Tutorial Extensions

django girls

Tabla de contenido

Tutorial Django Girls : Extensiones

Introducción

Este libro contiene tutoriales adicionales que puedes realizar luego de terminal el Tutorial de Django Girls.

Los tutoriales actuales son:

- Tarea: agregar mas a tu sitio web
- Tarea: Asegura tu sitio
- Tarea: Crea un modelo de comentarios
- Opcional: Instalación de PostgreSQL

Contribuir

Este tutorial es mantenido por DjangoGirls. Si encuentras algunos errores o quieres actualizar el tutorial, por favor Sigue la guía de contribución.

Tarea: Agregar mas a tu sitio web

Nuestro blog ha recorrido un gran camino hasta aquí, pero aún hay espacio para la mejora. Lo siguiente, vamos a agregar nuevas características para borradores y su publicación. También vamos a agregar eliminación de post que ya no queremos mas.

Guarda nuevos post como borradores

Actualmente, nosotros creamos nuevos post usando nuestro formulario *New Post* y el post es publicado directamente. En lugar de publicar el post vamos a guardarlos com borradores. **elimina** esta linea en `blog/views.py` en los méetodos `post_new` y `post_edit`:

```
post.published_date = timezone.now()
```

De esta manera los nuevos post serán guardados como borradores y se podrán revisar después en vez de ser publicados instantáneamente. Todo lo que necesitamos es una manera de listar los borradores. ¡Vamos a hacerlo!

Página con los post no publicados

¿Recuerdas el capítulo sobre los querysets? Creamos una vista `post_list` que muestra solamente los post publicados (aquellos que tienen un `publication_date` no vacío).

Es tiempo de hacer algo similiar, pero con borradores.

Vamos a añadir un enlace en `blog/templates/blog/base.html` en el encabezado. No queremos mostrar nuestro borradores a todo el mundo, entonces vamos a colocarlo dentro de la verificación `{% if user.is_authenticated %}`, justo después del botón de agregar posts.

```
<a href="{% url 'post_draft_list' %}" class="top-menu"><span class="glyphicon glyphicon-edit"></span></a>
```

Siguiente: ¡urls! en `blog/urls.py` vamos a agregar:

```
path('drafts/', views.post_draft_list, name='post_draft_list'),
```

Tiempo de crear una nueva vista en `blog/views.py`

```
def post_draft_list(request):
    posts = Post.objects.filter(published_date__isnull=True).order_by('created_date')
    return render(request, 'blog/post_draft_list.html', {'posts': posts})
```

Esta línea `posts = Post.objects.filter(published_date__isnull=True).order_by('created_date')` se asegura de que solamente vamos a tomar post no publicados (`published_date__isnull=True`) y los ordena por `created_date` (`order_by('created_date')`).

Ok, el último pedazo es la plantilla. Crea un archivo `blog/templates/blog/post_draft_list.html` y agrega lo siguiente:

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

{% block content %}
    {% for post in posts %}
        <div class="post">
            <p class="date">created: {{ post.created_date|date:'d-m-Y' }}</p>
            <h1><a href="{% url 'post_detail' pk=post.pk %}">{{ post.title }}</a></h1>
            <p>{{ post.text|truncatechars:200 }}</p>
        </div>
    {% endfor %}
{% endblock %}
```

Se ve muy similar a nuestr `post_list.html` ¿verdad?

Ahora cuando vayas a `http://127.0.0.1:8000/drafts/` vas a ver la lista de post no publicados.

¡Bien! ¡Nuestra primera tarea hecha!

Agrega un botón de publicación

Sería bueno tener un botón en el detalle del post que publique el post inmediatamente, ¿no?

Vamos a abrir `blog/templates/blog/post_detail.html` y cambiar estas líneas:

```
{% if post.published_date %}
    <div class="date">
        {{ post.published_date }}
    </div>
{% endif %}
```

por estas:

```
{% if post.published_date %}
    <div class="date">
        {{ post.published_date }}
    </div>
{% else %}
    <a class="btn btn-default" href="{% url 'post_publish' pk=post.pk %}">Publish</a>
{% endif %}
```

Como puedes ver, hemos agregado la línea `{% else %}` . Esto significa, que la condición de `{% if post.published_date %}` no es cumplida (entonces no hay `publication_date`), entonces queremos agregar la línea `<a class="btn btn-default" href="{% url 'post_publish' pk=post.pk %}">Publish</a>` . Nota que estamos pasando la variable `pk` en el `{% url %}` .

Tiempo de crear una URL (en `blog/urls.py`):

```
path('post/<pk>/publish/', views.post_publish, name='post_publish'),
```

Y finalmente una *vista* (como siempre, en `blog/views.py`):

```
def post_publish(request, pk):
    post = get_object_or_404(Post, pk=pk)
    post.publish()
    return redirect('post_detail', pk=pk)
```

Recuerda, cuando creamos el modelo `Post` escribimos un método `publush` . Se veía como esto:

```
def publish(self):
    self.published_date = timezone.now()
    self.save()
```

¡Ahora finalmente podemos usarlo!

Y de nuevo al publicar el post, ¡somos redirigidos inmediatamente a la página `post_detail` !

¡Felicitaciones! ¡Casi terminas, el último paso es un botón de eliminar!

Eliminar post

Vamos a abrir `blog/templates/blog/post_detail.html` de nuevo y vamos a añadir esta línea

```
<a class="btn btn-default" href="{% url 'post_remove' pk=post.pk %}"><span class="glyphicon glyphicon-remove"></span>
</a>
```

Justo debajo de la línea co el botón editar.

Ahora necesitamos una URL (`blog/urls.py`):

```
path('post/<pk>/remove/', views.post_remove, name='post_remove'),
```

Ahora, ¡Tiempo para la vista! Abre `blog/views.py` y agrega este código:

```
def post_remove(request, pk):
    post = get_object_or_404(Post, pk=pk)
    post.delete()
    return redirect('post_list')
```

La única cosa nueva es en realidad, eliminar el post. Cada modelo en django puede ser eliminado con `.delete()` . ¡Así de simple!

Y en este momento, después de eliminar un post, queremos ir a la página web con la lista de posts, por eso usamos el `redirect` .

¡Vamos a probarlo! ¡Vamos a la página con los post e intentemos eliminarlos!

Si, ¡esto es lo último! ¡Acabas de completar este tutorial! ¡Eres asombroso!

Tarea: Asegura tu sitio

Puedes haberte dado cuenta que no usaste tu contraseña, además de cuando iniciaste en la interfaz de administrador. También debes haber notado que esto significa que cualquiera puede editar tus post en tu blog. No se tu, pero a mi no me gustaría que cualquiera editara mi blog, así que vamos a hacer algo al respecto.

Autorizando la edición y creación de post.

Primero vamos a hacer las cosas seguras. Vamos a proteger nuestras vistas `post_new`, `post_edit`, `post_draft_list`, `post_remove` y `post_publish` para que solo usuarios que hayan ingresado puedan acceder a ellas. Django nos da algunas ayudas para hacer esto, llamados *decoradores*. No te preocupes por las tecnicalidades ahora; puedes leer sobre eso después. El decorador que vamos a usar está en el módulo `django.contrib.auth.decorators` y es llamado `login_required`.

Entonces editemos tu `blog/views.py` y agrega estas lineas al comienzo del archivo junto con las demás importaciones:

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

Entonces añade la línea antes de cada una de las vistas `post_new`, `post_edit`, `post_draft_list`, `post_remove` y `post_publish`, (decorándolas) como a continuación:

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

¡Eso es todo! Ahora intenta acceder a `http://localhost:8000/post/new/. ¿Notas la diferencia?

> Si obtienes un formulario vacío, posiblemente sigues logeado desde el capítulo en la interfaz de administrador. Ve a `http://localhost:8000/admin/logout/` para salir, y luego `http://localhost:8000/post/new` de nuevo.

Puedes obtener uno de nuestros amados errores. Esto es muy interesante: El decoradr que añadimos nos redireccionará a la página de ingreso, pero como no está disponible, retorna un "Page not found (404)".

No te olvides del decorador encima de `post_edit`, `post_remove`, `post_draft_list` y `post_publish` también.

!Perfecto!, !hemos alcanzado parte de nuestro objetivo! Ahora otras personas no podrán crear post en nuestro blog. Desafortunadamente, nosortos tampoco podemos crearlos. Así que vamos a arreglarlo a continuación.

Ingresar usuarios

Ahora podemos intentar hacer muchas cosas mágicas para implementar usuarios y contraseñas y autenticación, pero hacer esto correctamente es complicado. Como django viene con "baterías incluidas", alguien ya ha hecho el trabajo duro por nosotros, así que vamos a utilizarlas.

En `mysite/urls.py` agrega una nueva entrada `path('accounts/login/', views.login, name='login')`. El contenido del archivo debería verse similar a:

```
from django.urls import include, path
from django.contrib import admin

from django.contrib.auth import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/login/', views.login, name='login'),
    path('', include('blog.urls')),
]
```

Luego necesitamos agregar una plantilla para la página de ingreso, así que crearemos el directorio `blog/templates/registration` y dentro un archivo llamado `login.html`.

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

{% block content %}
    {% if form.errors %}
        <p>Your username and password didn't match. Please try again.</p>
    {% endif %}

    <form method="post" action="{% url '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 también hace uso de la plantilla *base* para mantener el estilo de tu blog.

La cosa buena quí es que funciona. No necesitamos lidiar con el manejo del for o las contraseñas y asegurarlas. Solamente una cosa mas para hacer. Entonces vamos a agregar esta configuración en `mysite/settings.py`:

```
LOGIN_REDIRECT_URL = '/'
```

Entonces cuando la página es accesada directamente, la redirecionará a la página de primer nivel, el index (la página de inicio de blog).

Mejorando el diseño

Ya definimos como autorizar usuarios. Vemos los botones para añadir post. Ahora queremos asegurarnos que el botón de ingreso le aparezca a todos.

Vamos a agregar el botón de ingreso así:

```
    <a href="{% url 'login' %}" class="top-menu"><span class="glyphicon glyphicon-lock"></span></a>
```

Para esto necesitamos editar la plantilla, así que vamos a abrir `blog/templates/blog/base.html` y cambiarlo, así que la parte en medio de `<body>` se verá así:

```
<body>
    <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 glyphicon-edit"></span></a>
        {% else %}
            <a href="{% url 'login' %}" class="top-menu"><span class="glyphicon glyphicon-lock"></span></a>
        {% endif %}
        <h1><a href="/">Django Girls Blog</a></h1>
    </div>
    <div class="content container">
        <div class="row">
            <div class="col-md-8">
            {% block content %}
            {% endblock %}
            </div>
        </div>
    </div>
</body>
```

Debes reconocer el patrón aquí. Hay una condición `if` en la plantilla que verifica si el usuario está autenticado para mostrar los botones de agregar y editar. De otra manera muestra el botón de ingreso.

Más para usuarios autenticados

Vamos a agregarle algo de dulce a nuestras plantillas mientras estamos ahí. Primero vamos a mostrar algunos detalles de cuando ingresamos. Editemos el archivo `blog/templates/blog/base.html` así:

```
<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 glyphicon-edit"></span></a>
        <p class="top-menu">Hello {{ user.username }} <small>(<a href="{% url 'logout' %}">Log out</a>)</small></p>
    {% else %}
        <a href="{% url 'login' %}" class="top-menu"><span class="glyphicon glyphicon-lock"></span></a>
    {% endif %}
    <h1><a href="/">Django Girls Blog</a></h1>
</div>
```

Esto agrega un lindo "Hello *<username>*" para recordarle al usuario como ingresó, y que está autenticado. También agrega un enlace de salida del blog -- como puedes ver, aún no funciona. ¡Vamos a arreglarlo!

Decidimos apoyarnos en Django para manejar el inicio de sesión, así que vamos ver si Django nos permite manejar el cierre de la sesión. Mira https://docs.djangoproject.com/en/2.0/topics/auth/default/ y ve si encuentras algo.

¿Terminaste de leer? Por ahora vamos a pensar en agregar una URL en `mysite/urls.py` apuntando a la vista de salida (`django.contrib.auth.views.logout`) así:

```
from django.urls import include, path
from django.contrib import admin

from django.contrib.auth import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/login/', views.LoginView.as_view(), name='login'),
    path('accounts/logout/', views.LogoutView.as_view(next_page='/'), name='logout'),
    path('', include('blog.urls')),
]
```

¡Eso es todo! Si has segido todo lo anterior, en este punto (e hiciste la tarea), tu blog ahora:

- necesita un nombre de usuario y contraseña para ingresar,

- necesita haber ingresado para agregar, editar, publicar o eliminar posts
- puede salir de nuevo

Tarea: Crea un modelo de comentarios

Actualmente, solamente tenemos un modelo de Post, ¿Qué si queremos recibir retro alimentación de tus lectores y dejarlos comentar?

Crea un modelo de comentarios para el blog

Vamos a abrir `blog/models.py` y pega esta pieza de código al final del archivo:

```
class Comment(models.Model):
    post = models.ForeignKey('blog.Post', on_delete=models.CASCADE, 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 ir atrás al modelo **Modelos en Django** de este tutorial si necesitas refrescar lo que cada tipo de campo significa.

En esta extensión del tutorial vamos a tener un nuevo tipo de campo:

- `models.BooleanField` - this is true/false field.

La opción `related_name` en `models.ForeignKey` nos permite acceder acceso a los comentarios desde el modelo Post.

Creando tablas para los modelos en tu base de datos.

Es tiempo de agregar nuestro modelo de comentarios a la base de datos. Para hacer esto le vamos a decir a Django que haga los cambios a nuestro modelo. Escribe `python manage.py makemigrations blog` en tu linea de comendos. Deberías ver algo como esto:

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

Como puedes ver, se ha creado otro archivo de migración en la carpeta `blog/migrations/` . Ahora necesitamos aplicar estos cambios escribiendo `python manage.py migrate blog` en la línea de comandos. La salida 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 de comentarios existe ahora en la base de datos! ¿No sería genial acceder a el desde nuestro panel de administración?

Registra el modelo de comentarios en el panel de administrador

Para registrar el modelo de Comentario en el panel de administrador, ve a `blog/admin.py` y agrega esta línea:

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

Justo debajo de esta línea:

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

Recuerda que es importante importar el modelo de comentarios al comienzo del archivo, 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 la línea de comandos y vas a http://127.0.0.1:8000/admin/ en tu navegador, ahora podras acceder a la lista de comentarios, y también tendrás la posibilidad de agregar y eliminar comentarios. ¡Juega con la nueva característica de comentarios!

