AWAKELAB

Desarrollo Web Python con Django

Desarrollador de aplicaciones Full Stack

Python Trainee

6.2.- Contenido 2: Utilizar las herramientas administrativas provistas por el framework para la configuración de un nuevo proyecto web Django.

Objetivo de la jornada

1. Utiliza el administrador de paquetes PIP para la instalación de los componentes Django.
2. Utiliza el utilitario manage.py para la creación de un nuevo proyecto Django.

INTRODUCCION A DJANGO

¿Qué es Django?

Django es un framework web de alto nivel que permite el desarrollo rápido de sitios web seguros y mantenibles. Desarrollado por programadores experimentados, Django se encarga de gran parte de las complicaciones del desarrollo web, por lo que puedes concentrarte en escribir tu aplicación sin necesidad de reinventar la rueda. Es gratuito y de código abierto, tiene una comunidad próspera y activa, una gran documentación y muchas opciones de soporte gratuito y de pago

¿Por qué usar Django en un mundo donde todo es Javascript?

¿De verdad vale la pena aprender un Framework de Python en un ecosistema que se empecina en Frameworks escritos en Javascript? Pues yo creo que sí y a continuación te expongo algunas de las razones por las que deberías usar Django. Y, para no perder objetividad, te hablaré tanto de las ventajas, como de las desventajas; ya sabes ninguna solución es perfecta.

Las ventajas de Django

Su ORM es sencillo y maravilloso

El ORM de Django abstrae la necesidad de escribir consultas SQL para crear tablas y consultar datos. Es bastante intuitivo de usar y tiene incluidas casi todos las consultas más comunes en su código. Desde filtrados, particionados, uniones e incluso hasta funciones búsquedas avanzadas de Postgres.

Para crear una tabla en la base de datos basta con crear un modelo y Django se encargará de todo el trabajo pesado.

```
class Review(models.Model):
    title = models.CharField(max_length=25)
    comment = models.TextField()
    name = models.CharField(max_length=20)
    created = models.DateTimeField(auto_now_add=True)
    modified = models.DateTimeField(auto_now=True)
    product = models.ForeignKey(
        Product, related_name="reviews", on_delete=models.CASCADE)
    user = models.ForeignKey(
        get_user_model(), related_name="reviews", null=True, on_dele
```

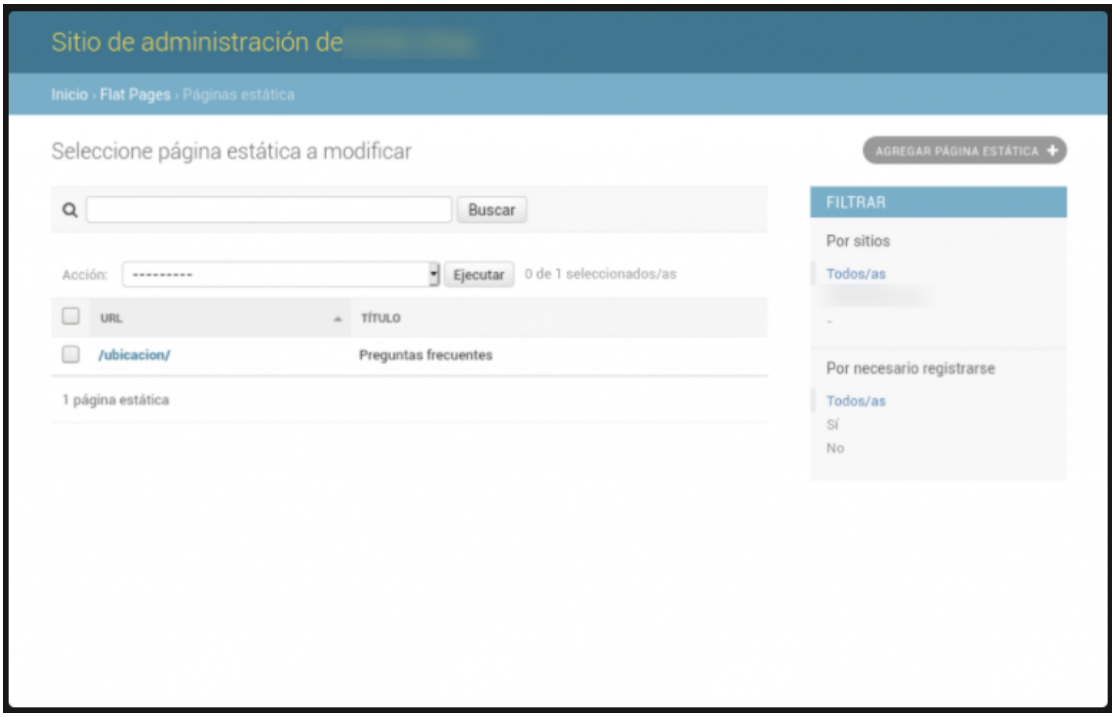
Además de lo anterior, su ORM soporta múltiples bases de datos, por lo que cambiar de motor de base de datos es bastante sencillo y tras unos pocos cambios puedes migrar perfectamente de Postgres a MySQL o viceversa, únicamente cambiando un par de lineas en la configuración.

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'OPTIONS': {
            'read_default_file': '/path/to/my.cnf',
        },
    }
}
```

Su única desventaja es su velocidad, pues se queda corto frente a otras alternativas como sqlAlchemy, o tortoise-orm.

Panel de administrador incluido

Django cuenta con el [django admin] panel, un panel de administración que viene instalado por defecto. Este administrador implementa un CRUD a la base de datos de una manera sencilla. Y, además, cuenta con un sólido sistema de permisos para restringir el acceso a los datos como tu quieras.



Ofrece seguridad ante los ataques más comunes

Django incluye ciertas utilidades, que se encargan de mitigar la mayoría de los ataques tales como XSS, XSRF, injecciones SQL, Clickjacking y otros. La mayoría ya están disponibles y basta con solo agregar el middleware o etiqueta de plantilla correspondiente.

