Cómo crear un servicio REST en 30 líneas de código de Django y Python

Por Alberto Casero.
Publicado en el espacio Desarrollo para Startups en Genbetadev el 11-02-2015
http://www.genbetadev.com/desarrolloparastartups/como-crear-un-servicio-rest-en-30-lineas-de-codigo-de-django-y-python

Lo primero que necesitamos, es instalar Django y REST Framework en nuestro sistema para poder empezar a trabajar. Para ello usamos pip (gestor de paquetes de Python) desde nuestra consola:

```
$ pip install django djangorestframework
```

Una vez instaladas nuestras armas, vamos a crear un projecto Django:

```
$ django-admin.py startproject files_manager
```

Esto nos creará una carpeta `files_manager` en cuyo interior tendrá otra carpeta `files_manager` con archivos `py` que actúan como archivos de configuración del proyecto.

La filosofía de Django es DRY (Don't Repeat Yourself), así que el propio framework nos obliga a crear aplicaciones para que podamos reutilizarlas. Así que en nuestro proyecto `files_manager` vamos a crear una aplicación llamada `files` .

Para ello, accedemos a la carpeta del proyecto y creamos nuestra aplicación:

```
$ cd files_manager
files_manager$ python manage.py startapp files
```

Esto nos creará una carpeta `files` con archivos `py` de nuestra aplicación.

Como otros tantos frameworks, Django usa el patrón MVC (aunque un poco de aquella manera: los *controladores* los escribimos en un archivo `views.py`).

Vamos a dejarnos ya de tanta consola y a tirar algo de código. Abrimos el archivo `models.py` de la carpeta `files` y escribimos los siguiente:

```
# models.py
from django.db import models
from django.conf import settings

UPLOADS_DIR = getattr(settings, "UPLOADS_DIR", "uploads")


class File(models.Model):
    file = models.FileField(upload_to=UPLOADS_DIR)
    created_on = models.DateTimeField(auto_now_add=True)
    modified_on = models.DateTimeField(auto_now_add=True, a
uto_now=True)
```

Lo que hemos hecho es crear un modelo `File` que tiene tres campos:

- **file**: que almacenará la ruta al archivo que subamos.
- **created_on**: campo que almacenará la hora de creación del archivo.
- **modified_on**: campo que almacenará la hora de última modificación del archivo.

Bien, sin tener que escribir nada más, Django se encargará pon nosotros de:

- Gestionar si la carpeta donde vayamos a subir los archivos existe o no (de no existir la creará).
- Controlar si ya existe un archivo con el mismo nombre que el archivo que subimos para renombrarlo.
- Asignar una fecha de creación del archivo automáticamente.
- Asignar una fecha de modificación del archivo cada vez se modifique (también de manera automática).

Lo siguiente que tenemos que hacer es crear un serializador, el cual se encargará de actuar como traductor entre nuestro modelo y los datos que nos envíen a través de las peticiones HTTP de nuestro API Rest.

Creamos un archivo `serializers.py` en la carpeta `files` con el siguiente contenido:

```
# serializers.py
from models import File
from rest_framework.serializers import ModelSerializer


class FileSerializer(ModelSerializer):
    class Meta:
        model = File
```

Listo! Como podéis ver, tan sólo tenemos que decirle al `ModelSerializer` cual es el modelo que debe representar y él se encarga de todo.

Lo siguiente es implementar nuestra API, así que antes vamos a pensar qué queremos que haga:

- En la URL /files/ quiero obtener un listado de los archivos que hay en mi sistema si hago una petición GET. Si hago una petición POST, entonces subiremos un nuevo archivo.
- En la URL /files/<fileID/ quiero poder obtener el detalle de un archivo si hago una petición GET, mientras que si hago una petición PUT actualizaremos el archivo y si hago una petición DELETE lo eliminaremos.

Bien entonces vamos a abrir nuestro archivo `views.py` de la carpeta

`files` para escribir lo siguiente:

```
# views.py
from models import File
from serializers import FileSerializer
from rest_framework.viewsets import ModelViewSet


class FileViewSet(ModelViewSet):
    queryset = File.objects.all()
    serializer_class = FileSerializer
```

De nuevo, con tan sólo darle un poco de información a la clase que creamos es suficiente: le decimos de dónde debe sacar la información (*queryset*) y qué debe de utilizar como traductor (*serializer_class*). Del resto se encargan Django y REST Framework.

Bien, casi hemos llegado al final. Ahora tenemos que conectar de algún modo este ViewSet con las URLs del API Rest que hablábamos antes. Para ello, utilizaremos un `Router`.

Abrimos el archivo `urls.py` esta vez de la carpeta `files_manager` añadimos 5 líneas:

```
# urls.py
from files.views import FileViewSet
from rest_framework.routers import SimpleRouter
from django.conf.urls import patterns, include, url
from django.contrib import admin

router = SimpleRouter()
router.register(r'files', FileViewSet)

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^', include(router.urls)),
)
```

Importamos nuestro `FileViewSet` y el `SimpleRouter` (líneas 2 y 3):

```
# urls.py
from views import FileViewSet
from rest_framework.routers import SimpleRouter
[...]
```

Registramos el FileViewSet en el router (líneas 7 y 8):

```
[...]
router = SimpleRouter()
router.register(r'files', FileViewSet)
[...]
```

Y añadimos las URLs que el router genera por nosotros a los patrones de URL de nuestro proyecto (línea 12):

```
[...]
urlpatterns = patterns('',
    [...]
    url(r'^', include(router.urls)),
)
```

Bien, penúltimo paso: registrar nuestra aplicación y REST Framework en el proyecto. Para ello en el archivo `settings.py` de la carpeta `files_manager` añadimos la siguiente línea a la tupla `INSTALLED_APPS`:

```
[...]
# Application definition

INSTALLED_APPS = (
    [...]
    'rest_framework`,
    'files',
)
[...]
```

Bien, este es todo el código que necesitamos! Ahora vamos a probarlo.

Primero tenemos que crear los archivos de migración para posteriormente crear nuestro esquema de base de datos (por defecto, Django utiliza SQLite

salvo que le indiquemos lo contraro, ideal para desarrollar sin tener que instalar software adicional).

Volvemos a nuestra consola y ejecutamos:

```
files_manager$ python manage.py makemigrations
```

Esto creará un archivo `0001_initial.py` en la carpeta `migrations` de nuestra carpeta `files`. Estas migraciones se crean cada vez que hay un cambio en los modelos (cambios en estructuras de tablas SQL) o cuando los creamos por primera vez. Django hace esto para evitar el tener que andar con scripts SQL de migración.

Con los scripts de migración creados, tenemos que aplicar la migración:

```
files_manager$ python manage.py migrate
```

Muy bien, ya estamos listos para probar nuestro API. Vamos a arrancar el **servidor HTTP de desarrollo** integrado en Django desde la consola:

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

Si ahora abrimos en nuestro navegador la URL: http://127.0.0.1:8000/files/ deberíamos ver el API navegable funcionando.

¿API navegable? Sí, REST Framework proporciona un API navegable que nos permite probar rápidamente nuestros APIs sin necesidad de utilizar clientes REST!

Para probar la subida de archivos con un cliente REST, deberemos poner la cabecera `Content-Type: multipart/form-data` para el envío de la petición POST (porque los archivos no se suben en JSON!).