView.as_view(), name='tasks-list'),

    url(r'^add/$', TaskCreateView.as_view(), name='task-create'),
)
```

Dans le fichier url de notre projet *tuto_django/urls.py*, on va inclure les urls des todo

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url
from django.views.generic import RedirectView

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^$', RedirectView.as_view(url='todo/')),
    url(r'^admin/', include(admin.site.urls)),
    url(r'^todo/', include('todo.urls')),
)
```

On demande aussi de rediriger notre page d'accueil vers *todo/*

tasks-list.html

```
<header id="header">
        <h1>{% trans 'Todos' %}</h1>
        <form action="{% url task-create %}" method="post">
            {% csrf_token %}
            <input id="new-todo" placeholder="{% trans 'What needs to be done?' %}"␣
↪name="content" autofocus>
        </form>
</header>
```

Le csrf_token est une sécurité de Django pour éviter qu'un script maveillant envoi notre formulaire sans le charger au préalable.

Le navigateur sait que lorsqu'on appuie sur *ENTER* il doit envoyer le formulaire.

Nous pouvons maintenant ajouter des tâches.

Supprimer les tâches terminées

views.py

```
TASK_LIST_URL = reverse_lazy('tasks-list')

def clear_resolved_tasks(request):
    if request.method == 'POST':
        # Modify an object in POST only
        Task.objects.filter(is_resolved=True).delete()
    return HttpResponseRedirect(TASK_LIST_URL)
```

Une toute petite fonction qui va récupérer les tâches terminées et les supprimer. On vérifie juste que la fonction a bien été appelé en *POST* car on modifie des données.

urls.py

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url

from todo.views import *

urlpatterns = patterns('',
    url(r'^$', TasksView.as_view(), name='tasks-list'),
    url(r'^clear/$', clear_resolved_tasks, name='tasks-clear'),
```

```
    url(r'^add/$', TaskCreateView.as_view(), name='task-create'),
)
```

tasks-list.html

```
<footer id="footer">
           <form action="{% url tasks-clear %}" method="post">
                   {% csrf_token %}
                   <button id="clear-completed" onclick="this.parentNode.submit();">{
↪% trans 'Clear completed' %}</button>
           </form>
</footer>
```

Marquer une tâche comme terminée

views.py

```
from django.shortcuts import get_object_or_404

def toggle_task(request, task_id):
    if request.method == 'POST':
        # Modify an object in POST only
        task = get_object_or_404(Task, pk=task_id)

        task.is_resolved = not task.is_resolved
        task.save()

    return HttpResponseRedirect(TASK_LIST_URL)
```

urls.py

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url

from todo.views import *

urlpatterns = patterns('',
    url(r'^$', TasksView.as_view(), name='tasks-list'),
    url(r'^clear/$', clear_resolved_tasks, name='tasks-clear'),

    url(r'^add/$', TaskCreateView.as_view(), name='task-create'),
    url(r'^toggle/(?P<task_id>\d+)/$', toggle_task, name='task-toggle'),
)
```

tasks-list.html

```
<form method="post" action="{% url task-toggle task.id %}">
        {% csrf_token %}
        <input class="toggle" type="checkbox"{% if task.is_resolved %} checked=
↪"checked"{% endif %} onclick="this.parentNode.submit();">
</form>
```

Marquer toutes les tâches comme terminées

views.py

```
def toggle_tasks(request):
    if request.method == 'POST':
        # Modify an object in POST only
        try:
            task = Task.objects.all()[0]
        except IndexError:
            task = None

        if task is not None:
            status = not task.is_resolved
            Task.objects.all().update(is_resolved=status)

    return HttpResponseRedirect(TASK_LIST_URL)
```

urls.py

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url

from todo.views import *

urlpatterns = patterns('',
    url(r'^$', TasksView.as_view(), name='tasks-list'),
    url(r'^clear/$', clear_resolved_tasks, name='tasks-clear'),
    url(r'^toggle/$', toggle_tasks, name='tasks-toggle'),

    url(r'^add/$', TaskCreateView.as_view(), name='task-create'),
    url(r'^toggle/(?P<task_id>\d+)/$', toggle_task, name='task-toggle'),
)
```

tasks-list.html

```
<form method="post" action="{% url tasks-toggle %}">
        {% csrf_token %}
        <input id="toggle-all" type="checkbox" onclick="this.parentNode.submit();">
</form>
```

Supprimer une tâche

views.py

```
from django.views.generic import ListView, CreateView, DeleteView

class TaskDeleteView(DeleteView):
    model = Task
    success_url = TASK_LIST_URL
```

urls.py

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url

from todo.views import *

urlpatterns = patterns('',
    url(r'^$', TasksView.as_view(), name='tasks-list'),
```

```
    url(r'^clear/$', clear_resolved_tasks, name='tasks-clear'),
    url(r'^toggle/$', toggle_tasks, name='tasks-toggle'),

    url(r'^add/$', TaskCreateView.as_view(), name='task-create'),
    url(r'^toggle/(?P<task_id>\d+)/$', toggle_task, name='task-toggle'),
    url(r'^delete/(?P<pk>\d+)/$', TaskDeleteView.as_view(), name='task-delete'),
)
```

tasks-list.html

```
<form method="post" action="{% url task-delete task.id %}">
        {% csrf_token %}
        <button class="destroy" onclick="this.parentNode.submit();"></button>
</form>
```

Conclusion

C'est extrèmement simple de faire une application web avec Django, la plupart des briques sont là et il ne reste plus qu'à les utiliser.

Le code complet est disponible ici : http://bitbucket.org/natim/django-story/src/tip/demos/tuto_django

Vous pouvez tester cette application ici : http://django-story.ionyse.com/demos/todo/

TP : Tester son application

date 2012-04-30 19:06

tags django, python

category Django

author Rémy Hubscher

Récapitulatif

Nous avons maintenant une application qui fonctionne.

Cependant nous allons sûrement la faire vivre et en modifiant quelque chose, on risque de créer des bugs.

Pour éviter cela, nous allons tester toutes nos views afin d'être sur qu'elle se comporte correctement.

Notre fichier de test

tests.py

Un squelette est créé automatiquement lors de la création de notre app.

```
"""
This file demonstrates writing tests using the unittest module. These will pass
when you run "manage.py test".

Replace this with more appropriate tests for your application.
"""
```

```
from django.test import TestCase


class SimpleTest(TestCase):
    def test_basic_addition(self):
        """
        Tests that 1 + 1 always equals 2.
        """
        self.assertEqual(1 + 1, 2)
```

Tester l'affichage de la liste

Pour commencer, nous allons créer une liste de tâche reprenant tous les cas de figures dans la méthode *setUp*

```
# -*- coding: utf-8 -*-
"""
Todo : Tests
"""
from django.test import TestCase
from django.core.urlresolvers import reverse

from todo.models import Task


class TodoTest(TestCase):

    def setUp(self):
        self.task1 = Task.objects.create(content=u'Ma première tâche', is_
↪resolved=True)
        task2 = Task.objects.create(content=u'Ma seconde tâche', is_resolved=False)
        task3 = Task.objects.create(content=u'Ma troisième tâche', is_resolved=True)
        task4 = Task.objects.create(content=u'Ma quatrième tâche', is_resolved=False)
```

Ensuite nous allons ajouter nos méthodes, une méthode par test

```
def test_task_list(self):
    url = reverse('tasks-list')

    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)
    self.assertEqual(len(response.context['object_list']), Task.objects.count())
```

Tester la suppression des taches terminées

```
def test_tasks_clear(self):
    url = reverse('tasks-clear')

    nb_tasks = Task.objects.count()
    response = self.client.post(url)
    self.assertEqual(nb_tasks-2, Task.objects.count())
```

Tester le changement de status de toutes les tâches

```
def test_tasks_toggle(self):
    url = reverse('tasks-toggle')

    nb_tasks = Task.objects.filter(is_resolved=True).count()
    response = self.client.post(url)
    self.assertEqual(0, Task.objects.filter(is_resolved=True).count())

    response = self.client.post(url)
    self.assertEqual(nb_tasks+2, Task.objects.filter(is_resolved=True).count())
```