```
<form method="post">{% csrf_token %}
```

Sistema de permisos

Django cuenta con un sólido sistema de permisos y grupos que vincula a sus usuarios con modelos en la base de datos que puedes empezar a usar solo con unas cuantas lineas de código.

Múltiples paquetes

Django cuenta con muchísimos paquetes para resolver la mayoría de los problemas comunes, además son paquetes supervisados y mejorados por la comunidad logrando una calidad impresionante.
Solo por nombrar algunos:

Django-watson (Búsquedas)
DRF (REST)
Graphene (Graphql)
Django-rest-auth (Autenticación)
Django-allauth (Autenticación)
Django-filter (Búsquedas)
Django-storage (AWS storage)
Django-braces (Funciones comunes)

Entre todos ellos me gustaría resaltar DRF (Django Rest Framework) que vuelve la creación de una API, permisos y throttling, una tarea mucho más corta que en otros lenguajes. La autenticación con DRF también es bastante sencilla usando los paquetes anteriores. La librería Graphene también es destacable, permite implementar graphql en Django con pocas lineas de código.

Te lleva de una idea a un prototipo funcional rápido

Yo considero esta la razón principal para usar Django. Django te lleva de una idea a un MVP rápido y sin necesidad de reinventar la rueda. Lo cual es una ventaja competitiva gigantesca frente a otros frameworks, especialmente cuando hay dinero y clientes involucrados. Probablemente con Django tendrías un prototipo funcionando más rápido que con cualquier otro framework «menos opinado».

Es una solución probada.

Hay muchísimos frameworks nuevos cada día. La mayoría de ellos son solo una moda y caen en desuso con el pasar de los años, dejando proyectos sin soporte. Django es un framework que lleva muchísimo tiempo funcionando y que ha pasado por

numerosas pruebas que lo han vuelto muy robusto y confiable. Django fue la opción que alguna vez eligieron sitios tan grandes como Instagram o Pinterest.

Flexibilidad de instalación y configuración

Ahora que sabes para qué se utiliza Django, te enseñaremos cómo configurar y probar un entorno de desarrollo Django en Windows, Linux (Ubuntu), y Mac OS X — cualquiera que sea el sistema operativo común que estés utilizando, este artículo te dará lo que necesita ser capaz de empezar a desarrollar aplicaciones Django.

Django es extremadamente flexible en términos de cómo y dónde puede instalarse y configurarse. Django puede ser:

- instalado en diferentes sistemas operativos.
- ser usado con Python 3 y Python 2.
- instalado desde las fuentes, desde el Python Package Index (PyPi) y en muchos casos desde la aplicación de gestión de paquetes de la computadora.
- Está configurado para usar una de entre varias bases de datos, que también pueden necesitar ser instaladas y configuradas por separado.
- ejecutarse en el entorno Python del sistema principal o dentro de entornos virtuales Python separados.

Cada una de estas opciones requiere configuraciones y puesta en marcha ligeramente diferentes. Las siguientes subsecciones explican algunas de sus opciones. En el resto del artículo te mostraremos como ajustar Django en un pequeño número de sistemas operativos, y se supondrá ese ajuste a lo largo del resto del módulo.

¿Cuál es el entorno de desarrollo de Django?

El entorno de desarrollo es una instalación de Django en tu computadora local que puedes usar para desarrollar y probar apps Django antes de desplegarlas al entorno de producción.

Las principales herramientas que el mismo Django proporcionan son un conjunto de scripts de Python para crear y trabajar con proyectos Django, junto con un simple *servidor web de desarrollo* que puedes usar para probar de forma local (es decir en tu computadora, no en un servidor web externo) aplicaciones web Django con el explorador web de tu computadora.

Hay otras herramientas periféricas, que forman parte del entorno de desarrollo, que no cubriremos aquí. Estas incluyen cosas como un editor de textos o IDE para editar código, una herramienta de gestión del control de fuentes como Git para gestionar con seguridad las diferentes versiones de tu código. Asumimos que tienes ya un editor de textos instalado.

¿Qué sistemas operativos están soportados?

Las aplicaciones web Django pueden ejecutarse en casi cualquier máquina donde pueda funcionar el lenguaje de programación Python: Windows, Mac OS X, Linux/Unix, Solaris, por nombrar sólo unos pocos. Casi cualquier computadora debería tener el rendimiento necesario para ejecutar Django durante el desarrollo.

En este artículo proporcionamos instrucciones para Windows, Mac OS X y Linux/Unix.

¿Qué versión de Python deberías usar?

Django se ejecuta por encima de Python, y puede usarse tanto con Python 2 o con Python 3 (o ambos). Cuando estés seleccionando una versión debería tener en cuenta que:

- Python 2 es una versión tradicional del lenguaje que no va a tener más características nuevas pero que tiene disponible para los desarrolladores, un enorme repositorio de bibliotecas de terceros de alta calidad (algunas de las cuales no están disponibles en Python 3).
- Python 3 es una actualización de Python 2 que, aunque similar, es más consistente y fácil de usar. Python 3 también es el futuro de Python, y continúa su evolución.
- También es posible soportar ambas versiones usando bibliotecas (ej. seis), aunque no sin un esfuerzo adicional de desarrollo.

Nota :
Históricamente Python 2 era la única elección realista, porque muy pocas bibliotecas de terceros estaban disponibles para Python 3. La tendencia actual es que la mayoría de paquetes nuevos y populares del Índice de paquetes de Python(PyPi) soportan ambas versiones de Python. Aunque todavía hay muchos paquetes que sólo están disponibles para Python 2, elegir Python 3 es actualmente una opción muy popular.

Te recomendamos que uses la última versión de Python 3 a menos que el sitio dependa de bibliotecas de terceros que solo están disponibles para Python 2.

Este artículo te explicará cómo instalar un entorno para Python 3 (el ajuste equivalente para Python 2 sería muy similar).

¿Dónde puedo descargarme Django?

Hay tres lugares para descargar Django:

- El Python Package Repository (PyPi), usando la herramienta *pip* . Este es el mejor modo de obtener la última versión estable de Django.
- Usar una versión del gestor de paquetes de tu computadora. Las distribuciones de Django que se empaquetan con los sistemas operativos ofrecen un mecanismo de instalación ya familiar. Ten en cuenta sin embargo que la versión empaquetada puede ser bastante antigua, y solo puede ser instalada en el entorno de Python del sistema (que no puede ser el que tu quieras).
- Instalar desde la fuente. Puedes obtener y descargar la versión con el último grito de Python partiendo de las fuentes. Esto no es lo recomendable para principiantes, pero es necesario cuando estás listo para empezar a contribuir codificando el propio Django.

Este artículo te muestra cómo instalar Django desde PyPi, para conseguir la última versión estable.

¿Qué base de datos?

Django soporta cuatro bases de datos importantes (PostgreSQL, MySQL, Oracle y SQLite), y hay bibliotecas comunitarias que proporcionan varios niveles de soporte para otras bases de datos populares SQL y NOSQL. Te recomendamos que elijas la misma base de datos tanto para la producción como para el desarrollo (aunque Django abstrae muchas de las diferencias entre las bases usando su Object-Relational Mapper (ORM), hay todavía problemas potenciales que es mejor evitar).

Durante este artículo (y la mayoría de este módulo) usaremos la base de datos *SQLite* , que almacena sus datos en un fichero. SQLite está pensado para ser usado como base ligera y no puede soportar un alto nivel de concurrencia. Es sin embargo una excelente elección para aplicaciones que son principalmente de sólo lectura.

Nota : Django está configurado para usar SQLite por defecto cuando comienzas tu proyecto de sitio web usando las herramientas estándar (*django-admin*). Es una gran elección cuando estás comenzando porque no requiere configuración o puesta en marcha adicional.

¿Instalar Python en un entorno de sistema o virtual?

Cuando instalas Python3 obtienes un único entorno global que es compartido con todo el código Python3. Si bien puedes instalar los paquetes que te gustan en el entorno, solo puedes instalar al mismo tiempo una versión en particular de cada paquete.

Nota : Las aplicaciones Python instaladas en el entorno global pueden entrar en conflicto potencialmente unas con otras (ej. si depende de diferentes versiones del mismo paquete).

Si instala Django dentro del entorno por defecto/global sólo podrá apuntar a una sola versión de Django en la computadora. Esto puede ser un problema si quieres crear nuevos sitios (usando la última versión de Django) pero manteniendo los sitios web que dependen de versiones más antiguas.

Como resultado, los desarrolladores experimentados de Python/Django normalmente ejecutan las aplicaciones Python dentro de *entornos virtuales* independientes de Python. De esta forma se habilitan múltiples entornos Django diferentes en la misma computadora. !El mismo equipo de desarrollo Django recomienda que use entornos virtuales Python!

Este módulo da por supuesto que ha instalado Django en un entorno virtual, y te mostraremos cómo hacerlo más abajo.

Instalación de Python 3

Para poder usar Django tendrás que instalar Python en tu sistema operativo. Si estás usando *Python 3* , utiliza la herramienta Índice de paquetes de Python — *pip3* — que se usa para gestionar (instalar, actualizar y eliminar) los paquetes/bibliotecas Python usados por Django y tus otras aplicaciones Python.

Esta sección explica quizás cómo puede comprobar qué versiones de Python están presentes, e instalar nuevas versiones cuando lo necesite, en Ubuntu Linux 16.04, Mac OS X y Windows 10.

Nota : Dependiendo de tu plataforma, podrías también ser capaz de instalar Python/pip desde la propia aplicación de gestión de paquetes de tu sistema o vía otros mecanismos. Para la mayoría de las plataformas puedes descargar los ficheros de instalación necesarios

desde https://www.python.org/descargas/ e instalarlos usando el método específico apropiado de la plataforma

Linux

Ubuntu Linux incluye Python 3 por defecto. Puedes confirmarlo con el siguiente comando en una terminal:

```
~$ python –version

Python 3.9
```

Sin embargo, la herramienta Python Package Index que necesitará para instalar paquetes de Python 3 (incluido Django) **no** está disponible por defecto. Puedes instalar pip3 en un terminal bash usando:

```
sudo apt-get install python3-pip
```

Mac OS X

Mac OS X "El Capitan" no incluye Python 3. Puedes confirmarlo logrado los siguientes comandos en un terminal bash:

```
python3 -V
 -bash: python3: command not found
```

Puedes instalar fácilmente Python 3 (junto con la herramienta *pip3*) desdepython.org:

1. Descarga el instalador requerido:
    1. Vete a https://www.python.org/descargas/
    2. Selecciona el botón **Descarga Python 3.9** (el número exacto de versión menor puede ser diferente).
2. Localiza el fichero usando *Finder* , haz doble clic sobre el fichero del paquete. Pincha siguiente en las ventanas de instalación.

Puede confirmar ahora una instalación satisfactoria comprobando *Python 3* como se muestra a continuación:

```
~$ python –version

Python 3.9
```

Puedes comprobar igualmente que *pip3* está instalado listando los paquetes disponibles:

```
pip3 list
```

Uso de Django dentro de un entorno virtual de Python

Las bibliotecas que usaremos para crear nuestros entornos virtuales estan en envoltorio virtual (Linux y Mac OS X) y virtualenvwrapper-win (Windows), que utilice a su vez la herramienta virtualenv . Las herramientas wrapper crean una interfaz consistente para la gestión de interfaces en todas las plataformas.

Instalación del software del entorno virtual

Puesta en marcha del entorno virtual en Ubuntu

Después de instalar Python y pip puedes instalar *virtualenvwrapper* (que incluye *virtualenv*) usando *pip3* como se muestra.

