Table of Contents

Tutorial Django Girls: Extensiones

Introducción

Este libro contiene tutoriales adicionales que pueder hacer luego de haber terminado el Tutorial de Django Girls.

Estos son los tutoriales actuales:

- Deberes: ¡agrega más a tu sitio web!
- Deberes: asegurar tu sitio web
- Deberes: crear el model comentario
- Opcional: installación de PostgreSQL

Colaborar

Estos tutoriales son mantenidos por DjangoGirls y su tradución al Español por Python Argentina. Si encuentras un error o quieres actualizar el tutorial por favor sigue la guía para colaborar de Django Girls o escribe un email a la lista de emails de Python Argentina.

Deberes: Añadiendo seguridad a tu sitio web

Te habrás dado cuenta de que no has tenido que usar ninguna clave, excepto cuando, con anterioridad, usamos la interfaz de administración. Te habrás dado cuenta también de que esto significa que cualquiera puede añadir o editar entradas (posts) en tu blog. Yo no se tú, pero yo no querría que cualquiera publicase en mi blog. Así que hagamos algo al respecto.

Autorización para añadir/editar entradas en el blog

Primero hagamos las cosas seguras. Protegeremos nuestras vistas `post_new`, `post_edit`, `post_draft_list`, `post_remove` y `post_publish` para que solamente usuarios registrados puedan acceder a ellas. Django viene con algunas herramientas para ello, son elementos bastante avanzados llamados *decoradores* (*decorators*) . No te preocupes por tecnicismos ahora, puedes leer más sobre este tema después. El decorador que debemos usar se encuentra en el módulo `django.contrib.auth.decorators` y se llama `login_required`.

Así que edita el archivo `blog/views.py` y añade estas líneas al principio junto al resto de los *imports*:

```
from django.contrib.auth.decorators import login_required
```

Después añade una línea antes de cada vista `post_new`, `post_edit`, `post_draft_list`, `post_remove` y `post_publish` (decorándolas) como la siguiente:

```
@login_required
def post_new(request):
    [...]
```

¡Y ya está! Ahora intenta acceder a `http://localhost:8000/post/new/`, ¿ves la diferencia?

> Si te ha salido un formulario vacío, seguramente estés autenticada desde el capítulo de la interfaz de administración. Ve a `http://localhost:8000/admin/logout/` para salir (log out), después ve a `http://localhost:8000/post/new` otra vez.

Deberías obtener uno de nuestros queridos errores. Este, de hecho, es bastante interesante: El decorador que añadimos antes te redirigirá a la pantalla de autenticación. Pero todavía no está disponible, así que recibirás un "Página no encontrada (404)".

No olvides añadir el decorador encima de `post_edit`, `post_remove`, `post_draft_list` y `post_publish` también.

Hurra, ¡hemos alcanzado parte de nuestro objetivo! Otras personas ya no podrán crear entradas en nuestro blog. Desafortunadamente nosotros tampoco podremos. Así que arreglemos esto ahora mismo.

Autenticación de usuario

Ahora podríamos intentar hacer un montón de magia para implementar usuarios, claves y autenticación, pero hacer este tipo de cosas correctamente es bastante complicado. Como Django viene con las "pilas incluidas", alguien ha hecho este trabajo duro por nosotros, así que haremos uso de estas herramientas de autenticación proporcionadas.

En tu archivo `mysite/urls.py` añade la url `url(r'^accounts/login/$', 'django.contrib.auth.views.login')`. De manera que ahora el archivo se parezca a esto:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^accounts/login/$', 'django.contrib.auth.views.login'),
    url(r'', include('blog.urls')),
)
```

Ahora necesitamos una plantilla para la página de autenticación o inicio de sesión, así que creamos un directorio `blog/templates/registration` y un archivo dentro llamado `login.html`:

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

{% block content %}

{% if form.errors %}
<p>Usuario o clave incorrectos. Vuelve a intentarlo, por favor.</p>
{% endif %}

<form method="post" action="{% url 'django.contrib.auth.views.login' %}">
{% csrf_token %}
<table>
<tr>
    <td>{{ form.username.label_tag }}</td>
    <td>{{ form.username }}</td>
</tr>
<tr>
    <td>{{ form.password.label_tag }}</td>
    <td>{{ form.password }}</td>
</tr>
</table>

<input type="submit" value="login" />
<input type="hidden" name="next" value="{{ next }}" />
</form>

{% endblock %}
```

Verás que esto hace uso de nuestra plantilla base para que tenga el aspecto general de tu blog.