Tester la création d'une tâche

```
def test_task_creation(self):
    url = reverse('task-create')

    nb_tasks = Task.objects.count()
    response = self.client.post(url, {'content': u'Ma cinquième tâche'})
    self.assertEqual(nb_tasks+1, Task.objects.count())
```

Tester le changement de statut d'une tâche

```
def test_task_toggle(self):
    task_id = self.task1.id
    url = reverse('task-toggle', args=[task_id])

    status = Task.objects.get(pk=task_id).is_resolved
    response = self.client.post(url)
    self.assertEqual(status, not Task.objects.get(pk=task_id).is_resolved)
```

Tester la suppression d'une tâche

```
def test_task_delete(self):
    task_id = self.task1.id
    url = reverse('task-delete', args=[task_id])

    nb_tasks = Task.objects.count()
    response = self.client.post(url)
    self.assertEqual(nb_tasks-1, Task.objects.count())

    response = self.client.post(url)
    assertEqual(response.status_code, 404)
```

Conclusion

Rien de bien compliqué, pour lancer nos tests

```
$ python manage.py test todo
Creating test database for alias 'default'...
......
----------------------------------------------------------------------
Ran 6 tests in 0.206s

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

Il faudra, au préalable, créer la base *test_tuto_django_bdd* et donner les droits à l'utilisateur *tuto_django_user*.

CHAPTER 4

Tour d'horizon

Les modèles - models.py

date 2012-05-01 17:52

tags django, python

category Django

author Rémy Hubscher

Le MVT de Django

Le découpage d'une application Django se fait en trois parties :

1. Le modèle
2. La vue
3. Le template

Vous avez déjà eu un léger aperçu de ces trois parties dans le TP précédent.

Le modèle

Django est basé sur un ORM. On va donc créer des objets Python qui vont être capable de se caller automatiquement à une base de données relationnelle.

Cela va nous permettre de faire tourner notre application sur le SGBD de notre choix :

- SQLite,
- MySQL,
- PostgreSQL,

- Oracle,
- MSSQL,
- etc.

Un modèle Django est donc simplement une classe Python qui hérite de *django.db.models.Model*.

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(u'Firstname', max_length=30)
    last_name = models.CharField(u'Lastname', max_length=30)
```

Les champs disponibles

Il en existe un certain nombre : https://docs.djangoproject.com/en/dev/ref/models/fields/

Certains que vous allez utiliser très souvent, d'autres moins souvent :

- BooleanField
- CharField
- IntegerField
- DateField
- DateTimeField
- EmailField
- ImageField
- FileField
- IPAdressField
- SlugField
- TextField
- URLField
- etc.

Vous allez même parfois devoir en créer pour vos propres besoins.

Chaque champ peut avoir des arguments spécifiques et une fonction de validation spécifique.

Utilisation d'un ModelField

Pour chaque field, il y a des paramètres.

1. Le premier, commun à tous, est le label
2. Ensuite on utilise des paramètres nommés spécifiques à chaque field.

Quelques paramètres globaux :

- *unique* : Boolean - Permet de spécifier que la valeur est unique (le champ identifie donc la fiche)
- *blank* : Boolean - On autorise le champ a être vide

- *null* : Boolean - Si le champ est vide, on stocke le champ comme Null - À utiliser pour les champs qui ne sont pas des chaines de caractères si vous acceptez qu'ils soient vides.
- *choices* : Tuple - Permet le choix d'une valeur dans une liste définie, éventuellement avec des groupes de choix.
- *default* : La valeur par défaut lorsque le champ n'est pas saisie.
- *editable* : Boolean - Si on le met à *False* le champ ne sera pas éditable par l'utilisateur
- *verbose_name* : String - Nom du champ pour les utilisateurs
- *help_text* : String - En plus du verbose_name permet de définir une explication sur ce que doit contenir le champ.

Il y a d'autres paramètres que vous pouvez trouver dans la documentation.

CharField

C'est une zone de texte simple avec une taille maximale, *max_length* comprise entre 0 et 255.

S'il vous faut plus, utilisez un TextField.

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

GENDER_CHOICES = (('Mr',  _('Mister')),
                  ('Mrs', _('Madam')),
                  ('Miss', _('Miss')))

class Person(models.Model):
    gender = models.CharField(_('gender'), max_length=4, choices=GENDER_CHOICES)
    first_name = models.CharField(_('firstname'), max_length=30)
    last_name = models.CharField(_('lastname'), max_length=30)

    def __unicode__(self):
        return u'%s %s %s' % (self.gender, self.first_name, self.last_name)
```

DateField et DateTimeField

Ces deux champs sont semblable, l'un contient une date et l'autre la date et l'heure.

Deux paramètres intéressant :

- *auto_now* : Boolean - Est mise à jour automatiquement dès que l'objet est sauvegardé.
- *auto_now_add* : Boolean - Est mise à jour automatiquement lors de la création de l'objet.

Lorsque *auto_now* ou *auto_now_add* sont sélectionné *editable* est automatiquement mis à False et *blank* à True.

ImageField et FileField

- *upload_to* : Le chemin vers lequel enregistrer le fichier dans le *MEDIA_ROOT*, ce peut aussi être l'adresse d'une fonction (un callable) qui va s'occuper de retourner le nom du fichier.

```
from django.db import models
from django.utils.translation import ugettext_lazy as _
from django.template.defaultfilters import slugify
from django.utils.encoding import smart_str
import os.path
```

```
def upload_to_valid_name(prefix_dir):

    def get_valid_name(instance, name):
        root, ext = os.path.splitext(name)
        root = smart_str(slugify(root).replace('-', '_'))
        return os.path.join(prefix_dir, '%s.%s' % (root, ext))

    return get_valid_name

class Firm(models.Model):
    name = models.CharField(_(u'name'), max_length=50,
                            help_text=_(u"Enter the name of the firm"))
    image = models.ImageField(_(u'logo'),
                              upload_to=upload_to_valid_name('uploads/logo'),
                              null=True, blank=True)

    def __unicode__(self):
        return u'%s' % self.name
```

La méthode *__unicode__*

La méthode *__unicode__* permet de définir le nom de l'instance de l'objet.

Elle est utilisé notamment lorsqu'on fait un *print* de l'objet.

La class *Meta*

Elle permet de définir des informations sur l'objet.

Notamment le *verbose_name* qui définit le nom de l'objet et le *ordering* qui définit l'ordre de tri par défaut des objets.

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

GENDER_CHOICES = (('Mr',  _('Mister')),
                  ('Mrs', _('Madam')),
                  ('Miss', _('Miss')))

class Person(models.Model):
    gender = models.CharField(_('gender'), max_length=4, choices=GENDER_CHOICES)
    first_name = models.CharField(_('firstname'), max_length=30)
    last_name = models.CharField(_('lastname'), max_length=30)

    def __unicode__(self):
        return u'%s %s %s' % (self.gender, self.first_name, self.last_name)

    class Meta:
        verbose_name = _('person')
        verbose_name_plural = _('people')
        ordering = ['last_name', 'first_name']
```

Autres méthodes

Voici quelques informations intéressante, vous pouvez aussi ajouter les méthodes que vous souhaites à vos objets, pour retourner l'âge à partir de la date de naissance par exemple.

```
from datetime import date

from django.db import models
from django.utils.translation import ugettext_lazy as _

class Person(models.Model):
    name = models.CharField(_('name'), max_length=60,
                            help_text=_('Enter the person full name'))
    dob  = models.DateField(_('date of birth'))

    def __unicode__(self):
        return u'%s' % self.name

    def age(self):
        today = date.today()
        num_years = int((today - self.dob).days / 365.2425)
        return num_years
```

Lors de l'utilisation :

```
$ python manage.py shell
>>> from person.models import Person
>>> from datetime import date
>>> me = Person(name=u'Rémy Hubscher', dob=date(1987, 2, 21))
>>> print me
Rémy Hubscher
>>> me.age()
25
```

Conclusion

Voici donc un bref aperçu des modèles.