```
sudo pip3 install virtualenvwrapper
```

A continuación se añaden las líneas siguientes al final del fichero de inicio de tu shell (éste es un fichero oculto llamado **.bashrc** que se encuentra en tu directorio de inicio del usuario). Ésto ajusta la localización de dónde puede vivir los entornos virtuales, la localización de los directorios de tus proyectos de desarrollo, y la localización del script instalado con este paquete:

```
export WORKON_HOME=$HOME/.virtualenvs
```

```
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3

export PROJECT_HOME=$HOME/Devel

source /usr/local/bin/virtualenvwrapper.sh
```

A continuación a volver a recargar el fichero de inicio producido el siguiente comando en el terminal:

```
source ~/.bashrc
```

En este punto deberías ver un puñado de scripts comenzando a ejecutarse como se muestra a continuación:

```
virtualenvwrapper.user_scripts creating
/home/ubuntu/.virtualenvs/premkproject

virtualenvwrapper.user_scripts creating
/home/ubuntu/.virtualenvs/postmkproject

...

virtualenvwrapper.user_scripts creating
/home/ubuntu/.virtualenvs/preactivate

virtualenvwrapper.user_scripts creating
/home/ubuntu/.virtualenvs/postactivate

virtualenvwrapper.user_scripts creating
/home/ubuntu/.virtualenvs/get_env_details
```

Uso de un entorno virtual

Hay sólo otros pocos comandos útiles que deberías conocer (hay más en la documentación de la herramienta, pero estos son los que usarás de forma habitual:

- `deactivate` — Salir del entorno virtual Python actual
- `workon` — Listar los entornos virtuales disponibles
- `workon name_of_environment` — Activar el entorno virtual Python especificado
- `rmvirtualenv name_of_environment` — Borrar el entorno especificado.

Instalación de Django

Una vez que ha creado el entorno virtual, y realizado la llamada `workon` para entrar en él, puede usar *pip3* para instalar Django.

```
pip3 install django
```

Copiar al portapapeles
Puedes comprobar que está instalado Django ejecutado el siguiente comando (esto sólo comprueba que Python puede encontrar el módulo Django):

```
# Linux/Mac OS X

python3 -m django --version

 1.11.7



# Windows

py -3 -m django --version

 1.11.7
```

VELOCIDAD DE DESARROLLO

ESTRUCTURA MINIMALISTA

ESTRUCTURA FLEXIBLE
LIBRERÍAS PROPIAS DE CADA PROYECTO
AISLACIÓN DEL ENTORNO PYTHON
MANEJO DE DISTINTAS VERSIONES

VENV — CREAR ENTORNOS VIRTUALES

Propósito:	Crear contextos de instalación y ejecución aislados.

Los entornos virtuales de Python, administrados por `venv`, están configurados para instalar paquetes y ejecutar programas de una manera que los aísle de otros paquetes instalados en el resto del sistema. Debido a que cada entorno tiene su propio ejecutable de intérprete y directorio para instalar paquetes, es fácil crear entornos configurados con varias combinaciones de Python y versiones de paquetes, todo en la misma computadora.

CREAR ENTORNOS

La interfaz de línea de comando principal para `venv` se basa en la capacidad de Python para ejecutar una función «main» en un módulo utilizando la opción `-m`.

```
$ python3 -m venv /tmp/demoenv
```

Se puede instalar una aplicación de línea de comandos `pyvenv` separada, dependiendo de cómo se construyó y empaquetó el intérprete de Python. El siguiente comando tiene el mismo efecto que el ejemplo anterior.

```
$ pyvenv /tmp/demoenv
```

Se prefiere usar `-m venv` porque requiere seleccionar explícitamente un intérprete de Python, por lo que no puede haber confusión sobre el número de versión o la ruta de importación asociada con el entorno virtual resultante.

Contenidos de un entorno virtual

Cada entorno virtual contiene un directorio `bin`, donde están instalados el intérprete local y las secuencias de comandos ejecutables, un directorio `include` para archivos relacionados con la construcción de extensiones C, y un directorio `lib`, con una ubicación separada de `site-packages` para instalar paquetes.

```
$ ls -F /tmp/demoenv

bin/
include/
lib/
pyvenv.cfg
```

El directorio predeterminado `bin` contiene secuencias de comandos de «activación» para varias variantes de shell de Unix. Se pueden usar para instalar el entorno virtual en la ruta de búsqueda del shell para garantizar que el shell recupere programas instalados en el entorno. No es necesario activar un entorno para usar programas instalados en él, pero puede ser más conveniente.

```
$ ls -F /tmp/demoenv/bin

activate
activate.csh
activate.fish
easy_install*
easy_install-3.6*
pip*
pip3*
pip3.6*
python@
python3@
```

En las plataformas que los admiten, se utilizan enlaces simbólicos en lugar de copiar los ejecutables como el intérprete de Python. En este entorno, `pip` se instala como una copia local pero el intérprete es un enlace simbólico.

Finalmente, el entorno incluye un archivo `pyvenv.cfg` con configuraciones que describen cómo se configura y debe comportarse el entorno. La variable `home` apunta a la ubicación del intérprete de Python donde se ejecutó `venv` para crear el entorno. `include-system-site-packages` es un booleano que indica si los paquetes instalados fuera del entorno virtual, a nivel del sistema, deberían ser visibles dentro del entorno virtual. Y `version` es la versión de Python utilizada para crear el entorno.

pyvenv.cfg

```
# pyvenv.cfg

home = /Library/Frameworks/Python.framework/Versions/3.6/bin
include-system-site-packages = false
version = 3.6.4
```

Un entorno virtual es más útil con herramientas como `pip` y **`setuptools`** disponibles para instalar otros paquetes, por lo que `pyvenv` los instala por defecto. Para crear un entorno sin estas herramientas, pasa `--without-pip` en la línea de comando.

Usar entornos virtuales

Los entornos virtuales se usan comúnmente para ejecutar diferentes versiones de programas o para probar una versión dada de un programa con diferentes

versiones de sus dependencias. Por ejemplo, antes de actualizar de una versión de Sphinx a otra, es útil probar los archivos de documentación de entrada utilizando las versiones antiguas y nuevas. Para comenzar, crea dos entornos virtuales.

```
$ python3 -m venv /tmp/sphinx1
$ python3 -m venv /tmp/sphinx2
```

Luego instala las versiones de las herramientas para probar.