Lo maravilloso aquí es que esto simplemente funciona. No tenemos que ocuparnos del envío seguro de formularios y de claves. Sólo nos queda una cosa, deberíamos añadir una configuración a `mysite/settings.py` :

```
LOGIN_REDIRECT_URL = '/'
```

Ahora cuando se accede directamente a la página de inicio de sesión, si el usuario se autentica con éxito, se le redirigirá a la raíz del sitio (/).

Mejorando el diseño

Así que ahora nos hemos asegurado de que solo los usuarios autorizados (es decir, nosotros) pueden añadir, editar o publicar entradas en el blog. Pero los botones de añadir y editar entradas pueden ser vistos todavía por todo el mundo, escondámoslos para los usuarios que no estén autenticados. Para esto necesitamos editar las plantillas, así que empecemos con la plantilla base `blog/templates/blog/base.html` :

```
<body>
    <div class="page-header">
        {% if user.is_authenticated %}
        <a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-p
lus"></span></a>
        <a href="{% url 'post_draft_list' %}" class="top-menu"><span class="glyphicon glyp
hicon-edit"></span></a>
        {% else %}
        <a href="{% url 'django.contrib.auth.views.login' %}" class="top-menu"><span class
="glyphicon glyphicon-lock"></span></a>
        {% endif %}
        <h1><a href="{% url 'blog.views.post_list' %}">Django Girls</a></h1>
    </div>
    <div class="content">
        <div class="row">
            <div class="col-md-8">
            {% block content %}
            {% endblock %}
            </div>
        </div>
    </div>
</body>
```

Tal vez reconozcas el patrón aquí. Hay una condición *if* dentro de la plantilla que comprueba que el usuario esté autenticado para mostrar los botones de editar. De lo contrario muestra el botón de inicio de sesión (login).

Deberes: Edita la plantilla `blog/templates/blog/post_detail.html` para mostrar solamente los botones de editar a los usuarios autenticados.

Más sobre usuarios autenticados

Añadamos un poco de salsa a nuestras plantillas mientras estemos en ellas. Primero añadiremos un indicador de que estamos autenticados. Edita `blog/templates/blog/base.html` de la siguiente manera:

```
<div class="page-header">
    {% if user.is_authenticated %}
    <a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"
></span></a>
    <a href="{% url 'post_draft_list' %}" class="top-menu"><span class="glyphicon glyphico
n-edit"></span></a>
    <p class="top-menu">Hola {{ user.username }}<small>(<a href="{% url 'django.contrib.au
th.views.logout' %}">Log out</a>)</small></p>
    {% else %}
    <a href="{% url 'django.contrib.auth.views.login' %}" class="top-menu"><span class="gl
yphicon glyphicon-lock"></span></a>
    {% endif %}
    <h1><a href="{% url 'blog.views.post_list' %}">Django Girls</a></h1>
</div>
```

Esto añade un bonito "Hola " para recordarnos quien somos y que estamos autenticados. También añade un link para cerrar la sesión. Pero como te habrás dado cuenta, esto todavía no funciona. ¡Ups! ¡Hemos roto internet! ¡Arreglémosla!

Hemos decidido dejar que Django gestione el inicio de sesión, veamos si Django puede gestionar también el cierre de sesión por nosotros. Echa un vistazo en https://docs.djangoproject.com/en/1.8/topics/auth/default/ a ver si puedes encontrar algo.

¿Terminaste de leer? Deberías estar pensando en añadir una url (en `mysite/urls.py`) apuntando a la vista `django.contrib.auth.views.logout`. Así:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^accounts/login/$', 'django.contrib.auth.views.login'),
    url(r'^accounts/logout/$', 'django.contrib.auth.views.logout', {'next_page': '/'}),
    url(r'', include('blog.urls')),
)
```

¡Ya está! Si has seguido todas las instrucciones hasta ahora (y hecho los deberes), tienes un blog donde:

- necesitas un usuario y una clave para iniciar sesión y autenticarte,
- necesitas estar autenticado para añadir/editar/publicar(/borrar) entradas
- y puedes cerrar la sesión de nuevo

Deberes: crear el modelo Comment

Hasta aquí solo tenemos un modelo Post, pero ¿qué hay de recibir algunos comentarios de tus lectores?

Creando el modelo de comentarios del blog

Vamos a abrir `blog/models.py` y agregar este fragmento de código al final del archivo:

```
class Comment(models.Model):
    post = models.ForeignKey('blog.Post', related_name='comments')
    author = models.CharField(max_length=200)
    text = models.TextField()
    created_date = models.DateTimeField(default=timezone.now)
    approved_comment = models.BooleanField(default=False)

    def approve(self):
        self.approved_comment = True
        self.save()

    def __str__(self):
        return self.text
```

Puedes volver al capítulo **Modelos en Django** si necesitas recordar lo que significa cada tipo de campo.

En este capítulo tenemos un nuevo tipo de campo:

- `models.BooleanField` - este es un campo verdadero/falso.

Además la opción `related_name` en `models.ForeignKey` nos permite tener acceso a los comentarios desde el modelo post.

Crear tablas para los modelos en tu base de datos

Ahora es momento de agregar el modelo de comentarios a la base de datos. Para esto tenemos que hacerle conocer a Django que hicimos cambios en nuestro modelo. Escribe `python manage.py makemigrations blog` . De esta forma:

```
(myvenv) ~/djangogirls$ python manage.py makemigrations blog
Migrations for 'blog':
  0002_comment.py:
    - Create model Comment
```

Puedes ver que este comando crea por nosotros otro archivo de migración en el directorio `blog/migrations/` . Ahora necesitamos aplicar estos cambios con `python manage.py migrate blog` . Debería verse así:

```
(myvenv) ~/djangogirls$ python manage.py migrate blog
Operations to perform:
  Apply all migrations: blog
Running migrations:
  Rendering model states... DONE
  Applying blog.0002_comment... OK
```

Nuestro modelo Comment ahora existe en la base de datos. Sería lindo si tenemos acceso a él en nuestro panel de administración.

Registrar el modelo comment en el panel de administración

Para registrar el modelo en el panel de administración ve a `blog/admin.py` y agrega la línea:

```
admin.site.register(Comment)
```

No olvides de importar el modelo Comment, el archivo debería verse como esto:

```
from django.contrib import admin
from .models import Post, Comment

admin.site.register(Post)
admin.site.register(Comment)
```

Si escribes `python manage.py runserver` en el símbolo de sistema y vas con el navegador a http://127.0.0.1:8000/admin/ deberías tener acceso a listar, agregar y eliminar comentarios. No dudes en jugar con esto.

Hacer visible nuestros comentarios

Ve al archivo `blog/templates/blog/post_detail.html` y agrega las líneas previas al tag `{% endblock %}` :

```
<hr>
{% for comment in post.comments.all %}
    <div class="comment">
        <div class="date">{{ comment.created_date }}</div>
        <strong>{{ comment.author }}</strong>
        <p>{{ comment.text|linebreaks }}</p>
    </div>
{% empty %}
    <p>No comments here yet :(</p>
```

```
{% endfor %}
```

Ahora podremos ver la sección de comentarios en las páginas con detalles del post.

Pero esto puede verse algo mejor, agrega algo de css a `static/css/blog.css` :

```
.comment {
    margin: 20px 0px 20px 20px;
}
```

Podemos también permitir a los visitantes conocer sobre los comentarios en la página de listado de publicaciones. Vea al archivo `blog/templates/blog/post_list.html` y agrega la línea:

```
<a href="{% url 'blog.views.post_detail' pk=post.pk %}">Comments: {{ post.comments.count }
}</a>
```

Luego de esto nuestra plantilla debería verse así:

```
{% extends 'blog/base.html' %}

{% block content %}
    {% for post in posts %}
        <div class="post">
            <div class="date">
                {{ post.published_date }}
            </div>
            <h1><a href="{% url 'blog.views.post_detail' pk=post.pk %}">{{ post.title }}</
a></h1>
            <p>{{ post.text|linebreaks }}</p>
            <a href="{% url 'blog.views.post_detail' pk=post.pk %}">Comments: {{ post.comm
ents.count }}</a>
        </div>
    {% endfor %}
{% endblock content %}
```

Deja que tus lectores escriban comentarios

Ahora mismo podemos ver comentarios en nuestro blog pero no podemos agregarlos, ¡vamos a cambiar esto!

Ve a `blog/forms.py` y agrega estas líneas al final del archivo:

```
class CommentForm(forms.ModelForm):

    class Meta:
        model = Comment
        fields = ('author', 'text',)
```

No olvides importar el modelo Comment, cambia la línea:

```
from .models import Post
```

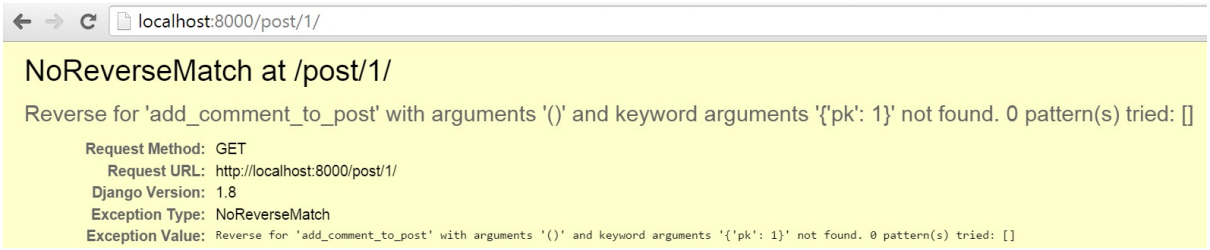
por:

```
from .models import Post, Comment
```

Ahora ve a `blog/templates/blog/post_detail.html` y antes de la línea `{% for comment in post.comments.all %}` agrega:

```
<a class="btn btn-default" href="{% url 'add_comment_to_post' pk=post.pk %}">Add comment</a>
```

Ve a la página de detalles del post y deberías ver el error:



Vamos a solucionarlo. Ve a `blog/urls.py` y agrega este patrón a `urlpatterns`:

```
url(r'^post/(?P<pk>[0-9]+)/comment/$', views.add_comment_to_post, name='add_comment_to_post'),
```

Ahora deberías ver este error:

Para arreglarlo, agrega este fragmento de código a `blog/views.py`:

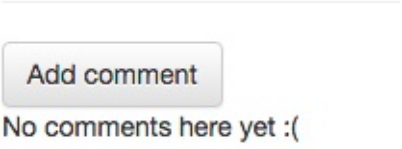
```
def add_comment_to_post(request, pk):
```

```
    post = get_object_or_404(Post, pk=pk)
    if request.method == "POST":
        form = CommentForm(request.POST)
        if form.is_valid():
            comment = form.save(commit=False)
            comment.post = post
            comment.save()
            return redirect('blog.views.post_detail', pk=post.pk)
    else:
        form = CommentForm()
    return render(request, 'blog/add_comment_to_post.html', {'form': form})
```

No olvides los imports al comienzo del archivo:

```
from .forms import PostForm, CommentForm
```

Ahora, en la página de detalles de la publicación, deberías ver el botón "Agregar Comentario"



Sin embargo, cuando hagas click en el botón deberías ver:

Tal como menciona el error, la plantilla no existe, crea una como `blog/templates/blog/add_comment_to_post.html` y agrega estas líneas:

```
{% extends 'blog/base.html' %}

{% block content %}
    <h1>New comment</h1>
    <form method="POST" class="post-form">{% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Send</button>
    </form>
{% endblock %}
```

¡Sí! ¡Ahora tus lectores pueden hacerte saber qué es lo que piensan de tus publicaciones!

Moderar tus comentarios

No todos los comentarios deberían ser mostrados. El dueño del blog debería tener la opción de aprobar o eliminar comentarios. Vamos a hacer algo sobre esto.

Ve a `blog/templates/blog/post_detail.html` y cambia las líneas:

```
{% for comment in post.comments.all %}
    <div class="comment">
        <div class="date">{{ comment.created_date }}</div>
        <strong>{{ comment.author }}</strong>
        <p>{{ comment.text|linebreaks }}</p>
    </div>
{% empty %}
    <p>No comments here yet :(</p>
{% endfor %}
```

por:

```
{% for comment in post.comments.all %}
    {% if user.is_authenticated or comment.approved_comment %}
    <div class="comment">
        <div class="date">
            {{ comment.created_date }}
            {% if not comment.approved_comment %}
                <a class="btn btn-default" href="{% url 'comment_remove' pk=comment.pk %}"
><span class="glyphicon glyphicon-remove"></span></a>
                <a class="btn btn-default" href="{% url 'comment_approve' pk=comment.pk %}
"><span class="glyphicon glyphicon-ok"></span></a>
            {% endif %}
        </div>
        <strong>{{ comment.author }}</strong>
        <p>{{ comment.text|linebreaks }}</p>
    </div>
    {% endif %}
{% empty %}
    <p>No comments here yet :(</p>
{% endfor %}
```

Deberías ver `NoReverseMatch`, porque no existe una url que coincida con el patrón `comment_remove` y `comment_approve`

Agrega el patrón de url a `blog/urls.py`:

```
url(r'^comment/(?P<pk>[0-9]+)/approve/$', views.comment_approve, name='comment_approve'),
url(r'^comment/(?P<pk>[0-9]+)/remove/$', views.comment_remove, name='comment_remove'),
```

Ahora deberías ver `AttributeError`. Para deshacerte de esto, crea mas vistas en `blog/views.py`:

```
@login_required
def comment_approve(request, pk):
    comment = get_object_or_404(Comment, pk=pk)
    comment.approve()
    return redirect('blog.views.post_detail', pk=comment.post.pk)

@login_required
def comment_remove(request, pk):
    comment = get_object_or_404(Comment, pk=pk)
    post_pk = comment.post.pk
    comment.delete()
    return redirect('blog.views.post_detail', pk=post_pk)
```

Y por supuesto arregla los imports.

Todo funciona, pero hay un error. En nuestra página de listado de publicaciones bajo la publicación podemos ver el número de todos los comentarios agregados, pero ahí queremos tener el número de comentarios aprobados.

Ve a `blog/templates/blog/post_list.html` y modifica la línea:

```
<a href="{% url 'blog.views.post_detail' pk=post.pk %}">Comments: {{ post.comments.count }
}</a>
```

por:

```
<a href="{% url 'blog.views.post_detail' pk=post.pk %}">Comments: {{ post.approved_comment
s.count }}</a>
```

Y también agrega este método al modelo Post en `blog/models.py`:

```
def approved_comments(self):
    return self.comments.filter(approved_comment=True)
```

Ahora la funcionalidad sobre comentarios está finalizada! Felicitaciones :-)

Instalación de PostgreSQL

> Parte de este capítulo está basado en los tutoriales de Geek Girls Carrots (http://django.carrots.pl/).
>
> Parte de este capítulo esta basado en django-marcador tutorial licenciado bajo Creative Commons Attribution-ShareAlike 4.0 International License. El tutorial django-marcador tiene derechos de autor de Markus Zapke-Gründemann et al.

Windows

La forma mas fácil de instalar PostgreSQL en Windows es usando un programa que puedes encontrar aquí http://www.enterprisedb.com/products-services-training/pgdownload#windows

Elige la versión mas nueva disponible para tu sistema operativo. Descarga el instalador, ejecútalo y sigue las instrucciones disponibles aquí: http://www.postgresqltutorial.com/install-postgresql/. Toma nota del directorio de instalación ya que lo necesitarás en el siguiente paso (usualmente es `C:\Program Files\PostgreSQL\9.3`).

Mac OS X

La forma mas fácil es descargar Postgres.app e instalarla como cualquier otra aplicación en tu sistema operativo.

Descárgala, arrástrala al directorio Aplicaciones y ejecútala con doble click. ¡Eso es todo!

También vas a tener que agregar las herramientas de línea de comandos de Postgres a tu variable `PATH` , que se explica aquí.

Linux

Los pasos de instalación varían de acuerdo a tu distribución. Abajo están los comandos para Ubuntu y Fedora, pero si estas usando una distribución diferente revisa la documentación de PostgreSQL.

Ubuntu

Ejecuta el siguiente comando:

```
sudo apt-get install postgresql postgresql-contrib
```

Fedora

Ejecuta el siguiente comando:

```
sudo yum install postgresql93-server
```

Crear la base de datos

A continuación, necesitamos crear nuestra primera base de datos y un usuario que pueda acceder a esa base de datos. PostgreSQL te permite crear tantas bases de datos y usuarios como gustes, así que si estas corriendo mas de un sitio deberías crear una base de datos para cada uno.

Windows

Si estás usando Windows, hay algunos pasos más que necesitamos completar. Por ahora no es importante que entiendas la configuración que estamos haciendo aquí, pero siéntete libre de preguntarle al Guía si tienes curiosidad por saber qué está sucediendo.

1. Abre el Símbolo del Sistema (Menú inicio → Todos los programas → Accesorios → Simbolo del Sistema)
2. Ejecuta los siguiente escribiéndolo y presionando la tecla `Enter`: `setx PATH "%PATH%;C:\Program Files\PostgreSQL\9.3\bin"` . Puedes pegar esto en el Simbolo del Sistema haciendo click derecho y seleccionando `Pegar` . Asegúrate de que la dirección del directorio es igual a la que tomaste nota durante la instalación más el agregado de `\bin` al final. Deberías ver un mensaje similar a `ÉXITO: El valor específico fue guardado` .
3. Cierra y vuelve a abrir el Simbolo de Sistema.

Crear la base de datos

Primero, vamos a lanzar la consola de Postgres ejecutando `psql` . ¿Recuerdas como lanzar la consola?

> En Mac OS X puedes hacer esto lanzando la aplicación `Terminal` (está en Aplicaciones → Utilidades). En Linux, está probablemente en Aplicaciones → Accesorios → Terminal. En Windows necesitas ir a Menú inicio → Todos los programas → Accesorios → Simbolo del Sistema. Además, en Windows, `psql` podría requerir iniciar sesión usando el usuario y contraseña que elegiste durante la instalación. Si `psql` pregunta por tu contraseña y parece no funcionar, prueba `psql -U <usuario> -W` presiona Enter e ingresa tu contraseña.