Has tus comentarios visibles

Ve al archivo `blog/templates/blog/post_detail.html` y agrega el siguiente código antes de la etiqueta `{% 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 puedes ver la sección de comentarios en los detalles del post.

Pero puede verse un poco mejor, así que vamos a agregar algún css al final del archivo `static/css/blog.css` :

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

También vamos a dejar a los visitantes sobre los comentarios que dejan en la página. Ve al archivo `blog/templates/blog/post_list.html` y agrega una lína como esta:

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

Después de esto, tu 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 'post_detail' pk=post.pk %}">{{ post.title }}</a></h1>
            <p>{{ post.text|linebreaksbr }}</p>
            <a href="{% url 'post_detail' pk=post.pk %}">Comments: {{ post.comments.count }}</a>
        </div>
    {% endfor %}
{% endblock content %}
```

Deja a tus lectores escribir comentarios

Ahora podemos ver los comentarios hechos en nuestro blog, pero no podemos agregarlos. ¡Vamos a cambiar eso!

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

```
class CommentForm(forms.ModelForm):

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

Recuerda importar el modelo Comentario, cambiando la línea:

```
from .models import Post
```

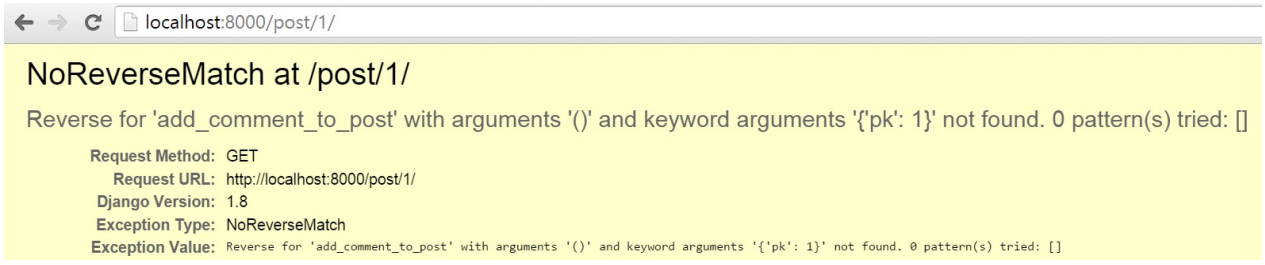
en:

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

Ahora vamos 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>
```

Si quieres ir a los detalles del post, deberías ver este error:



¡Ahora sabemos como arreglarlo! Ve a `blog/urls.py` y agrega esto a `urlpatterns`:

```
url(r'^post/(?P<pk>\d+)/comment/$', views.add_comment_to_post, name='add_comment_to_post'),
```

¡Refresca la página y ahora tenemos un erro diferente!

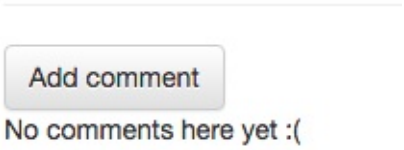
Para agregar este error, agrega esto a `blog/views.py` :

```
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('post_detail', pk=post.pk)
    else:
        form = CommentForm()
    return render(request, 'blog/add_comment_to_post.html', {'form': form})
```

Recuerda importar `CommentForm` al comienzo del archivo:

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

Ahora vamos a los detalles de la página y deberíamos ver el botón "Add Comment":



Sin embargo, cuando des click en el botón verás:

Como el error nos dice, la plantilla no existe aún. Entonces vamos a crear una en `blog/templates/blog/add_comment_to_post.html` y agregar el siguiente código:

```
{% 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 %}
```

¡Genial! !Ahora nuestroe lectores puede agregar lo que piensan en nuestros post!

Moderando los comentarios

No todos los comentarios deberían ser mostrados. Como dueño del blog, posiblemtente quieras la opción de aprovar 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 %}
```

a:

```
{% 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 gly
phicon-remove"></span></a>
                <a class="btn btn-default" href="{% url 'comment_approve' pk=comment.pk %}"><span class="glyphicon gl
yphicon-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 un `NoReverseMatch`, porque no hay ninguna URL que concuerde con `comment_remove` y `comment_approve` aún

Para arreglar este error, agregemos estas urls a `blog/urls.py`:

```
url(r'^comment/(?P<pk>\d+)/approve/$', views.comment_approve, name='comment_approve'),
url(r'^comment/(?P<pk>\d+)/remove/$', views.comment_remove, name='comment_remove'),
```

Ahora debereias ver un `AttributeError`. Para arreglar este error, agrega las siguiente vistas en `blog/views.py`:

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

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

Necesitas importar `Comment` al comienzo del archivo:

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

¡Todo funciona! Hay un pequeño cambio que podemos hacer. en nuestra página de lista -- debajo de posts -- actualmente vemos el número de los comentarios que el post ha recibifo. Vamos a cambiar esto para ver el número de comentarios aprobados.

Para arreglar esto, ve a `blog/templates/blog/post_list.html` y cambia la línea:

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

a:

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

finalmente, añade el método al modelo `Post` en `blog/models.py`:

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

Ahora puedes ver la característica de comentarios finalizada ¡Felicitaciones! :-)

Opcional: Instalación de PostgreSQL

> Parte de este capítulo está basado en tutoriales de Geek Girls Carrots (http://django.carrots.pl/).
>
> Parte de este capítulo está basado en django-marcador tutorial Licenciado bajo la licencia Creative Commons Attribution-ShareAlike 4.0. La licencia del tutorial le pertenece a Zapke-Gründemann.

Windows

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

Escoge 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 donde se instaló porque lo necesitarás en el siguiente paso (tipicamente, es `C:\Program Files\PostgreSQL\9.3).

Mac OS X

La manera mas fácil es descargar gratis Postgres.app e instalalo como cualquier otra aplicación en tu sistema operativo.

Descargalo, y arrástralo al directorio de Aplicaciones, y ejecutalo dando doble click en el. ¡Eso es todo!

Ahora debes agregar las herramientas de línea de comandos de Postgres a tu variable `PATH`, que es descrita aquí.

Linux

Los pasos de instalación varían de distribucióna distribución. A continuación encontraremos comandos para Ubuntu y Fedora, pero si estás usando una distribución diferente mira 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
```

Crea la base de datos

Lo siguiente será crear nuestra primera base de datos, y un usuario que pueda acceder a ella. PostgreSQL nos deja crear tantas bases de datos como usuarios queramos, entonces si vamos a ejecutar mas de un sitio, deberíamos crear una base de datos para cada uno.

Windows

Si estás usando Windows, aquí hay un par de pasos adicionales que necesitas completas. Por ahora no es importante si no entiendes la configuración que estamos haciendo aquí, pero sientente libre de preguntarle a tu Coach si estas curioso de lo que estás haciendo.

1. Abre la línea de comandos (Inicio → Todos los programas → Accesorios → Linea de comandos)
2. Ejecuta la siguiente configuración y presiona enter: `setx PATH "%PATH%;C:\Program Files\PostgreSQL\9.3\bin"` . Puedes pegar cosas a la línea de comandos dando click derecho y seleccionando `Pegar` . Asegúrate de que el la carpeta sea la misma que anotaste cuando estabas instalando con un `\bin` al final. Deberías ver el mensaje `SUCCESS: Specified value was saved.` .
3. Cierra y vuelve a abrie la línea de comando

Crea la base de datos

Primero, vamos a iniciar la consola dePostgres ejecutando `psql` . ¿Recuerdas como lanzar la consola?

> En Mac OS X debes hacer esto lanzando la aplicación `Terminal` (está en Aplicaciones → Utilidades). En linux, posiblemente esté bajo Aplicaciones → Accesorios → Terminal. En Windows necesitas ir a Inicio → Todos los programas → Accesorios → Línea de Comandos. Mas allá, en Windows, `psql` requiere que ingreses con un nombre de usuario y contraseña, el que escogiste durante la instalación. Si `psql` está pregunando por una contras ´ña y no parece funcionar, trata `psql -U <username> -W` primero y luego presioa Enter.

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

Nuestro `$` ahora ha cambiado a `#` , lo cual significa que ahora estamos enviando comandos a PostgreSQL. Vamos a crear un usuario con `CREATE USER name;` (recuerda usar el punto y coma):

```
# CREATE USER name;
```

Reemplaza `nombre` con tu propio nombre. No deberías usar letras acentadas o espacios en blanco. (ejemplo `bożena maria` es inválida - necesitas convertirla en `bozena_maria`). Si esto va bien, puedes tener una respuesta `CREATE ROLE` de la consola.

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

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

Recuerda reemplazar `name` con el nombre que hayas escogido (ejemplo `bozena_maria`). Esto crea una base de datos vacía que puedes ahora usar en tus proyectos. Si esto va bien, puedes obtener una respuesta `CREATE DATABASE` de la consola.

¡Genial - las bases de dates están en orden!

Actualizar la configuración

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 reemplazalo con esto:

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

Recuerda cambiar `name` con el nombre de usuario que creaste antes en este capítulo.

Instalando el paquete PostgreSQL para Python

Primero, instala `Heroku Toolbelt` desde https://toolbelt.heroku.com/ porque lo necesitaremos mas adelante para desplegar tu sitio, esto también include Git, lo cual puede ser util.

Lo siguiente es instalar un paquete que le permita a Python hablar con PostgreSQL - este es llamado `psycopg2`. Las intrucciones de instalación difieren un poco entre Windows y Linux/OS X.

Windows

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

Asegúrate de que tienes la versión de Python correspondiente (3.4 debería ser la última línea) y la arquitectura correcta (32 bit en la columna de la izquiera o 64 bit en la columna derecha).

Renombra el archivo descargado y muévelo para que estédisponible en `C:\psycopg2.exe`.