```
$ /tmp/sphinx1/bin/pip install Sphinx==1.3.6

Collecting Sphinx==1.3.6
  Using cached Sphinx-1.3.6-py2.py3-none-any.whl
Collecting Pygments>=2.0 (from Sphinx==1.3.6)
  Using cached Pygments-2.2.0-py2.py3-none-any.whl
Collecting sphinx-rtd-theme<2.0,>=0.1 (from Sphinx==1.3.6)
  Using cached sphinx_rtd_theme-0.2.4-py2.py3-none-any.whl
Collecting babel!=2.0,>=1.3 (from Sphinx==1.3.6)
  Using cached Babel-2.5.3-py2.py3-none-any.whl
Collecting alabaster<0.8,>=0.7 (from Sphinx==1.3.6)
  Using cached alabaster-0.7.10-py2.py3-none-any.whl
Collecting Jinja2>=2.3 (from Sphinx==1.3.6)
  Using cached Jinja2-2.10-py2.py3-none-any.whl
Collecting docutils>=0.11 (from Sphinx==1.3.6)
  Using cached docutils-0.14-py3-none-any.whl
Collecting snowballstemmer>=1.1 (from Sphinx==1.3.6)
  Using cached snowballstemmer-1.2.1-py2.py3-none-any.whl
Collecting six>=1.4 (from Sphinx==1.3.6)
  Using cached six-1.11.0-py2.py3-none-any.whl
Collecting pytz>=0a (from babel!=2.0,>=1.3->Sphinx==1.3.6)
  Using cached pytz-2018.3-py2.py3-none-any.whl
Collecting MarkupSafe>=0.23 (from Jinja2>=2.3->Sphinx==1.3.6)
  Using cached MarkupSafe-1.0.tar.gz
Installing collected packages: Pygments, sphinx-rtd-theme, pytz,
babel, alabaster, MarkupSafe, Jinja2, docutils, snowballstemmer,
six, Sphinx
  Running setup.py install for MarkupSafe: started
    Running setup.py install for MarkupSafe: finished with
status 'done'
Successfully installed Jinja2-2.10 MarkupSafe-1.0 Pygments-2.2.0
Sphinx-1.3.6 alabaster-0.7.10 babel-2.5.3 docutils-0.14
pytz-2018.3 six-1.11.0 snowballstemmer-1.2.1 sphinx-rtd-
theme-0.2.4

$ /tmp/sphinx2/bin/pip install Sphinx==1.4.4

Collecting Sphinx==1.4.4
  Using cached Sphinx-1.4.4-py2.py3-none-any.whl
Collecting imagesize (from Sphinx==1.4.4)
  Using cached imagesize-1.0.0-py2.py3-none-any.whl
Collecting Pygments>=2.0 (from Sphinx==1.4.4)
  Using cached Pygments-2.2.0-py2.py3-none-any.whl
Collecting snowballstemmer>=1.1 (from Sphinx==1.4.4)
  Using cached snowballstemmer-1.2.1-py2.py3-none-any.whl
Collecting alabaster<0.8,>=0.7 (from Sphinx==1.4.4)
  Using cached alabaster-0.7.10-py2.py3-none-any.whl
Collecting Jinja2>=2.3 (from Sphinx==1.4.4)
  Using cached Jinja2-2.10-py2.py3-none-any.whl
Collecting docutils>=0.11 (from Sphinx==1.4.4)
```

```
  Using cached docutils-0.14-py3-none-any.whl
Collecting babel!=2.0,>=1.3 (from Sphinx==1.4.4)
  Using cached Babel-2.5.3-py2.py3-none-any.whl
Collecting six>=1.4 (from Sphinx==1.4.4)
  Using cached six-1.11.0-py2.py3-none-any.whl
Collecting MarkupSafe>=0.23 (from Jinja2>=2.3->Sphinx==1.4.4)
  Using cached MarkupSafe-1.0.tar.gz
Collecting pytz>=0a (from babel!=2.0,>=1.3->Sphinx==1.4.4)
  Using cached pytz-2018.3-py2.py3-none-any.whl
Installing collected packages: imagesize, Pygments,
snowballstemmer, alabaster, MarkupSafe, Jinja2, docutils, pytz,
babel, six, Sphinx
  Running setup.py install for MarkupSafe: started
    Running setup.py install for MarkupSafe: finished with
status 'done'
Successfully installed Jinja2-2.10 MarkupSafe-1.0 Pygments-2.2.0
Sphinx-1.4.4 alabaster-0.7.10 babel-2.5.3 docutils-0.14
imagesize-1.0.0 pytz-2018.3 six-1.11.0 snowballstemmer-1.2.1
```

Entonces es posible ejecutar las diferentes versiones de Sphinx desde los entornos virtuales por separado, para probarlas con los mismos archivos de entrada.

```
$ /tmp/sphinx1/bin/sphinx-build --version

Sphinx (sphinx-build) 1.3.6

$ /tmp/sphinx2/bin/sphinx-build --version

Sphinx (sphinx-build) 1.4.4
```

INSTALANDO DJANGO

Ahora que tienes tu `virtualenv` iniciado, puedes instalar Django.
Antes de hacer eso, debemos asegurarnos que tenemos la última versión de `pip`, el software que utilizamos para instalar Django:
command-line

```
(myvenv) ~$ python -m pip install --upgrade pip
```

`(myvenv) ~$` pip install django

Creación del proyecto

Crear el nuevo proyecto usando el comando `django-admin startproject` como se muestra, y navega luego dentro de la carpeta.