Une fois le modèle fait, il faut le créer dans la base de données

```
$ python manage.py syncdb
```

Vous pouvez aussi utiliser django-south pour gérer la modification de vos modèles sans perdre les données qui sont dedans.

Les templates - templates/

date 2012-05-01 19:12

tags django, python

category Django

author Rémy Hubscher

Un système hiérarchique

Vous faites du web, donc vous avez déjà du découper un design.

La première chose que l'on constate c'est que le graphisme reste quasiment le même d'une page à l'autre et il n'y a finalement que le bloc de contenu qui change et le menu sélectionné dans la barre de navigation.

Les templates de Django parte de ce postulat pour nous proposer un système extensible.

Pour mettre en place ses templates Django, nous allons donc créer le template de base, souvent appelé *base.html*.

Il va définir la structure de notre design puis nous allons l'étendre pour chaque page différente.

Les blocs

Les parties que nous souhaitons pouvoir surcharger sont appelé des blocks.

Voici un template de base :

```
{# Minimum HTML5 template #}
<!DOCTYPE html>
<html lang="fr">
  <head>
    {% block head %}
      <title>{{ PAGE_TITLE }} - {{ WEBSITE_TITLE }}</title>
      <meta charset="utf-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <!-- Le HTML5 shim, for IE6-8 support of HTML5 elements -->
      <!--[if lt IE 9]>
    <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
      <![endif]-->
      {% block head-medias %}{% endblock %}
    {% endblock %}
  </head>

  <body>
    {% block body %}

      <!-- Header -->
        <header>
          {% block header %}
                      <h1>Tuto Django</h1>
                      {% endblock %}
        </header>
      <!-- End of Header -->

      <!-- Navigation -->
      <nav>
        {% block navigation %}
                    <ul>
                      <li><a href="#">Accueil</a></li>
                      <li><a href="#">Mon portfolio</a></li>
                      <li><a href="#">Contact</a></li>
                    </ul>
        {% endblock %}
      </nav>
      <!-- End of Navigation -->

      <!-- Content -->
```

```
      <section id="main-content">
        {% block content %}{% endblock %}
      </section>
      <!-- End of Content -->

      <!-- Footer -->
      <footer>
        {% block footer %}
          <p>&copy 2012 - Rémy HUBSCHER</p>
        {% endblock %}
      </footer>
      <!-- End of Footer -->
    {% endblock %}

    {% block extra-medias %}{% endblock %}
  </body>
</html>
```

Bon effectivement il ne fait pas grand chose mais on a définit un certain nombre de block.

L'héritage

Pour étender du template de base et surcharger le contenu, on peut faire :

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

{% block content %}
    <h1>Bienvenue sur mon site internet</h1>
<p>
  Lorem ipsum dolor sit amet, consectetur adipiscing
  elit. Phasellus risus dolor, porttitor in laoreet non, posuere at
  mi. Nullam ultricies congue nunc, et accumsan ipsum pharetra
  non. Quisque vitae metus orci. Sed consequat condimentum ligula eu
  volutpat. Nulla facilisi. Suspendisse ac viverra elit. Mauris eget
  felis nec nisi cursus aliquet. Nulla facilisi.
</p>
<p>
  Vivamus quis nunc nibh. Mauris diam tellus, tincidunt quis adipiscing
  eleifend, auctor vel tortor. Vivamus fermentum ipsum quis ligula
  ornare et dictum mi suscipit. Praesent malesuada scelerisque sapien
  quis congue. Aenean leo turpis, consequat at pulvinar eu, dignissim id
  sapien. Nullam sit amet neque tortor, et interdum ipsum. Pellentesque
  porttitor sollicitudin diam quis lacinia. Vestibulum pharetra dictum
  arcu, nec cursus diam pretium ut. Nullam ultrices congue elit ac
  vulputate. Curabitur iaculis massa commodo elit viverra ornare. Nullam
  bibendum augue a nisi lobortis ornare. Donec lobortis, magna ut
  elementum eleifend, dolor justo aliquet quam, vitae ornare ipsum augue
  sed massa.
</p>
{% endblock %}
```

Le tag *extends* doit être seul sur la première ligne du fichier de template.

Le context

Dans notre vue, on va passer un *context* de rendu au template.

Ce sont des variables qui seront disponible dans notre template

En passant ce *context* :

```
{'PAGE_TITLE': 'Accueil',
 'WEBSITE_TITLE': 'Tuto Django'}
```

Les variables seront remplacé dans mon template.

Pour afficher une variable on utilise :

```
{{ PAGE_TITLE }}
```

Avec PAGE_TITLE le nom de la variable que l'on souhaite afficher.

On a aussi accès au champ des objets avec . :

```
{{ me.age }}
```

Pour afficher l'âge de la person *me*, il faudra passer l'objet *Person* en question au *context*:

```
from django.shortcuts import render_to_response, get_object_or_404
from person.models import Person

def profile(request, person_id):
    me = get_object_or_404(Person, pk=person_id)
    return render_to_response('person/profile.html', {'me': me})
```

get_object_or_404 permet de sécuriser l'accès à ma view en récupèrant l'objet ou en disant qu'il n'existe pas (erreur 404).

render_to_response permet de donner le nom du template et le context associé puis de retourner directement la réponse au format html.

Les tags et les filters

Puisqu'on vient de parler du tag *extends* il y a les *tags* et les *filters* qui permette de jouer sur le rendu de la page.

Tout est là : https://docs.djangoproject.com/en/dev/ref/templates/builtins/

Rien de bien compliqué :

Les tags

En deux parties :

```
{% autoescape on %}
  {{ body }}
{% endautoescape %}
```

Il y a le nom de la commande entre *{% commande %}* et pour fermer *{% endcommande %}*

En une partie :

```
{% csrf_token %}
```

Il n'y a pas de commande fermante.

Les filters

Il se collent à une variable et on peut les chaîner pour modifier l'affichage de cette dernière :

```
{{ me.dob|date:'l d F Y- H:i' }}

{{ me.name|wordcount }}
```

Conclusion

C'est assez complet, je vous invite à lire la doc, mais rien de bien compliqué en soit si on a compris ces quelques points.

Bonus : django-sekizai

Si vous avez à gérer des templates compliqués avec de la gestion d'inclusion multiples de fichiers css et js, alors il faut jeter un œil à django-sekizai

Les vues - views.py

date 2012-05-01 19:58

tags django, python

category Django

author Rémy Hubscher

Les views

C'est le dernier point de notre modèle MTV, il s'agit de faire le lien entre les modèles et les templates.

On peut les faire à l'aide de simple fonction qui prenne au minimum une *request* comme argument et retourne un *HttpResponse*:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse('<h1>Hello world</h1>')
```

L'objet *request*

Il contient les informations renvoyée par WSGI plus certaines déjà préparées :

- *request.method* le verbe HTTP utilisé pour atteindre la page
- *request.GET* un dictionnaire avec le querystring
- *request.POST* les données envoyées en POST
- *request.FILES* les informations sur les fichiers lors de l'upload de fichiers

Plus d'infos ici : https://docs.djangoproject.com/en/dev/ref/request-response/#httprequest-objects

Les vues génériques

Les *class-based views* ont fait leur apparition avec Django 1.3.

Ça permet de considérer plus facilement une vue comme une ressource HTTP car c'est le verbe HTTP qui définie la méthode qui sera appellée.

De plus elles utilisent l'héritage multiple de Django pour mettre ensemble des mixins et faire très simplement les actions souhaités en en écrivant le moins possible.

C'est très DRY

On a déjà parlé des *ListView*, *CreateView*, *TemplateView* dans le TP précédent.

Pour plus d'informations : https://docs.djangoproject.com/en/dev/ref/class-based-views/

Les urls

On ne peut pas parler des *views* sans parler des *urls*.

En effet on va pouvoir "brancher" une vue sur une ou plusieurs urls.