```
$ psql
psql (9.3.4)
Type "help" for help.
#
```

Nuestro `$` ahora cambió a `#`, lo cual significa que ahora estamos enviando comandos a PostgreSQL. Vamos a crear un usuario:

```
# CREATE USER name;
CREATE ROLE
```

Reemplaza `name` con tu propio nombre. No deberías usar letras con acentos o espacios en blanco (ejemplo: `bożena maria` es inválido - necesitas convertirlo a `bozena_maria`).

Ahora es tiempo de crear una base de datos para tu proyecto Django:

```
# CREATE DATABASE djangogirls OWNER name;
CREATE DATABASE
```

Recuerda reemplazar `name` con el nombre que elegiste (ejemplo: `bozena_maria`).

Genial - ¡El tema bases de datos ya está listo!

Actualizar la configuración de django

Encuentra esta parte en tu archivo `mysite/settings.py`:

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

Y reemplázala con esto:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'djangogirls',
        'USER': 'name',
        'PASSWORD': '',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

Recuerda cambiar `name` por el nombre de usuario que creaste anteriormente en este capítulo.

Instalar el paquete de PostgreSQL para Python

Primero, instala Heroku Toolbelt desde https://toolbelt.heroku.com/. Vamos a necesitarlo para desplegar tu sitio mas adelante, esto incluye Git, lo que podría ser útil ahora.

Luego, necesitamos instalar un paquete que le permite a Python hablar con PostgreSQL - este se llama `psycopg2`. Las instrucciones de instalación difieren levemente entre Windows y Linux/OS X.

Windows

Para Windows, descarga el archivo pre-construido desde http://www.stickpeople.com/projects/python/win-psycopg/

Asegúrate de obtener el archivo correspondiente a tu versión de Python (3.4 debería ser la última linea) y la correcta arquitectura (32 bits en la columna izquierda o 64 bits en la columna derecha).

Renombra el archivo descargado y muévelo de modo que este disponible en `C:\psycopg2.exe`.

Una vez que este hecho, ingresa el siguiente comando en la terminal (asegúrate que tu `entorno virtual` este activado):

```
easy_install C:\psycopg2.exe
```

Linux y OS X

Ejecuta el siguiente comando en tu consola:

```
(myvenv) ~/djangogirls$ pip install psycopg2
```

Si todo va bien, verás algo como esto

```
Downloading/unpacking psycopg2
Installing collected packages: psycopg2
Successfully installed psycopg2
Cleaning up...
```

Una vez que este completo, ejecuta `python -c "import psycopg2"`. Si no obtienes ningún error, todo esta instalado satisfactoriamente.

Despliegue en Heroku (así como en PythonAnyhere)

Siempre es bueno para un desarrollador tener un par de opciones diferentes de despliegue en su haber. ¿Por qué no intentar desplegar tu sitio web en Heroku, así como en PythonAnywhere?

Heroku es también gratuito para pequeñas aplicaciones que no tienen muchas visitas, pero un poco más complicado para desplegar.

Seguiremos este tutorial: https://devcenter.heroku.com/articles/getting-started-with-django, pero lo pegamos aquí por si es más fácil para ti.

El archivo `requirements.txt`

Si no lo creaste antes, necesitamos crear ahora un archivo `requirements.txt` para decirle a Heroku qué paquetes de Python necesitan estar instalados en nuestro servidor.

Pero primero, Heroku necesita que nosotros instalemos algunos paquetes nuevos. Ve a la consola con tu `virtualenv` (entorno virtual) activado y escribe lo siguiente:

```
(myvenv) $ pip install dj-database-url gunicorn whitenoise
```

Cuando finalice la instalación, ve al directorio `djangogirls` y ejecuta este comando:

```
(myvenv) $ pip freeze > requirements.txt
```

Esto creará un archivo llamado `requirements.txt` con una lista de tus paquetes instalados (i.e. bibliotecas de Python que estás usando, por ejemplo Django :)).

> **Nota**: `pip freeze` lista todas las bibliotecas Python instaladas en tu virtualenv, y el `>` toma la salida de `pip freeze` y la pone en un archivo. ¡Intenta ejecutar `pip freeze` sin el `>` `requirements.txt` para ver qué pasa!

Abre este archivo y añádele la siguiente línea al final:

```
psycopg2==2.5.4
```

Esta línea se necesita para que tu aplicación funcione en Heroku.

Procfile

Otro archivo que Heroku necesita es el Procfile. Éste le dice a Heroku qué comandos ejecutar para iniciar nuestro sitio web. Abre el editor de código, crea un archivo llamado `Procfile` en el directorio `djangogirls` y añade esta línea:

```
web: gunicorn mysite.wsgi
```

Esta línea significa que vamos a desplegar una aplicación `web` , y que lo haremos ejecutando el comando `gunicorn mysite.wsgi` (`gunicorn` es como una versión más potente del comando `runserver` de Django).

Ahora guárdalo. ¡Hecho!

El archivo `runtime.txt`

También necesitamos decirle a Heroku qué versión de Python queremos usar. Esto se hace creando un archivo `runtime.txt` en el directorio `djangogirls` usando el comando "nuevo archivo" de tu editor, y poniendo el siguiente texto (¡y nada más!) dentro:

```
python-3.4.2
```

`mysite/local_settings.py`

Como es más restrictivo que PythonAnywhere, Heroku necesita usar una configuración diferente de la que usamos localmente (en nuestro ordenador). Por ejemplo, Heroku quiere usar Postgres mientras que nosotros usamos SQLite. Por eso necesitamos crear un archivo de configuración separado que estará sólo disponible para nuestro entorno local.

Continúa y crea el archivo `mysite/local_settings.py` . Debería contener la configuración para `DATABASE` de tu archivo `mysite/settings.py` . Así:

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(__file__))

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

DEBUG = True
```

Ahora ¡sólo guárdalo! :)

mysite/settings.py

Otra cosa que necesitamos hacer es modificar el archivo `settings.py` de nuestro sitio web. Abre `mysite/settings.py` en tu editor y añade las siguientes líneas al final del archivo:

```
import dj_database_url
DATABASES['default'] = dj_database_url.config()

SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

ALLOWED_HOSTS = ['*']

DEBUG = False

try:
    from .local_settings import *
except ImportError:
    pass
```

Esto hará la configuración necesaria para Heroku y también importará tu configuración local si existe el archivo `mysite/local_settings.py` .

Guarda el archivo.

mysite/wsgi.py

Abre el archivo `mysite/wsgi.py` y añade estas líneas al final:

```
from whitenoise.django import DjangoWhiteNoise
application = DjangoWhiteNoise(application)
```

¡Perfecto!

Cuenta en Heroku

Necesitas instalar la herramienta Heroku *toolbelt* que puedes encontrar aquí (puedes saltarte la instalación si ya lo has hecho durante la configuración inicial): https://toolbelt.heroku.com/

> Si ejecutas el programa de instalación de Heroku *toolbelt* en Windows asegúrate de elegir "Custom Installation" cuando te pregunte qué componentes instalar. En la lista de componentes que aparece después añade a la selección "Git and SSH".
>
> En Windows también debes ejecutar la siguiente instrucción para añadir Git y SSH al `PATH` de tu ventana de comandos: `setx PATH "%PATH%;C:\Program Files\Git\bin"` . Reinicia tu ventana de comandos para habilitar el cambio.

> Después de reiniciar tu ventana de comandos, ¡no olvides ir a tu carpeta `djangogirls` otra vez y activar tu entorno virtual! (Pista: Comprueba el capítulo de instalación de Django)

Crea también, por favor, una cuenta gratuita de Heroku aquí: https://id.heroku.com/signup/www-home-top

Luego autentica tu cuenta de Heroku en tu ordenador ejecutando el siguiente comando:

```
$ heroku login
```

En caso de que no tengas una clave SSH, este comando generará una automáticamente. Las claves SSH se necesitan para hacer *push* de código a Heroku, es decir, enviar tu código al servidor.

Git commit

Heroku usa git para sus despliegues. A diferencia de PythonAnywhere, puedes hacer un *push* a Heroku directamente, sin utilizar Github como intermediario. Pero necesitamos hacer unas pequeñas modificaciones antes.