```
django-admin startproject locallibrary .

cd locallibrary
```

Copy to Clipboard
La herramienta `django-admin` crea una estructura de carpetas/ficheros como se muestra abajo:

```
locallibrary/

    manage.py

    locallibrary/

        settings.py

        urls.py

        wsgi.py
```

Copy to Clipboard
La subcarpeta del proyecto *locallibrary* es el punto de entrada al sitio web:

- **settings.py** contiene todos los ajustes del sitio. Es donde registramos todas las aplicaciones que creamos, la localización de nuestros ficheros estáticos, los detalles de configuración de la base de datos, etc.
- **urls.py** define los mapeos url-vistas. A pesar de que éste podría contener *todo* el código del mapeo url, es más común delegar algo del mapeo a las propias aplicaciones, como verás más tarde.
- **wsgi.py** se usa para ayudar a la aplicación Django a comunicarse con el servidor web. Puedes tratarlo como código base que puedes utilizar de plantilla.

El script **manage.py** se usa para crear aplicaciones, trabajar con bases de datos y empezar el desarrollo del servidor web.

El utilitario manage.py
RUNSERVER

STARTUP
CREATESUPERUSER
ARCHIVOS DE CONFIGURACIÓN
__INIT__.PY
SETTINGS.PY
LEVANTANDO EL SERVIDOR CON MANAGE.PY
REVISANDO EL PROYECTO WEB DE DJANGO

CREANDO APLICACIONES EN DJANGO

Este artículo muestra como puedes crear un sitio web "esqueleto", que puedes luego llenar con configuraciones específicas del sitio, urls, modelos, vistas y plantillas (trataremos ésto en artículos posteriores).

El proceso es sencillo:

1. Usar la herramienta `django-admin` para crear la carpeta del proyecto, los ficheros de plantillas básicos y el script de gestión del proyecto (**manage.py**).
2. Usar **manage.py** para crear una o más *aplicaciones*.
   Nota: Un sitio web puede consistir de una o más secciones, ej. sitio principal, blog, wiki, area de descargas, etc. Django te recomienda encarecidamente que desarrolles estos componentes como *aplicaciones* separadas que podrían ser reutilizadas, si se desea, en otros proyectos.
3. Registrar las nuevas aplicaciones para incluirlas en el proyecto.
4. Conectar el mapeador url de cada aplicación.

Para el sitio web de la BibliotecaLocal la carpeta del sitio y la carpeta de su proyecto se llamarán *locallibrary*, y tendremos sólo una aplicación llamada *catalog*. El nivel más alto de la estructura de carpetas quedará por tanto como sigue:

```
locallibrary/         # Carpeta del sitio web

    manage.py         # Script para ejecutar las herramientas de Django
para este proyecto (creadas usando django-admin)

    locallibrary/     # Carpeta del Sitio web/Proyecto (creada usando
django-admin)

    catalog/          # Carpeta de la Aplicación (creada usando manage.py)
```

MVC Y LA CREACIÓN DE APLICACIONES
CORRELACIÓN DEL MODELO

COMANDOS DE AYUDA DJANGO

Los comandos de administración son scripts potentes y flexibles que pueden realizar acciones en su proyecto Django o en la base de datos subyacente. ¡Además de varios comandos predeterminados, es posible escribir los tuyos!

En comparación con los scripts de Python normales, el uso del marco de comandos de administración significa que un trabajo de configuración tedioso se realiza automáticamente entre bastidores.

Observaciones

Los comandos de gestión se pueden llamar desde:

- `django-admin <command> [options]`
- `python -m django <command> [options]`
- `python manage.py <command> [options]`
- `./manage.py <command> [options]` si manage.py tiene permisos de ejecución (`chmod +x manage.py`)

Para utilizar comandos de gestión con Cron:

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

Creación y ejecución de un comando de gestión

Para realizar acciones en Django utilizando la línea de comandos u otros servicios (donde no se usa el usuario / solicitud), puede usar los `management commands` .

Los módulos Django se pueden importar según sea necesario.

Para cada comando se necesita crear un archivo separado: `myapp/management/commands/my_command.py`

(Los directorios de `management` y `commands` deben tener un archivo __init__.py vacío)

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

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

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

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

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

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

Aquí es obligatorio el nombre de clase **Comando,** que amplía **BaseCommand** o una de sus subclases.
El nombre del comando de administración es el nombre del archivo que lo contiene. Para ejecutar el comando en el ejemplo anterior, use lo siguiente en el directorio de su proyecto:

```
python manage.py my_command
```

Tenga en cuenta que iniciar un comando puede demorar unos segundos (debido a la importación de los módulos). Entonces, en algunos casos, se recomienda crear procesos de `daemon` lugar de `management commands` de `management commands` .

CAMBIAR LA CONFIGURACIÓN

Vamos a hacer algunos cambios en `mysite/settings.py`. Abre el archivo usando el editor de código que has instalado anteriormente.
Nota: Ten en cuenta que `settings.py` es un archivo normal, como cualquier otro. Puedes abrirlo con el editor de texto, usando "file -> open" en el menu de acciones. Esto te debería llevar a la típica ventana donde puedes buscar el archivo `settings.py` y seleccionarlo. Como alternativa, puedes abrir el archivo haciendo click derecho en la carpeta djangogirls en tu escritorio. Luego, selecciona tu editor de texto en la lista. Elegir el editor es importante puesto que puede que tengas otros programas que pueden abrir el archivo pero que no te dejaran editarlo.
Sería bueno tener el horario correcto en nuestro sitio web. Ve a lista de Wikipedia de las zonas horarias y copia tu zona horaria (TZ) (e.g. `Europa/Berlín`).
En `settings.py`, encuentra la línea que contiene `TIME_ZONE` y modifícala para elegir tu zona horaria. Por ejemplo:
mysite/settings.py

```
TIME_ZONE = 'Europe/Berlin'
```
Un código de idioma tiene dos partes: el idioma, p.ej. `en` para inglés o `de` para alemán, y el código de país, p.ej. `de` para Alemania o `ch` para Suiza. Si tu idioma nativo no es el inglés, puedes añadir lo siguiente para cambiar el idioma de los botones y notificaciones de Django. Así tendrás el botón "Cancel" traducido al idioma que pongas aquí. Django viene con muchas traducciones preparadas.
Si quieres un idioma diferente, cambia el código de idioma cambiando la siguiente línea:

mysite/settings.py

```
LANGUAGE_CODE = 'es-es'
```
También tenemos que añadir una ruta para archivos estáticos. (Veremos todo acerca de archivos estáticos y CSS más adelante.) Ve al *final* del archivo, y justo debajo de la entrada `STATIC_URL`, añade una nueva llamada `STATIC_ROOT`:
mysite/settings.py

```
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```
Cuando `DEBUG` es `True` y `ALLOWED_HOST` esta vacío, el host es validado contra `['localhost', '127,0.0.1', '[::1]']`. Una vez despleguemos nuestra aplicación este no sera el mismo que nuestro nombre de host en PythonAnywhere así que cambiaremos la siguiente opción:
mysite/settings.py

```
ALLOWED_HOSTS = ['127.0.0.1', '.pythonanywhere.com']
```
Nota: si estas usando un Chromebook, añade esta linea al final del archivo settings.py: `MESSAGE_STORAGE = 'django.contrib.messages.storage.session.SessionStorage'`
Añade también `.c9users.io` a `ALLOWED_HOSTS` si estás usando cloud9.