Pour cela on définit un *urlpattern* dans le fichier *urls.py* :

```
# -*- coding: utf-8 -*-
from django.views.generic import TemplateView

urlpatterns = patterns('',
    url(r'^$', 'person.views.index'),
    url(r'^profile/$', TemplateView.as_view(template_name='profile.html')),
)
```

Ici on utilise la fonction définie ci-dessus pour la page d'accueil et pour la page profile, la *TemplateView* qui va simplement charger le template *profile.html*.

Urls et Expression Régulières

En fait ce que je ne vous dis pas, c'est que les urls c'est très puissant.

On peut définir des expressions régulières pour passer des arguments à notre vue.

Par exemple, si je veux afficher le profil d'une personne avec cette vue :

```
from django.shortcuts import render_to_response, get_object_or_404
from person.models import Person

def profile(request, person_id):
    me = get_object_or_404(Person, pk=person_id)
    return render_to_response('person/profile.html', {'me': me})
```

Et ce template

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

{% block content %}
  <p><strong>{{ me.name }}</strong> a <em>{{ me.age }} ans</em>.</p>
  <p>Cette personne est née le <em>{{ me.dob|date:'d/m/Y' }}</em>.</p>
{% endblock %}
```

Et bien je vais pouvoir créer cette URL :

```
# -*- coding: utf-8 -*-
from django.views.generic import TemplateView

urlpatterns = patterns('',
    url(r'^profile/(?P<person_id>\d+)/$', 'person.views.profile'),
)
```

- \d+ veut dire qu'on attends plusieurs entiers à cet endroit.
- *(?P<person_id>d+)* veut dire que cette valeur sera passé comme *person_id* à la view.

On peut aussi passer l'information sous cette forme :

```
# -*- coding: utf-8 -*-
from django.views.generic import TemplateView

urlpatterns = patterns('',
    url(r'^profile/(?P<person_id>\d+)/$', 'person.views.profile'),
    url(r'^remy/$', 'person.views.profile', {'person_id': 1}),
)
```

Ainsi, l'URL http://servername/remy/ affichera le profil de la fiche numéro 1 tandis qu'avec l'autre URL on utilisera : http://servername/profile/1/

Conclusion

Voici ce qu'il fallait savoir sur les views et les urls.

Internationalisation

date 2012-05-01 20:46

tags django, python

category Django

author Rémy Hubscher

Coder en Anglais, déployer en Français

Le code Python a une syntaxe proche de l'anglais.

while, *if*, *for*, *open*, etc.

Par conséquent arriver avec du Français là dedans n'est pas une bonne idée.

De plus, comme vous utilisés des produits Open Source, vous avez un rôle à jouer pour déployer vous aussi vos programmes en Open Source ou participer à des projets existants que vous allez utiliser et sûrement patcher.

Il faut donc prendre l'habitude de coder en Anglais.

D'un autre côté, nos utilisateurs eux sont français et comme dans un premier temps notre projet n'a pas une visé internationale, on s'en fout un peu que tout s'affiche en Anglais.

Heureusement Django a prévu le coup.

Babel s'appelle gettext

gettext c'est une application qui a pour unique but de faire de la traduction de programmes open source. Quasiment tous les programmes Open Source l'utilise pour l'internationalisation de leur programme.

Django n'échappe pas à la règle car *gettext* fonctionne plutôt bien.

Intégrer gettext dans son code

Pour traduire des chaines de caractères

Aux endroits où l'on souhaite traduire, on va utiliser :

```
from django.utils.translation import ugettext as _
translated_string = _('This will be translated')
```

Dans les models, on utilise toujours *ugettext_lazy* qui permet de retarder la traduction de la chaîne au moment de l'affichage.

Si on traduit une phrase avec des valeurs issues de variables, il faut penser à nommer les variables pour que le traducteur s'y retrouve :

```
output = _('Today is %(month)s %(day)s.') % {'month': m, 'day': d}
```

Si on souhaite gérer les accords en nombre, on utilise *ungettext* :

```
from django.utils.translation import ungettext

def hello_world(request, count):
    page = ungettext(
        'there is %(count)d object',
        'there are %(count)d objects',
    count) % {
        'count': count,
    }
    return HttpResponse(page)
```

Dans les templates

Dans les templates, on utilise un templatetag de la librairie *i18n*

```
{% load i18n %}
{% trans 'This will be translated' %}
```

Si l'on a des variables dans la chaine à traduire

```
{% blocktrans %}This string will have {{ value }} inside.{% endblocktrans %}
```

S'il s'agit du champ d'une variable ou que l'on souhaite faire passer la valeur dans un filter

```
{% blocktrans with amount=article.price %}
That will cost $ {{ amount }}.
{% endblocktrans %}

{% blocktrans with myvar=value|filter %}
```

```
This will have {{ myvar }} inside.
{% endblocktrans %}
```

Pour plus d'information : https://docs.djangoproject.com/en/dev/topics/i18n/translation/

Traduire son application

Pour l'utiliser c'est assez simple, on se rends dans le répertoire de son application, on créé un répertoire *locale* et on lance

```
$ cd tuto_django/todo/
$ mkdir locale
$ django-admin.py makemessage -l fr
processing language fr
```

Cela va nous créer le fichier : *tuto_django/todo/locale/fr/LC_MESSAGES/django.po*

On peut éditer ce fichier et entrer notre traduction

```
# Traduction de notre application de liste de tâches
# Copyright (C) 2012 Rémy HUBSCHER
# This file is distributed under the same license as the PACKAGE package.
# Rémy HUBSCHER <remy.hubscher@ionyse.com>, 2012
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: Todos\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2012-05-01 20:54+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"
"Plural-Forms: nplurals=2; plural=(n > 1)\n"

#: models.py:6
msgid "task"
msgstr "tâche"

#: models.py:7
msgid "Resolved?"
msgstr "Terminée ?"

#: templates/todo/tasks-list.html:6
msgid "Todo list"
msgstr "Liste de tâches"

#: templates/todo/tasks-list.html:14
msgid "Todos"
msgstr "À faire"

#: templates/todo/tasks-list.html:17
msgid "What needs to be done?"
```

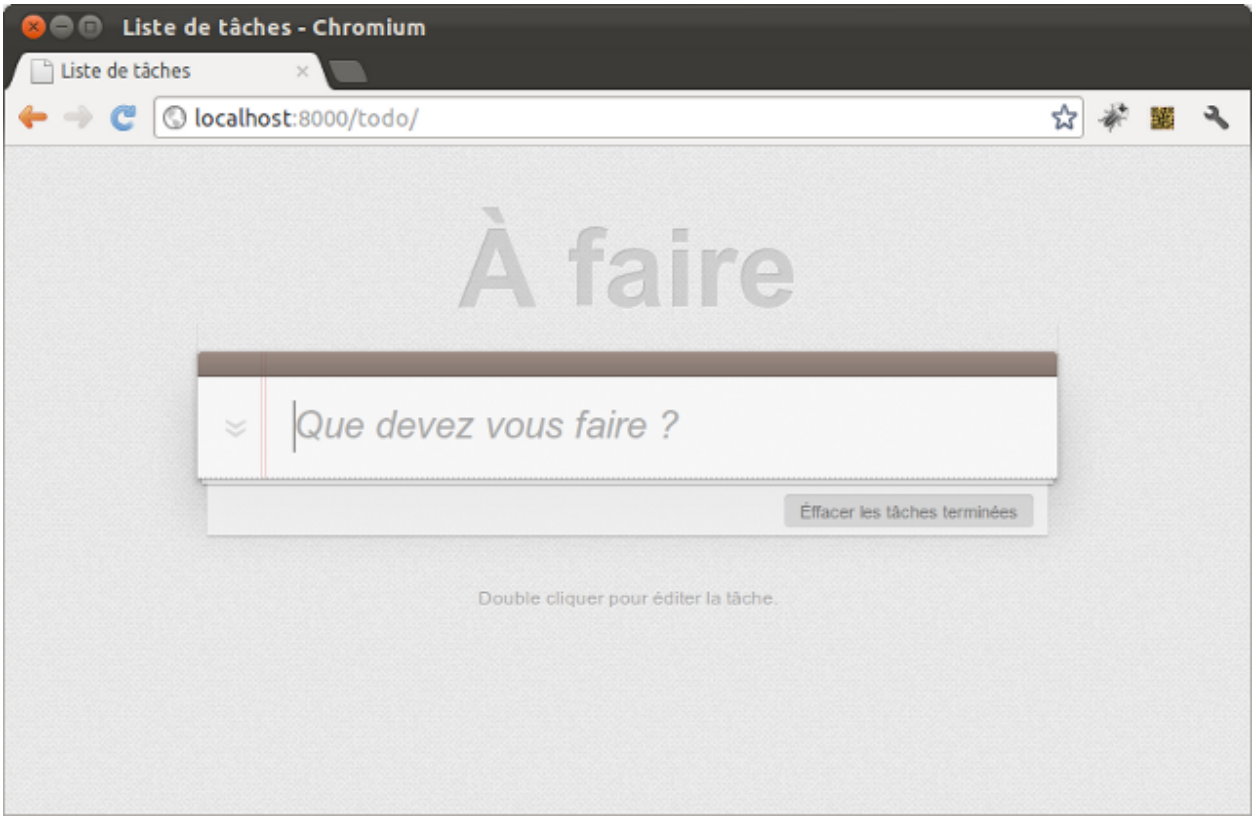
```
msgstr "Que devez vous faire ?"

