Laboratorio de sistemas operativos y redes

Django

Tutorial de instalación y uso

Muñoz, Néstor Gabriel
Yegro, Juan Ignacio

Introducción

¿Qué es Django?

Django es un framework web open source, desarrollado en Python, que permite construir aplicaciones web más rápido y con menos código. Este framework sigue el patrón de arquitectura MVC(Modelo-Vista-Controlador).
Django fue inicialmente desarrollado para gestionar aplicaciones web de páginas orientadas a noticias de World Online, más tarde se liberó bajo licencia BSD. Django se centra en automatizar todo lo posible y se adhiere al principio DRY (don’t repeat yourself).

Instalación y primeros pasos en Django[1]

Instalación

Antes que nada hay que instalar python, las últimas versiones de Linux ya lo tienen instalado. La instalación común de Django es muy simple. Hay que descargarse los archivos comprimidos de la página web, descomprimirlos, y desde la carpeta descomprimida ejecutar:

```
$ sudo python setup.py install
```

Para corroborar que se instaló correctamente:

```
$ python
>>> import django
>>> django.VERSION
```

Pero aquí vamos a explicar una instalación un poco más sofisticada con pip y virtualenv.

Creación de un entorno de desarrollo con pip y virtualenv[2]

La idea de un entorno de desarrollo virtual es que las instalaciones que hagamos en él no afecten a las que ya están en nuestro sistema y que las instalaciones futuras tampoco corrompan lo que tenemos funcionando dentro de nuestro entorno.

Pip es un instalador de paquetes de python alternativo. Lo instalamos:

```
$ sudo apt-get install python-pip
```

Con pip instalamos virtualenv

```
$ pip install virtualenv
```

Creamos un directorio para nuestro entorno. Dentro creamos dos directorios: src y env. En src pondremos el código fuente de nuestra aplicación y en env estará nuestro entorno. Para crear el entorno, desde el primer directorio que creamos, ejecutamos:

```
$ sudo virtualenv --distribute --no-site-packages env/
```

Luego instalamos Django con pip:

[1] Para obtener información más detallada visitar http://www.djangobook.com

[2] Para hacer esta sección nos basamos en el artículo http://www.userlinux.net/django-virtualenv-pip.html visitado el 11 de diciembre de 2012

```
$ sudo pip -E env/ install django
```

-E significa que estamos instalando algo en el entorno desde afuera.

Yolk

Yolk es una libreria y una herramienta de línea de comandos para listar paquetes instalados con sus dependencias y metadata. Para instalarlo ejecutar:

```
$ pip -E env/ install yolk
$ source env/bin/activate
$ yolk -l
```

Debug toolbar

Características:

- Navaja suiza de django
- Información sobre uso de la DB
- Información sobre rendering
- Información de HTTP
- Información de ruteo de URLs

Para instalarla ejecutar:

```
$ pip -E env/ install django-debug-toolbar
```

Mysql

```
$ sudo apt-get install python-mysqldb
$ sudo apt-get install libmysqld-dev
```

Ingresar y salir del entorno de desarrollo

Para ingresar al entorno:

```
$ source env/bin/activate
```

Para salir del entorno:

```
deactivate
```

Replicar el entorno

Para replicar el entorno en otra máquina hay que pasarle el archivo requirements.txt. Para crearlo ejecutar desde dentro del entorno:

```
pip freeze > requirements.txt
```

En la otra máquina instalar pip y virtualenv, crear un entorno nuevo y ejecutar desde afuera:

```
$ pip install -E env/ -r bar/src/requirements.txt
```

Para comprobar que todo se instaló correctamente ingresar al entorno y ejecutar:

```
(env)$ pip freeze
```

Empezando un proyecto nuevo

Para crear un proyecto nuevo llamado mysite ejecutar:

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

Se creará la siguiente estructura:

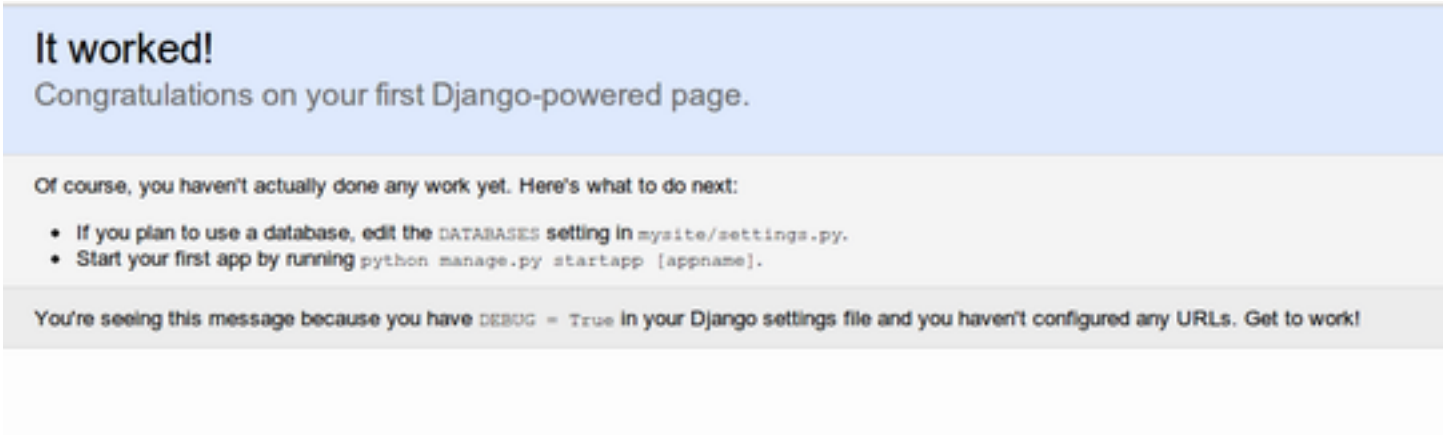
```
mysite/
  __init__.py
  manage.py
  settings.py
  urls.py
```

__init__.py: es un archivo requerido por Python para tratar el directorio mysite como un paquete (es decir un grupo de modulos Python). Es un archivo vacio y generalmente no se le agrega nada.
manage.py: una herramienta de linea de comandos que nos permite interactuar con Django de varias maneras. Tipear *python manage.py help* para tener una idea de lo que puede hacer. Nunca se debe editar este archivo.
settings.py: configuraciones del proyecto.
urls.py: las urls del proyecto. Sería como una tabla de contenidos del sitio web.

Para corroborar que todo funciona ejecutar:

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

Abrir el navegador e ir hacia: http://localhost:8000/



Vistas

Las vistas se ubican dentro del directorio templates. Hay que crearlo. Para indicar a Django donde tiene que ir a buscar las vistas editar el archivo settings.py:
En settings.py buscar TEMPLATE_DIRS y agregar el path de la carpeta templates que creamos, deberia quedar algo así:

```
TEMPLATE_DIRS = (
        '/path/hacia/la/carpeta/templates',
)
```

Ahora cada vez que creamos una vista la ponemos dentro de la carpeta templates. Por ejemplo si queremos crear una página hola.html, la creamos y la ponemos en la carpeta templates.
Luego en el archivo views.py agregamos la función que se va a llamar cuando vayamos a hola.html:

```
from django.http import HttpResponse
from django.template.loader import get_template
from django.template import Template, Context

def hola(request):
        t = get_template('hola.html')
        html = t.render(Context())
        return HttpResponse(html)
```

una función equivalente a la anterior es la siguiente:

```
from django.shortcuts import render_to_response

def hola(request):
        return render_to_response('hola.html')
```

render_to_response() es un shortcut para hacer get_template() y render(Context()) en un solo paso. No hay que olvidar importarlo.

Luego agregamos el mapeo en urls.py:

```
from django.conf.urls.defaults import *
from mysite.views import hola

urlpatterns = patterns('',
        ('^hola/$', hola),
)
```

En http://localhost:8000/hola/ deberiamos poder ver nuestra nueva página

Páginas con contenido dinámico

Si tenemos una página a la que queremos pasarle valores por parámetro lo podemos hacer de la siguiente manera. Supongamos que esta es nuestra página:

```
<html>
<head> pagina </head>
<body>
        <p>Mi nombre es {{ nombre }}</p>
</body>
</html>
```

Naturalmente tenemos que definir una función en views.py para esa página y darle un valor a la variable {{ nombre }}:

```
from django.shortcuts import render_to_response

def pagina(request)
        return render_to_response('pagina.html', {'nombre': 'Nestor'} )
```

Así le damos el valor 'Nestor' a la variable 'nombre'.

Forms

Para crear un form para que las personas puedan registrarse en nuestro sitio web, por ejemplo, podemos hacer lo siguiente:
1) crear nuestro html y ponerlo dentro de la carpeta templates.

```
<html>
<head><title>registro</title></head>
<body>
        <h1>Registrate</h1>
        <form action="/registrado/" method="get">
                Nombre: <input type="text" name="nombre"><br>
                Mail: <input type="text" name="mail"><br>
                Contraseña: <input type="password" name="pass">
        <input type="submit" value="Registrarme">
        </form>
</body>
</html>
```

2) Como siempre, agregar la función que se corresponda con el nombre del html en views.py y agregar el mapeo en urls.py
3) Para obtener la información que se submiteó en el form utilizamos request.GET que devuelve un diccionario donde las claves son los campos del form. Por ejemplo para obtener lo que se submiteó en el campo 'nombre' hacemos: request.GET['nombre']