CONFIGURAR UNA BASE DE DATO

Hay una gran variedad de opciones de bases de datos para almacenar los datos de tu sitio. Utilizaremos la que viene por defecto, `sqlite3`.
Esta ya está configurado en esta parte de tu archivo `mysite/settings.py`:
mysite/settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Para crear una base de datos para nuestro blog, ejecutemos lo siguiente en la consola: `python manage.py migrate` (necesitamos estar en el directorio de `djangogirls` que contiene el archivo `manage.py`). Si eso va bien, deberías ver algo así:
command-line

```
(myvenv) ~/djangogirls$ python manage.py migrate
Operations to perform:
  Apply all migrations: auth, admin, contenttypes, sessions
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying sessions.0001_initial... OK
```

Y, ¡terminamos! ¡Es hora de iniciar el servidor web y ver si está funcionando nuestro sitio web!

CONFIGURACION DE TEMPLATES

Django provee una práctica y poderosa API para cargar plantillas del disco, con el objetivo de quitar la redundancia en la carga de la plantilla y en las mismas plantillas.

Para usar la API para cargar plantillas, primero necesitas indicarle al framework dónde están guardadas tus plantillas. El lugar para hacer esto es en el *archivo de configuración*.

El archivo de configuración de Django es el lugar para poner configuraciones para tu instancia o proyecto de Django. Es un simple módulo de Python con variables, una por cada configuración.

Cuando ejecutaste `django-admin.py startproject mysite` en el Capítulo 2, el script creó un archivo de configuración por omisión por ti, llamado `settings.py`. Échale un vistazo al contenido del archivo. Este contiene variables que se parecen a estas (no necesariamente en este orden):

- DEBUG = True
- TIME_ZONE = 'America/Chicago'

- USE_I18N = True
- ROOT_URLCONF = 'mysite.urls'

Éstas se explican por sí solas; las configuraciones y sus respectivos valores son simples variables de Python. Como el archivo de configuración es sólo un módulo plano de Python, puedes hacer cosas dinámicas como verificar el valor de una variable antes de configurar otra. (Esto también significa que debes evitar errores de sintaxis de Python en los archivos de configuración).

Cubriremos el archivo de configuración en profundidad en el Apéndice E, pero por ahora, veamos la variable de configuración `TEMPLATE_DIRS`. Esta variable le indica al mecanismo de carga de plantillas dónde buscar las plantillas. Por omisión, ésta es una tupla vacía. Elige un directorio en el que desees guardar tus plantillas y agrega este a `TEMPLATE_DIRS`, así:

```
TEMPLATE_DIRS = ('/home/django/mysite/templates',)
```

Hay algunas cosas para notar:

- Puedes especificar cualquier directorio que quieras, siempre y cuando la cuenta de usuario en el cual se ejecuta el servidor web tengan acceso al directorio y su contenido. Si no puedes pensar en un lugar apropiado para poner las plantillas, te recomendamos crear un directorio `templates` dentro del proyecto de Django (esto es, dentro del directorio `mysite` que creaste en el Capítulo 2 , si vienes siguiendo los ejemplos a lo largo del libro).
- ¡No olvides la coma al final del string del directorio de plantillas! Python requiere una coma en las tuplas de un solo elemento para diferenciarlas de una expresión de paréntesis. Esto es común en los usuarios nuevos.

```
Si quieres evitar este error, puedes hacer `TEMPLATE_DIRS` una lista,


en vez de una tupla, porque un solo elemento en una lista no requiere


estar seguido de una coma:



TEMPLATE_DIRS = ['/home/django/mysite/templates']
```

- Una tupla es un poco más correcta semánticamente que una lista (las tuplas no pueden cambiar luego de ser creadas, y nada podría cambiar las configuraciones una vez que fueron leídas), nosotros recomendamos usar tuplas para la variable `TEMPLATE_DIRS`.
- Si estás en Windows, incluye tu letra de unidad y usa el estilo de Unix para las barras en vez de barras invertidas, como sigue::

```
TEMPLATE_DIRS = ('C:/www/django/templates',)
```

- Es más sencillo usar rutas absolutas (esto es, las rutas de directorios comienzan desde la raíz del sistema de archivos). Si quieres ser un poco más flexible e independiente, también, puedes tomar el hecho de que el archivo de configuración de Django es sólo código de Python y construir la variable `TEMPLATE_DIRS` dinámicamente.

```
Este ejemplo usa la variable de Python "mágica" `__file__`, la cual es

automáticamente asignada al nombre del archivo del módulo de Python en

el que se encuentra el código.



import os.path



    TEMPLATE_DIRS = (

        os.path.join(os.path.dirname(__file__),
'templates').replace('\\','/'),

    )
```

Con la variable `TEMPLATE_DIRS` configurada, el próximo paso es cambiar el código de vista que usa la funcionalidad de carga de plantillas de Django, para no incluir la ruta a la plantilla. Volvamos a nuestra vista `current_datetime`, vamos a cambiar esta como sigue:

```
from django.template.loader import get_template

from django.template import Context

from django.http import HttpResponse

import datetime
```

```
def current_datetime(request):

    now = datetime.datetime.now()

    t = get_template('current_datetime.html')

    html = t.render(Context({'current_date': now}))

    return HttpResponse(html)
```

En este ejemplo, usamos la función `django.template.loader.get_template()` en vez de cargar la plantilla desde el sistemas de archivos manualmente. La función `get_template()` toma el nombre de la plantilla como argumento, se da cuenta de dónde está la plantilla en el sistema de archivos, lo abre, y retorna un objeto `Template` compilado.