#: templates/todo/tasks-list.html:26
msgid "Mark all as complete"
msgstr "Toutes marquer comme terminées"

#: templates/todo/tasks-list.html:50
msgid "Clear completed"
msgstr "Éffacer les tâches terminées"

#: templates/todo/tasks-list.html:56
msgid "Double-click to edit a todo."
msgstr "Double cliquer pour éditer la tâche."
```

Compiler le fichier de traduction

Une fois ceci fait

```
$ django-admin.py compilemessages
processing file django.po in tuto_django/todo/locale/fr/LC_MESSAGES
```

Ensuite puisque votre settings définit le français comme étant la langue par défaut, après avoir relancé votre serveur, votre application est maintenant en Français.



Informations complémentaires

Pour rajouter une langue, vous refaites un *makemessages -l de* et vous avez un fichier prêt à être traduit en allemand que vous pouvez donner à votre traducteur.

Lorsqu'il y a des modifications, il suffit de relancer la commande *makemessages -l fr* pour mettre à jour le fichier po.

Les lignes précédés de *#, fuzzy* ne sont pas prises en compte lors de la compilation, il faut donc vérifier la traduction puis supprimer le *#, fuzzy*.

Conclusion

Sans rien faire de spécial, avec juste un peu de rigueur, vous avez maintenant une application en Français et en Anglais prette à l'emploi.

CHAPTER 5

TP : Un agenda web

TP : Agenda - Models

date 2012-05-01 21:34

tags django, python

category Django

author Rémy Hubscher

Énoncé

Nous souhaitons réaliser un agenda qui nous permettra :

- De gérer plusieurs agenda
- D'intégrer des événements dans l'agenda
- D'ajouter, modifier, supprimer, déplacer un événement

L'IHM

En fait vous allez voir que ce qu'il y a de plus dur dans un agenda web, c'est l'IHM.

Ce n'est pas le but de ce TP, mais à quoi bon avoir un code basé sur Django si au niveau du frontend, la qualité ne suit pas ?

Comme IHM, je vous propose d'utiliser FullCalendar.

Création de l'app

```
$ python manage.py startapp schedule
```

C'est une app réutilisable.

On ajoute *schedule* dans le *INSTALLED_APP*.

Gérer les migrations avec South

Cette fois, comme nous allons sûrement améliorer notre application par la suite en ajoutant un auteur, une description, un lieu, etc. Nous allons donc utiliser South

```
$ pip install South
```

On ajoute *south* dans le *INSTALLED_APP* et on lance le *syncdb*

```
$ python manage.py syncdb
```

Les modèles

Nous avons dit, plusieurs calendriers/agenda, il faut donc stocker les agendas.

models.py

```
# -*- coding: utf-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _

class Calendar(models.Model):
    name = models.CharField(_('name'), max_length=50)
    slug = models.SlugField(_('slug'), unique=True)

    def __unicode__(self):
        return u'%s' % self.name

    class Meta:
        verbose_name = _('calendar')
        ordering = ['name']
```

Ensuite nous souhaitons créer des événements liés à un calendrier :

```
class Event(models.Model):
    title = models.CharField(_('title'), max_length=100)
    start = models.DateTimeField(_('start'))
    end = models.DateTimeField(_('start'))

    calendar = models.ForeignKey(Calendar)

    def __unicode__(self):
        return u'%s' % self.title

    class Meta:
        verbose_name = _('event')
        ordering = ['start', 'end']
```

Notre première migration

On ne fait pas de *syncdb*, mais on va créer notre migration initiale

```
$ python manage.py schemamigration --initial schedule
Creating migrations directory at '/home/rhubscher/hg/django-story/demos/tuto_django/
↪schedule/migrations'...
Creating __init__.py in '/home/rhubscher/hg/django-story/demos/tuto_django/schedule/
↪migrations'...
 + Added model schedule.Calendar
 + Added model schedule.Event
Created 0001_initial.py. You can now apply this migration with: ./manage.py migrate␣
↪schedule
```

Appliquer une migration

Pour appliquer notre migration, nous pouvons faire

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

Le premier syncdb

Lors de la création d'une base neuve, il sera possible de faire

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

ou en une fois

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

Ainsi les app gérées avec *South*, feront le *migrate* et les autres le *syncdb*.

Migration suivante

Si nous souhaitons avoir la possibilité d'annuler un événement, nous pouvons modifier notre modèle comme suit :

```
class Event(models.Model):
    title = models.CharField(_('title'), max_length=100)
    start = models.DateTimeField(_('start'))
    end = models.DateTimeField(_('end'))
    is_cancelled = models.BooleanField(_('Cancelled?'), default=False, blank=True)

    calendar = models.ForeignKey(Calendar)

    def __unicode__(self):
        return u'%s' % self.title

    class Meta:
        verbose_name = _('event')
        ordering = ['start', 'end']
```

Et nous voyons ici tout l'intérêt de *South*.

Précédement, il nous aurait fallu supprimer la table event pour la recréer. Maintenant, nous allons simplement faire une nouvelle migration