Validaciones

Para validar, por ejemplo, que el campo 'nombre' no esté vacío podemos hacer lo siguiente en la función correspondiente a la página siguiente al form:

```
def registrado(request):
        r = request.GET
        if not r.get('nombre') :
                return render_to_response('registro.html',{'error':True})
        else:
                return render_to_response('registrado.html')
```

Este código dice que si el campo 'nombre' está vacío entonces se vuelve a mostrar la página del form con 'error' en True. Para mostrar un mensaje de error el html del form nos debería quedar algo por el estilo:

```
<html>
```

```
<head><title>registro</title></head>
<body>
        <h1>Registrate</h1>
        <form action="/registrado/" method="get">
                Nombre: <input type="text" name="nombre"><br>
                Mail: <input type="text" name="mail"><br>
                Contraseña: <input type="password" name="pass">
        <input type="submit" value="Registrarme">
        </form>
        {% if error %}
                <p style="color: red;">Debe completar todos los campos</p>
        {% endif %}
</body>
</html>
```

Persistiendo en una Base de Datos

Usando MYSQL

Se crea una BD llamada “registro”

```
mysql> create database registro;
```

Query OK, 1 row affected (0.00 sec)

La seleccionamos

```
mysql> use registro;
```

Database changed

Luego se pasa a configurar esa BD en Django

En settings.py

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.mysql'
      'NAME': 'registro',                  # O el path a la base de datos si se usa sqlite3
      'USER': 'root',                  # No usado con sqlite3
'PASSWORD': 'nestorvive',                  # No usado con sqlite3
'HOST': '',                  # Dejarlo vacio para localhost.
'PORT': '',                  # Dejarlo vacio para configuracion por defecto
   }
}
```

Creando un modelo

```
python manage.py startapp registro
```

Esto no produce ningún output, pero crea un directorio llamado “registro” con el siguiente contenido:

registro/

```
  __init__.py
  models.py
  tests.py
  views.py
```

En models.py

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

```
class Usuario(models.Model):
    nombre = models.CharField(max_length=30)
    mail = models.CharField(max_length=50)
    contrasenia = models.CharField(max_length=16)
```

Cada modelo corresponde con una tabla de la base de datos, y cada atributo del modelo corresponde a una columna de esa tabla. El nombre del atributo corresponde con el nombre de la columna, y el tipo de dato (por ejemplo CharField) corresponde con el tipo de columna (varchar). Por ejemplo, el modelo Usuario es **equivalente** al siguiente código sql:

```
CREATE TABLE "registro_usuario" (
    "id" serial NOT NULL PRIMARY KEY,
    "nombre" varchar(30) NOT NULL,
    "mail" varchar(50) NOT NULL,
    "contrasenia" varchar(16) NOT NULL,
);
```

Instalando el modelo

Hay que agregar nuestro modelo a settings.py, dirigiéndonos a la parte de INSTALLED_APPS

```
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',
    'registro'
)
```

Primero hay que validar que la sintaxis y la lógica del modelo es correcta:

```
python manage.py validate
```

Si todo salió bien, aparecerá el mensaje `0 errors found`. En caso contrario, deberás arreglar el código.

Luego, verificar el código sql generado:

```
 manage.py sqlall registro
BEGIN;
CREATE TABLE `registro_usuario` (
    `id` integer AUTO_INCREMENT NOT NULL PRIMARY KEY,
    `nombre` varchar(30) NOT NULL,
    `mail` varchar(50) NOT NULL,
    `contrasenia` varchar(16) NOT NULL
)
;
COMMIT;
```

El comando `sqlall` no crea las tablas ni toca la base de datos, solo imprime en pantalla el código sql que luego Django ejecutará si se lo piden.

Para ésto se usa el comando `syncdb`

```
python manage.py syncdb
```

Ejecuta el comando y vas a ver algo como ésto:

```
Creating table registro_usuarios
Installing index for registro.Usuario model
```

Acceso a datos

Una vez creado el modelo, Django provee una API de alto nivel para trabajar con esos modelos.

Probá ejecutando `python manage.py shell` y luego lo siguiente:

```
>>> from registro.models import Usuario
>>> usuario1 = Usuario(nombre='Jorge', mail='jorge@gmail.com', contrasenia='1234')
>>> usuario1.save()
>>> usuario2 = Usuario(nombre='Braulio', mail='braulito_de_bera@gmail.com', contrasenia='contrasenia')
>>> usuario2.save()
>>> lista_usuarios = Usuario.objects.all()
>>> lista_usuarios
[<Usuario: Usuario object>, <Usuario: Usuario object>]
```