Una vez esté terminado, ejecuta el siguiente comando en la terminal (asegúrate que el `virtualenv` esté activado):

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

Linux y OS X

Ejecuta el siguiente código en la consola:

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

Si eso sale bien, verás algo así:

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

---
```

Una vez esto esté completo, ejecuta `python -c "import psycopg2"`. Si no tienes ningún error, todo está instalado correctamente.

Aplicando las migraciones y creando un super usuario

En orden de usar la base de datos recién creada en nuestro proyecto, necesitarás aplicar las migraciones en tu ambiente virtual ejecutando el siguiente código:

```
(myvenv) ~/djangogirls$ python manage.py migrate
```

Para agregar nuevos post en nuestro blog, también necesitas crear un super usuario, ejecutando el siguiente código:

```
(myvenv) ~/djangogirls$ python manage.py createsuperuser --username name
```

Recuerda reemplazar `name` con el nombre de usuario. Se te preguntará por un email y una contraseña.

Ahora puedes ejecutar el servidor. Entra en la aplicación con la cuenta de super usuario y comienza a agregar posts a tu nueva base de datos.

Dominio

PythonAnywhere nos da un dominio gratuito, pero tal vez tu no quieres tener un ".pythonanywhere.com" al final de tu URL de blog. Quizas tu quieres tener ago como "www.infinite-kitten-pictures.org" o "www.3d-printed-steam-engine-parts.com" o "www.antique-buttons.com" o "www.mutant-unicornz.net", o lo que quieras.

Aquí vamos a hablar un poco sobre como obtener un dominio, y como enlazarlo con la aplicación en PythonAnywhere. Sin embargo, debes saber que los dominios cuestan dinero, y PythonAnywhere también te cobra mensualmente una tarifa para usar tu propio nombre de dominio -- no es mucho dinero en total, pero esto es posiblemente que solamente quieras si estas realmente comprometido.

¿Dónde registrar un dominio?

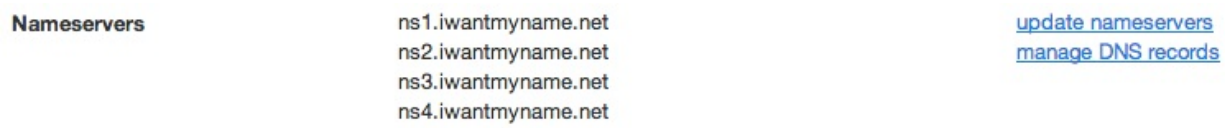
Un nombre de dominio cuesta típicamente $15 USD al año. Hay opciones mas baratas y mas caras, dependiendo del probeedor. Existen muchas compañías con las que puedes comprar un dominio: una simple busqueda en google te dará muchas opciones.

Nuestra favorita es I want my name. Su anuncion dice "manejo de dominio sin dolor" y realmente es sin dolor.

También puedes obtener dominios gratis. dot.tk es un lugar para obtener uno, pero debes estar conciente que los dominios gratuitos a veces se sienten baratos -- si tu sitio quiere ser para un negocio profesional, deberías pensar sobre pagar por un dominio "propio" que termine en `.com`.

¿Cómo apuntar tu dominio a PythonAnywhere?

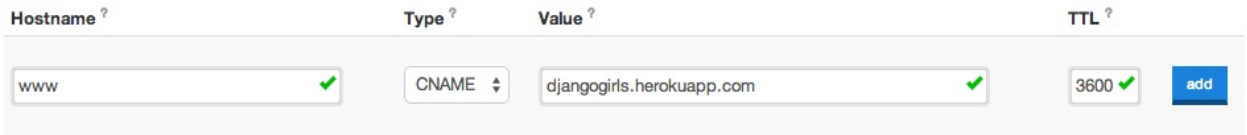
Si fuiste a través de *iwantmyname.com*, da click en `Domains` en el menú y escoge tu dominio recién comprado. Entonces localiza y da click en `manage DNS records`:



Ahora necesitas localizar este formulario:

Y llena la siguiente información

- Hostname: www
- Type: CNAME
- Value: your domain from PythonAnywhere (for example djangogirls.pythonanywhere.com)
- TTL: 60



Click en el botón `Add` y Guarda los cambios al final.

> Si quieres un proveedor de dominio distinto, la interfaz para encontrar tu configuración de DNS/ CNAME será diferente, pero el objetivo es el mismo: Definir un CNAME que apunte tu nuevo dominio a `nombredeusuario.pythonanywhere.com`.

También puede tomar algunos minutos para que tu dominio comience a funcionar. ¡Se paciente!

Configura el dominio a través de la aplicación en PythonAnywhere.

También necesitas decirle a PythonAnywhere que quieres utilizar tu dominio personalizado.

Ve a la página de cuenta de PythonAnywhere y mejora tu cuenta. La opcieon mas barata (el plan "Hacker:) está bien para comenzar, y siempre puedes mejorarlo después cuando te vuelvas super famoso y tengas millones de visitas.

Siguiente, ve sobre la pestaña Web y anota un par de cosas:

- Copia la **ruta a tu virtualenv** y colocala en un lugar seguro.
- Da click a través de tu **archivo de configuración wsgi**, copia el contenido y pegalo en un lugar seguro.

Siguiente, **elimina** tu vieja aplicación. No te preocupes, esto no elimina nada de tu código, solamente apaga el dominio en `nombredeusuario.pythonanywhere.com`. Siguiente, crea una nueva aplicación y sigue estos pasos:

- Ingresa tu nombre de dominio
- Escoge "configuración manual"
- Selecciona Python 3.4
- Y estamos listos

Cuando seas enviado de vuelta a la pestaña web

- Pega la ruta del ambiente virtual que guardaste antes.
- Da click através del archivo de configuración, y pega el contenido de tu archivo de configuración viejo.

Da click en actualizar web app y ¡deberías encontrar tu sitio corriendo en tu nuevo dominio!

Si tienes problemas, da click en "Send Feedback" en el sitio PythonAnywhere, y uno de sus amables administradores te ayudará a solucionarlo en poco tiempo.

Despliega tu sitio en Heroku (También como PythonAnywhere)

Siempre es bueno como un desarrollador tiene un par de diferentes opciones de despliegue bajo su cinturón. ¿Por qué no tratar de desplegar tu sitio en Heroku, también como PythonAnywhere?

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

Te guiaremos por el siguiente tutorial: https://devcenter.heroku.com/articles/getting-started-with-django, pero pegamos aquí para hacerlo mas fácil para ti.

El archivo `requirements.txt`

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

Pero primero, Heroku necesita que instalemos algunos paquetes. Ve a tu consola con tu `virtualenv` activado y escribe:

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

Después que la instalación está finalizada, ve a la carpeta `djangogirls` y ejecuta este comando:

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

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

> **Nota**: `pip freeze` imprime la lista de todas las librerías instaladas en tu ambiente virtual, y el `>` toma la salida de `pip freeze` y la coloca en un archivo. ¡Trata ejecutar `pip freeze` sin el `< requirements.txt` para ver lo que pasa!

Abre este archivo y añada esto al final:

```
psycopg2==2.6.2
```

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

Procfile

Otra cosa que Heroku necesita es un `Procfile`. Esto le dice a Heroku que comandos ejecutar para iniciar nuestro sitio web. Ve a tu editor de código y crea un archivo llamado `Procfile` en la carpeta `djangogirls` y agrega esta línea:

```
web: gunicorn mysite.wsgi --log-file -
```

Esta línea significa que vas a desplegar una aplicación `web` y lo vas a hacer ejecutando el comando `gunicorn mysite.wsgi` (`gunicorn` es un programa que es la versión mas poderosa del comando de django `runserver`).

Entonces guardalo. ¡Listo!

El archivo `runtime.txt`

También necesitamos decirle a Heroku que versión de Python vamos a usar. Esto es hecho creando un archivo `runtime.txt` en la carpeta `djangogirls` usando tu editor de texto y colocando el siguiente texto (y nada mas) dentro:

```
    python-3.5.2
```

`mysite/local_settings.py`

Porque es mas restrictivo que PythonAnywhere, Heroku quiere usar diferentes configuraciones de las que nosotros usamos localmente (en nuestro computador). Heroku quiere que usar Postgres mientras que nosotros usamos SQLite, por ejemplo. Por eso necesitamos crear un archivo separado para las configuraciones que estarán disponibles en nuestro ambiente local.

Ve adelante y crea un archivo `mysite/local_settings.py`. Este debe contener tu configuración de `DATABASE` de tu archivo `mysite/settings.py`. Justo como esto:

```
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
```

¡Luego guardalo! :)

mysite/settings.py

Otra cosa que debemos hacer es modificar nuestro archivo de sitio web `settings.py`. Abre `mysite/settings.py` en tu editor y cambia las siguientes líneas:

```
import dj_database_url

...

DEBUG = False

ALLOWED_HOSTS = ['127.0.0.1', '.herokuapp.com']

...

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

...

db_from_env = dj_database_url.config(conn_max_age=500)
DATABASES['default'].update(db_from_env)
```

Esto hará la configuración necesaria para Heroku.

Luego guarda el archivo.

mysite/wsgi.py

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

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

¡Todo bien!

Cuenta Heroku

Necesitas instalar `Heroku toolbelt` el cual encontrarás aquí (puedes saltarte la instalación si ya lo instalaste durante la configuración): https://toolbelt.heroku.com/

> Cuando ejecutas la instalación de Heroku toolbelt en windows asegúrate de escoger "Instalación personalizada" cuando te pregunten que componentes instalar. En la lista de componentes que se muestra luego por favor selecciona "Git y SSH"
>
> En windows también debes ejecutar el siguiente comando para agregar Git y SSH a tu `PATH` de la línea de comanods: `setx PATH "%PATH%;C:\Program Files\Git\bin"` . Reinicia la línea de comandos después para habilitar el cambio.
>
> ¡Después de reiniciar tu línea de comandos, no olvides ir a tu carpeta `djangogirls` de nuevo y habilitar el ambiente virtual! (Truco: Ve al capítulo de instalación de Django)

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

Luego autenticate con tu cuenta Heroku en tu computador ejecutando este comando:

```
$ heroku login
```

En cas que no tengas una llave SSH este comando automáticamente creará una. Las llaves SSH son requeridas para colocar código en Heroku.

Git commit

Heroku usa git para sus propios despliegues. A diferencia de PythonAnywhere, puedes hacer push a Heroku directamente, sin Github. Pero necesitamos hacer un par de cosas antes.

Abre un archivo llamado `.gitignore` en tu carpeta `djangogirls` y agrega `local_settings.py` a el. Queremos que git ignore `local_settings` , entonces este permanece en nuestro computador local y no termina en Heroku. Open the file named `.gitignore` in your `djangogirls` directory and add `local_settings.py`

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

Y guarda los cambios

```
$ git status
$ git add -A .
$ git commit -m "additional files and changes for Heroku"
```

Escoge un nombre de aplicación

Vamos a hacer que tu blog esté disponible en la web en `[tu nombre de blog].herokuapp.com` así vamos a escoger un nombre que nadie mas haya tomado. Este nombre no tiene que estar relacionado con el blog de `Django` o `mysite` o nada de lo que hemos creado hasta ahora. El nombre debe estar en minúscula (sin letras mayúsculas o acentos), nombres y guiones (`-`).

Una ves tengas un nombre (tal vez algo con tu nombre o nick en el), ejecuta este comando, reemplazaod `djangogirlsblo` con tu propio nombre de aplicación:

```
$ heroku create djangogirlsblog
```

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

Si no puedes pensar en un nombre, puedes en lugar ejecutar:

```
$ heroku create
```

Y heroku escogerá un nombre no disponible por ti (posiblemente algo como `enigmatic-cove-2527`).

Si tu alguna vez sientes que debes cambiar el nombre de tu aplicación Heroku, puedes hacerlo en cualquier momento con est eomando (reemplaza `the-new-game` con el nuevo nombre que quieres usar):

```
$ heroku apps:rename the-new-name
```

> **Nota**: Recuerda que luego que cambias el nombre de tu aplicación, necesitas visitar `[the-new-name].herokuapp.com` para ver tu sitio

¡Despliega a Heroku!

Esto fue mucho de configuración e instalación, ¿cierto? ¡Pero esto solo lo necesitas hacer una vez! ¡Ahora puedes desplegar!

Cuando tu ejecutas `heroku create` , el automáticamente agregará un `remote` de Heroku a nuestro repositorio de aplicaciones. Ahora simplemente necesitamos hacer push a Heroku para desplegar nuestra aplicación:

```
$ git push heroku master
```

> **Nota**: Esto posiblemente produzca *un montón* de salida si es la primera vez que lo ejecutas, como Heroku compila e instala psycopg. Verás que tu comando funcionó si ves algo como `https://yourapplicationname.herokuapp.com/deployed to Heroku` cerca al final de output.

Visita tu aplicación

Tu has desplegado tu código a Heroku, y especificado los tipos de proceso en un archivo `Procfile` (nosotros escogimos un proceso `web` anteriormente). Ahora puedes decirle a Heroku que inicie este proceso `web`

Para hacerlo, ejecuta el siguiente comando:

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

Esto le dice a Heroku que ejecute solamente una instancia de nuestro proceso `web`. Desde que nuestra aplicación es muy simple, nosotros no necesitamos mucho poder, y por eso está bien un solo proceso. Es posible solicitar a Heroku que ejecute mas procesos (por cierto, Heroku llama a estos procesos "Dynos" así que no te sorprendas si ves estos términos) pero ya no serán gratis.

Puedes visitar la app en tu navegador ejecutando `heroku open`.

```
$ heroku open
```

> **Nota**: ¡verás una página de error! Hablaremos de eso en un minuto.

Esto abrirá una url como https://djangogirlsblog.herokuapp.com/ en tu navegador, y por el momento, posiblemente veas una página de error.

El error que ves es porque cuando desplegamos a Heroku, creamos una nueva base de datos y está vacía. Necesitamos ejecutar los comandos `migrate` y `createsuperuser`, justo como hicimos en PythonAnywhere. Esta vez, ellos vienen en una versión especial en nuestro computador, `heroku run`:

```
$ heroku run python manage.py migrate

$ heroku run python manage.py createsuperuser
```

Este comando te preguntará que escojas un nombre de usuario y contraseña de nuevo. Estas serán tus credenciales de acceso a la página de administración de tu sitio.

¡Refresca en tu navegador y ahí estás! Ahora sabes como desplegar a dos diferentes plataformas de hosting. Escoge tu favorita :)