```
$ python manage.py schemamigration --auto schedule
 + Added field is_cancelled on schedule.Event
Created 0002_auto__add_field_event_is_cancelled.py. You can now apply this migration␣
↪with: ./manage.py migrate schedule
```

Puis on applique la migration

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

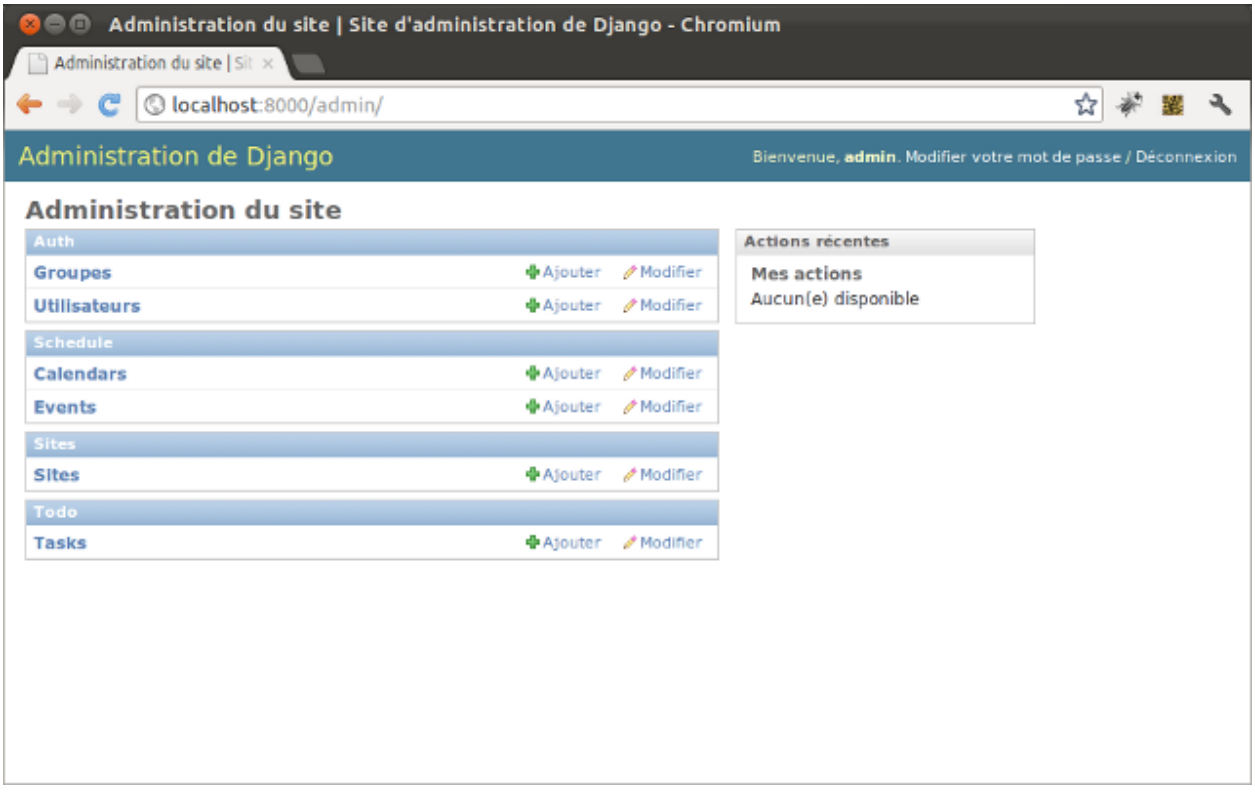
L'administration

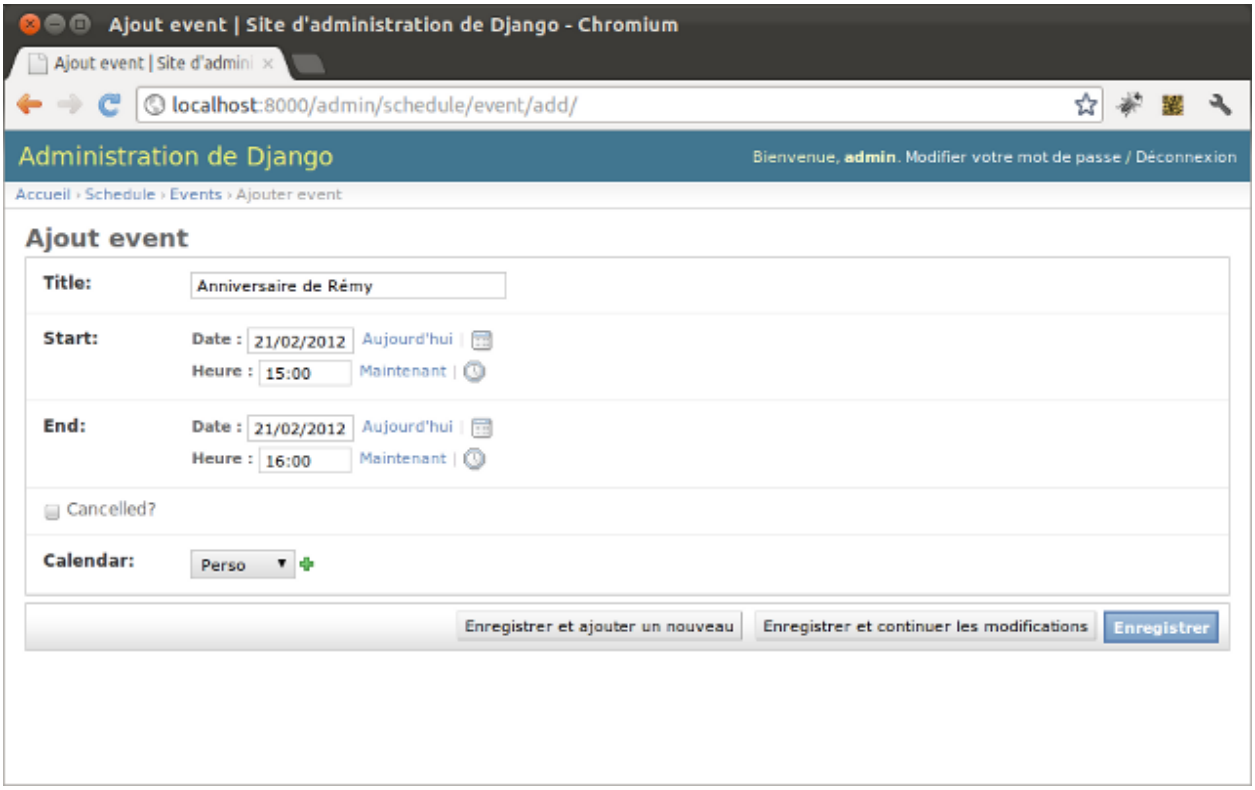
Activation

On va ensuite activer l'administration.

admin.py

Maintenant nous pouvons créer des calendriers et y lier des événements.





Configuration

L'administration Django peut être configurée au petit oignons.

Par exemple, nous souhaitons que le slug se remplisse automatiquement à partir du nom de notre calendrier :

```
class CalendarAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("name",)}

admin.site.register(Calendar, CalendarAdmin)
```

Nous souhaitons aussi pouvoir rechercher dans les événements, afficher la date de début et de fin dans la liste et pouvoir naviguer par date dans les événements :

```
class EventAdmin(admin.ModelAdmin):
    list_display = ('title', 'start', 'end')
    search_fields = ['title']
    date_hierarchy = 'start'

admin.site.register(Event, EventAdmin)
```

Bonus

Vous pouvez améliorer l'interface avec django-grappelli.

Installer grappelli

```
$ pip install django-grappelli
```

Configurer le *settings.py* en ajoutant *grappelli* juste avant *django.contrib.admin* dans le *INSTALLED_APP* :

```
INSTALLED_APPS = (
    ...
    'grappelli',
    'django.contrib.admin',
    ...
)
```

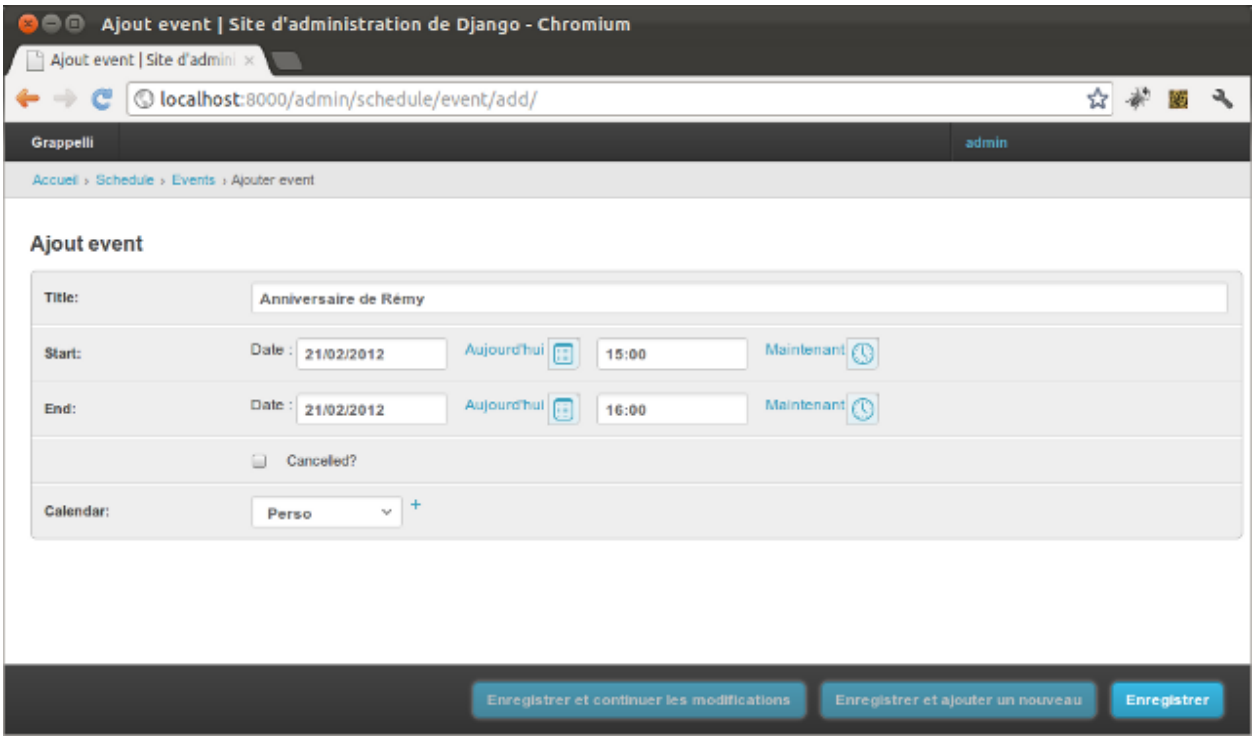
Ajouter les urls vers les views de grappelli et servez les fichiers *static* :