Si `get_template()` no puede encontrar la plantilla con el nombre pasado, esta levanta una excepción `TemplateDoesNotExist`. Para ver que cómo se ve eso, ejecutar el servidor de desarrollo de Django otra vez, como en el Capítulo 3, ejecutando `python manage.py runserver` en el directorio de tu proyecto de Django. Luego, escribe en tu navegador la página que activa la vista `current_datetime` (o sea, `http://127.0.0.1:8000/time/`). Asumiendo que tu variable de configuración `DEBUG` está asignada a `True` y todavía no has creado la plantilla `current_datetime.html`, deberías ver una página de error de Django resaltando el error `TemplateDoesNotExist`.

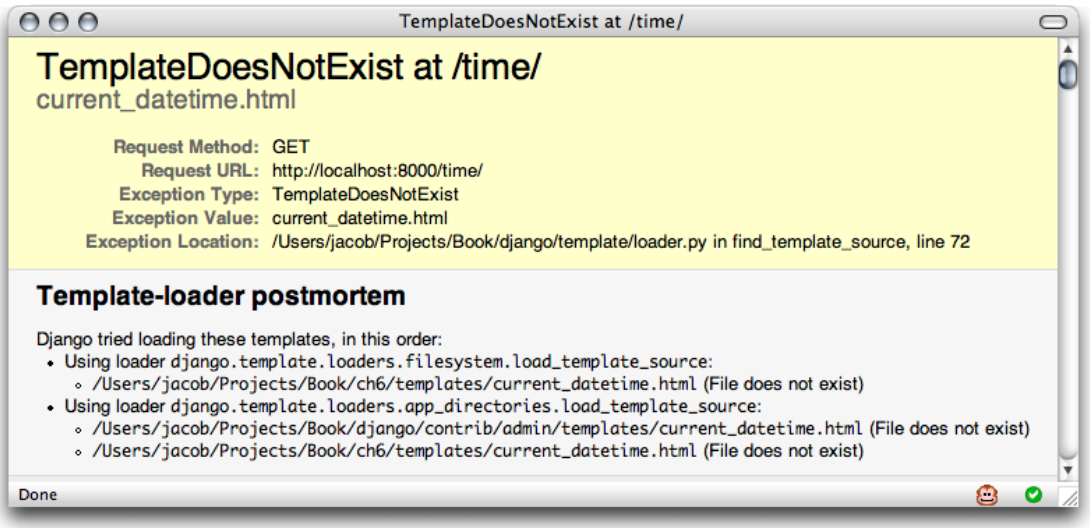


Figura 4.1 La página de error que se muestra cuando una plantilla no se encuentra

Esta página de error es similar a la que explicamos en el Capítulo 3, con una pieza adicional de información de depuración: una sección "Postmortem del cargador de plantillas". Esta sección te indica qué plantilla intentó cargar Django acompañado de una razón para cada intento fallido (por ej. *"File does not exist"*). Esta información es invaluable cuando hacemos depuración de errores de carga de plantillas.

Como probablemente puedas distinguir de los mensajes de error de la Figura 4-1, Django intentó buscar una plantilla combinando el directorio de la variable `TEMPLATE_DIRS` con el nombre de la plantilla pasada a `get_template()`. Entonces si tu variable `TEMPLATE_DIRS` contiene `'/home/django/templates'`, Django buscará `'/home/django/templates/current_datetime.html'`.
Si `TEMPLATE_DIRS` contiene más que un directorio, cada uno de estos es examinado hasta que se encuentre la plantilla o hasta que no haya más directorios.

Continuando, crea el archivo `current_datetime.html` en tu directorio de plantillas usando el siguiente código:

```
<html><body>It is now {{ current_date }}.</body></html>
```

Refresca la página en tu navegador web, y deberías ver la página completamente renderizada.

CONFIGURACION DE PATHS

CONFIGURACION URLS.PY

6.2.1.- Creación de un proyecto de Django.

Hasta el momento en que se escribe este módulo, se tiene la versión 4.0 de Django disponible desde PIP, por lo que utilizaremos esta versión junto con su documentación actualizada. Antes de comenzar, primero debemos tener instalado un gestor de bases de datos. como vimos en la introducción, podría ser MySQL, PostgreSQL, MariaDB, SQLite, Oracle. En este módulo utilizaremos el gestor de bases de datos PostgreSQL puesto que es el más usado para los proyectos diseñados con Python Django.

6.2.1.1- Instalación de PostgreSQL.

WINDOWS

En primera instancia iremos al link de descarga de abajo para realizar la instalación del gestor de bases de datos en windows 10.

PostgreSQL: Windows installers

El link a continuación muestra un mini tutorial para enseñarte a instalarlo luego que este dentro de la página de postgresql.

https://www.awesomescreenshot.com/video/6693128?key=9a5db573c76b354529e715a695164819

Luego de haber descargado el .exe le dan click y solo siguen los instructivos de la instalación de la aplicación.

LINUX

En linux iremos de igual forma al link de descarga a continuación y escogeremos nuestra distribución linux.

PostgreSQL: Linux downloads (other)

Luego de estar dentro de la página de descarga de postgres para linux veremos el siguiente video.

https://www.awesomescreenshot.com/video/6693370?key=159ef4e5f42137be5cfef6e29a6a08ff

6.2.1.2. Instalar Django desde PIP

Instalar pip. Lo más fácil es usar el instalador de pip independiente. Si su distribución ya tiene pip instalado, es posible que deba actualizarla si está desactualizada. Si está desactualizado, lo sabrá porque la instalación no funcionará.

```
$ python -m pip install Django
```

6.3.2.- Referencias

[1] Django Documentation
https://docs.djangoproject.com/en/3.1/

[2] D. Feldroy & A. Feldroy , Two Scoops of Django 3.x, Best Practices for the Django Web Frame-work, 2020.

[3] William S. Vincent, Django for Beginners Build websites with Python & Django, 2020.

[4] Nigel George, Build a Website with Django 3, 2019.

[5] Django Tutorial
https://www.geeksforgeeks.org/django-tutorial/