Abre el archivo llamado `.gitignore` en tu directorio `djangogirls` y añade `local_settings.py` al mismo. Queremos que git ignore `local_settings`, para que se quede en nuestro ordenador local y no sea subido a Heroku.

```
*.pyc
db.sqlite3
myvenv
__pycache__
local_settings.py
```

Y hacemos un commit de nuestros cambios

```
$ git status
$ git add -A .
$ git commit -m "archivos adicionales y cambios para Heroku"
```

Selecciona un nombre para la aplicación

Estamos haciendo nuestro blog disponible en la Web en `[nombre de tu blog].herokuapp.com`, por lo que necesitamos seleccionar un nombre que nadie más esté usando. Este nombre no tiene que estar relacionado con nuestra aplicación `blog` de Django o con `mysite` ni con nada de lo que hemos creado hasta ahora. El nombre puede ser cualquier cosa que tú quieras, pero Heroku es bastante estricto con respecto a los caracteres que puedes usar: sólo se permiten letras minúsculas simples (no mayúsculas ni acentos), números y guiones (`-`).

Una vez que hayas pensado en un nombre (quizás algo con tu nombre o tu apodo), ejecuta este comando, reemplazando `djangogirlsblog` con el nombre seleccionado:

```
$ heroku create djangogirlsblog
```

> **Nota**: Recuerda reemplazar `djangogirlsblog` con el nombre de tu aplicación en Heroku.

Si no se te ocurre ningún nombre, también puedes ejecutar simplemente:

```
$ heroku create
```

y Heroku elegirá un nombre no usado para ti (probablemente algo como `enigmatic-cove-2527`).

Si en algún momento te apetece cambiar el nombre de tu aplicación en Heroku, puedes hacerlo con este comando (reemplaza `nombre-nuevo` con el nuevo nombre que quieras usar):

```
$ heroku apps:rename nombre-nuevo
```

> **Nota**: Recuerda que después de cambiar el nombre de la aplicación, tienes que visitar `[nombre-nuevo].herokuapp.com` para ver tu sitio web.

¡Desplegar en Heroku!

Eso fue un montón de configuración e instalación, ¿verdad? ¡Pero sólo necesitas hacerlo una vez! ¡Ahora ya puedes desplegar!

Cuando ejecutaste `heroku create` , automáticamente se añadió el Heroku *remote* para la aplicación en nuestro repositorio. Ahora podemos hacer un simple git push para desplegar nuestra aplicación:

```
$ git push heroku master
```

> **Nota**: Esto probablemente producirá un *montón* de salida por consola la primera vez que lo ejecutas, ya que Heroku compila e instala psycopg. Sabrás que has tenido éxito si ves algo como `https://tunombredeaplicacion.herokuapp.com/ deployed to Heroku` hacia el final del texto.

Visita tu aplicación

Has desplegado tu código en Heroku, y especificado el tipo de proceso en un `Procfile` (nosotros elegimos tipo de proceso `web`). Ahora podemos decirle a Heroku que inicie este `proceso web` .

Para hacerlo, ejecuta el siguiente comando:

```
$ heroku ps:scale web=1
```

Este comando le dice a Heroku que ejecute solamente una instancia de nuestro proceso `web`. Ya que nuestra aplicación de blog es bastante simple, no necesitamos demasiada potencia y por tanto está bien ejecutar solamente un proceso. Es posible pedirle a Heroku que ejecute más procesos (por cierto, Heroku le llama a estos procesos "Dynos" así que no te sorprendas si ves ese nombre) pero entonces dejaría de ser gratuito.

Ahora podemos visitar la aplicación en nuestro navegador con `heroku open`.

```
$ heroku open
```

> **Nota**: ¡Verás una pagina de error! Hablaremos de eso en un minuto.

Esto abrirá una url como https://djangogirlsblog.herokuapp.com/ en tu navegador, y al momento verás probablemente una página de error.

El error que ves es porque cuando desplegamos en Heroku creamos una nueva base de datos que está vacía. Necesitamos ejecutar los comandos `migrate` y `createsuperuser`, tal como hicimos en PythonAnywhere. Esta vez, se ejecutan vía una línea de comandos especial en nuestro ordenador, `heroku run`:

```
$ heroku run python manage.py migrate

$ heroku run python manage.py createsuperuser
```

Este comando te pedirá que elijas un nombre de usuario y password otra vez. Éstas serán tus credenciales en la página de administración de tu sitio web.

Refresca tu navegador, y ¡ahí está! Ahora ya sabes como desplegar en dos plataformas de hosting diferentes. Elige tu favorita :)