```
urlpatterns = patterns('',
        ...
    url(r'^grappelli/', include('grappelli.urls')),
)

from django.contrib.staticfiles.urls import staticfiles_urlpatterns
urlpatterns += staticfiles_urlpatterns()
```

Juste après la définition de *STATIC_URL* dans le fichier *settings.py* ajoutez :

```
ADMIN_MEDIA_PREFIX = STATIC_URL + "grappelli/"
```

Et voici le résultat :



Conclusion

Nous avons deux modèles, nous avons mis en place South pour nous permettre de continuer à améliorer simplement nos modèles, nous avons configurer l'administration et installer *django-grappelli*.

Nous sommes prêt pour mettre en place les templates.

TP : Agenda - Templates

date 2012-05-01 22:52

tags django, python

category Django

author Rémy Hubscher

Énoncé

Nous souhaitons réaliser un agenda qui nous permettra :

- De gérer plusieurs agenda
- D'intégrer des événements dans l'agenda
- D'ajouter, modifier, supprimer, déplacer un événement

L'IHM

En fait vous allez voir que ce qu'il y a de plus dur dans un agenda web, c'est l'IHM.

Ce n'est pas le but de ce TP, mais à quoi bon avoir un code basé sur Django si au niveau du frontend, la qualité ne suit pas ?

Comme IHM, je vous propose d'utiliser FullCalendar.

Installation des static

On va extraire fullcalendar dans le répertoire *static/schedule/fullcalendar* de notre app *schedule*.

```
schedule/
- static
|   - schedule
|       - fullcalendar
|       |   - fullcalendar.css
|       |   - fullcalendar.js
|       |   - fullcalendar.min.js
|       |   - fullcalendar.print.css
|       |   - gcal.js
|       - js
|           - jquery-1.7.1.min.js
|           - jquery-ui-1.8.17.custom.min.js
- ...
- views.py
```

Mise en place du template

On va récupèrer le template *agenda-views.html* et le configurer pour *Django*

```
schedule/
- templates
|   - schedule
|       - base.html
- ...
- views.py
```

base.html

```
{% load static from staticfiles %}
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
↪DTD/xhtml1-strict.dtd">
<html>
<head>
<link rel='stylesheet' type='text/css' href='{% static 'schedule/fullcalendar/
↪fullcalendar.css' %}' />
<link rel='stylesheet' type='text/css' href='{% static 'schedule/fullcalendar/
↪fullcalendar.print.css' %}' media='print' />
<script type='text/javascript' src='{% static 'schedule/js/jquery-1.7.1.min.js' %}'></
↪script>
<script type='text/javascript' src='{% static 'schedule/js/jquery-ui-1.8.17.custom.
↪min.js' %}'></script>
<script type='text/javascript' src='{% static 'schedule/fullcalendar/fullcalendar.min.
↪js' %}'></script>
<script type='text/javascript'>

    $(document).ready(function() {

            var date = new Date();
            var d = date.getDate();
            var m = date.getMonth();
            var y = date.getFullYear();

            $('#calendar').fullCalendar({
                    header: {
                            left: 'prev,next today',
                            center: 'title',
                            right: 'month,agendaWeek,agendaDay'
                    },
                    editable: true,
                    events: [
                            {
                                    title: 'All Day Event',
                                    start: new Date(y, m, 1)
                            },
                            {
                                    title: 'Long Event',
                                    start: new Date(y, m, d-5),
                                    end: new Date(y, m, d-2)
                            },
                            {
                                    id: 999,
                                    title: 'Repeating Event',
                                    start: new Date(y, m, d-3, 16, 0),
                                    allDay: false
                            },
                            {
                                    id: 999,
```

```
                                        title: 'Repeating Event',
                                        start: new Date(y, m, d+4, 16, 0),
                                        allDay: false
                                },
                                {
                                        title: 'Meeting',
                                        start: new Date(y, m, d, 10, 30),
                                        allDay: false
                                },
                                {
                                        title: 'Lunch',
                                        start: new Date(y, m, d, 12, 0),
                                        end: new Date(y, m, d, 14, 0),
                                        allDay: false
                                },
                                {
                                        title: 'Birthday Party',
                                        start: new Date(y, m, d+1, 19, 0),
                                        end: new Date(y, m, d+1, 22, 30),
                                        allDay: false
                                },
                                {
                                        title: 'Click for Google',
                                        start: new Date(y, m, 28),
                                        end: new Date(y, m, 29),
                                        url: 'http://google.com/'
                                }
                        ]
                });

        });

</script>
<style type='text/css'>

        body {
                margin-top: 40px;
                text-align: center;
                font-size: 14px;
                font-family: "Lucida Grande",Helvetica,Arial,Verdana,sans-serif;
                }

        #calendar {
                width: 900px;
                margin: 0 auto;
                }

</style>
</head>
<body>
<div id='calendar'></div>
</body>
</html>
```

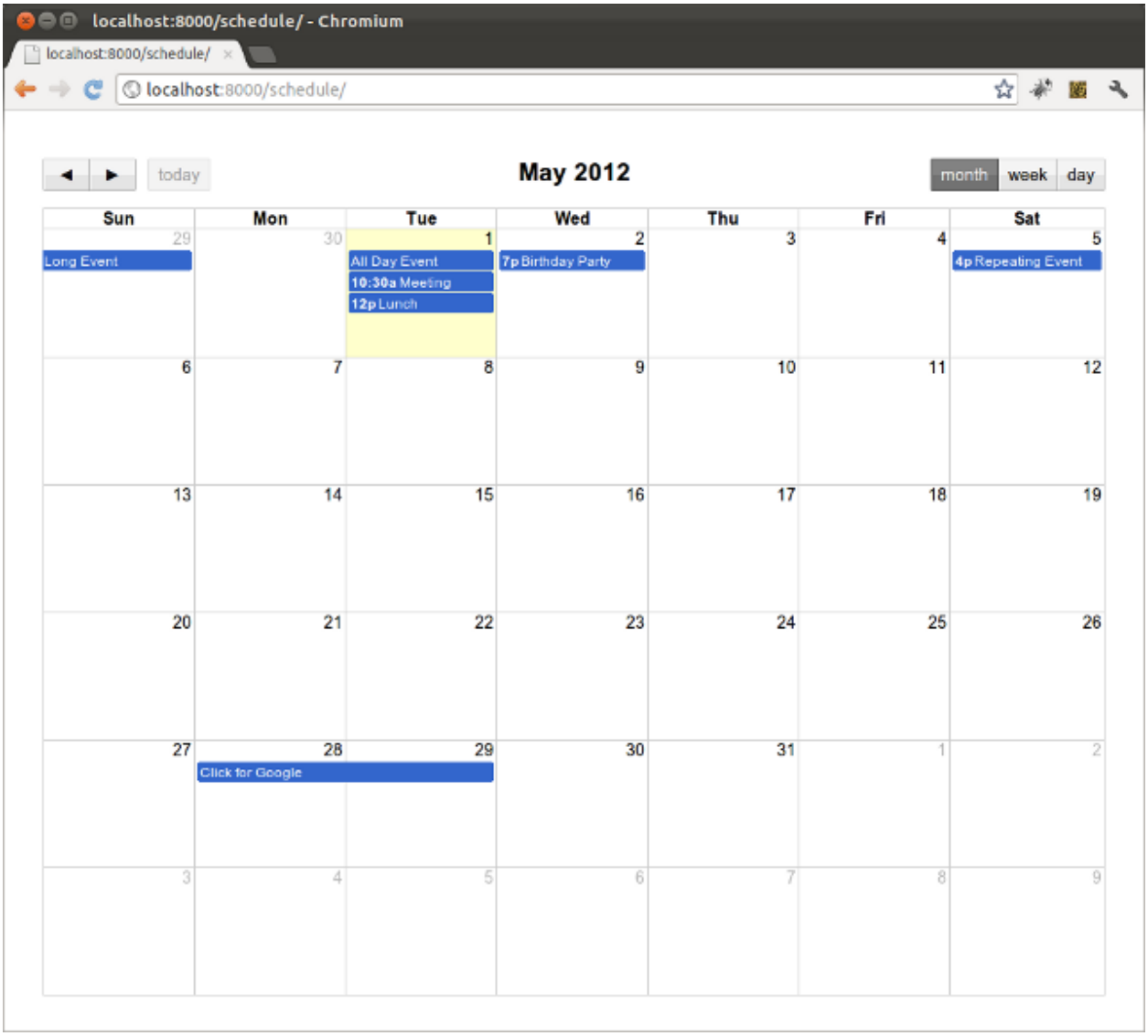
Vérification du template avec *TemplateView*

On créé le fichier d'URLs :

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url
from django.views.generic import TemplateView

urlpatterns = patterns('',
    url(r'^$', TemplateView.as_view(template_name="schedule/base.html"), name=
↪'schedule'),
)
```

On le lie au fichier principal :

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url
from django.views.generic import RedirectView
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^$', RedirectView.as_view(url='todo/')),
    url(r'^grappelli/', include('grappelli.urls')),
    url(r'^admin/', include(admin.site.urls)),
    url(r'^todo/', include('todo.urls')),
    url(r'^schedule/', include('schedule.urls')),
)

from django.contrib.staticfiles.urls import staticfiles_urlpatterns
urlpatterns += staticfiles_urlpatterns()
```



TP : Agenda - Views

date 2012-05-01 23:26

tags django, python

category Django

author Rémy Hubscher

Énoncé

Nous souhaitons réaliser un agenda qui nous permettra :

- De gérer plusieurs agenda
- D'intégrer des événements dans l'agenda

- D'ajouter, modifier, supprimer, déplacer un événement

Afficher nos événements

FullCalendar, peut s'occuper seul d'afficher les informations si on lui envoi un flux JSON.

On va donc faire une vue qui retourne nos events sous forme de JSON.

```
# -*- coding: utf-8 -*-
from django import http
from django.utils import simplejson as json
from django.utils import timezone

from schedule.models import Event

def events_json(request):
    # Get all events - Pas encore terminé
    events = Event.objects.all()

    # Create the fullcalendar json events list
    event_list = []

    for event in events:
        # On récupère les dates dans le bon fuseau horaire
        event_start = event.start.astimezone(timezone.get_default_timezone())
        event_end = event.end.astimezone(timezone.get_default_timezone())

        # On décide que si l'événement commence à minuit c'est un
        # événement sur la journée
        if event_start.hour == 0 and event_start.minute == 0:
            allDay = True
        else:
            allDay = False

        if not event.is_cancelled:
            event_list.append({
                    'id': event.id,
                    'start': event_start.strftime('%Y-%m-%d %H:%M:%S'),
                    'end': event_end.strftime('%Y-%m-%d %H:%M:%S'),
                    'title': event.title,
                    'allDay': allDay
                    })

    if len(event_list) == 0:
        raise http.Http404
    else:
        return http.HttpResponse(json.dumps(event_list),
                                 content_type='application/json')
```

On créé l'URL :

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, include, url
from django.views.generic import TemplateView

urlpatterns = patterns('',
    url(r'^$', TemplateView.as_view(template_name="schedule/base.html"), name=
↪'schedule'),
```

```
    url(r'^events.json$', 'schedule.views.events_json', name='events.json'),
)
```

On configure *fullcalendar* pour qu'il utilise le flux JSON

```
<script type='text/javascript'>

    $(document).ready(function() {
        $('#calendar').fullCalendar({
            header: {
                left: 'prev,next today',
                center: 'title',
                right: 'month,agendaWeek,agendaDay'
            },
            editable: true,
            events: '/schedule/events.json'
        });
    });

</script>
```

Bonus

Si maintenant on souhaite afficher les dates et boutons de fullcalendar en Français

```
$(document).ready(function() {
        $('#calendar').fullCalendar({
            header: {
                left: 'prev,next today',
                center: 'title',
                right: 'month,agendaWeek,agendaDay'
            },
            editable: true,
            monthNames: ['Janvier','Février','Mars','Avril','Mai','Juin','Juillet',
                         'Août','Septembre','Octobre','Novembre','Décembre'],
            monthNamesShort:
            ['Janv.','Févr.','Mars','Avr.','Mai','Juin','Juil.','Août','Sept.','Oct.',
↪'Nov.','Déc.'],
            dayNames: ['Dimanche','Lundi','Mardi','Mercredi','Jeudi','Vendredi',
↪'Samedi'],
            dayNamesShort: ['Dim', 'Lun', 'Mar', 'Mer', 'Jeu', 'Ven', 'Sam'],
            titleFormat: {
                 month: 'MMMM yyyy', // ex : Janvier 2010
                 week: "d[ MMMM][ yyyy]{ - d MMMM yyyy}", // ex : 10 -- 16 Janvier␣
↪2010, semaine à cheval : 28 Décembre 2009 - 3 Janvier 2010
                // todo : ajouter le numéro de la semaine
                 day: 'dddd d MMMM yyyy' // ex : Jeudi 14 Janvier 2010
            },
            columnFormat: {
                 month: 'ddd', // Ven.
                 week: 'ddd d', // Ven. 15
                day: '' // affichage déja complet au niveau du 'titleFormat'
            },
            axisFormat: 'H:mm', // la demande de ferdinand.amoi : 15:00 (pour 15,␣
↪simplement supprimer le ':mm'
            timeFormat: {
```

```
                '': 'H:mm', // événements vue mensuelle.
                agenda: 'H:mm{ - H:mm}' // événements vue agenda
            },
            firstDay:1, // Lundi premier jour de la semaine
            buttonText: {
                prev:     ' &#9668; ',  // left triangle
                next:     ' &#9658; ',  // right triangle
                prevYear: ' &lt;&lt; ', // <<
                nextYear: ' &gt;&gt; ', // >>
                today:    'Aujourd\'hui',
                month:    'Mois',
                week:     'Semaine',
                day:      'Jour'
            },
            events: '/schedule/events.json'
        });
    });
```

Clic sur un événement pour le modifier

Il suffit d'ajouter ceci pour pouvoir le modifier dans l'admin

```
eventClick: function(calEvent, jsEvent, view) {
    document.location.href = '/admin/schedule/event/'+ calEvent.id + '/';
},
dayClick: function(date, allDay, jsEvent, view) {
    document.location.href = '/admin/schedule/event/add/';
}
```

Bien évidement, c'est un peu trop simpliste.

Nous allons donc voir dans l'article suivant comment utiliser les forms pour intégrer cette fonctionnalité à notre app.

Conclusion

Nous avons maintenant une view pour afficher notre agenda, en cliquant sur un événement nous pouvons le modifier en cliquant sur un case vide en ajouter un.

Vous pouvez le tester ici : http://django-story.ionyse.com/demos/schedule/ - (login/pwd : demo)