Tutorial Extensions

django girls

Table of Contents

Django Girls Tutorial: Extensions

Introduction

This book contains additional tutorials you can do after you're finished with Django Girls Tutorial.

Current tutorials are:

- Homework: add more to your website!
- Homework: secure your website
- Homework: create comment model
- Optional: PostgreSQL installation

Contributing

This tutorials are maintained by DjangoGirls. If you find any mistakes or want to update the tutorial please follow the contributing guidelines.

Homework: add more to your website!

Yes, this is the last thing we will do in this tutorial. You have already learned a lot! Time to use this knowledge.

Page with list of unpublished posts

Remember the chapter about querysets? We created a view `post_list` that displays only published blog posts (those with non-empty `published_date`).

Time to do something similar, but for draft posts.

Let's add a link in `blog/templates/blog/base.html` near the button for adding new posts (just above `<h1><a href="/">Django Girls Blog</a></h1>` line!):

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

Next: urls! In `blog/urls.py` we add:

```
url(r'^drafts/$', views.post_draft_list, name='post_draft_list'),
```

Time to create a view in `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})
```

This line `Post.objects.filter(published_date__isnull=True).order_by('created_date')` makes sure we take only unpublished posts (`published_date__isnull=True`) and order them by `created_date` (`order_by('created_date')`).

Ok, the last bit is of course a template! Create a file `blog/templates/blog/post_draft_list.html` and add the following:

```
{% 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 "blog.views.post_detail" pk=post.pk %}">{{ post.title }}</a></h1>
            <p>{{ post.text|truncatechars:200 }}</p>
        </div>
    {% endfor %}
{% endblock %}
```

It looks very similar to our `post_list.html`, right?

Now when you go to `http://127.0.0.1:8000/drafts/` you will see the list of unpublished posts.

Yay! Your first task is done!

Add publish button

It would be nice to have a button on the blog post detail page that will immediately publish the post, right?

Let's open `blog/template/blog/post_detail.html` and change these lines:

```
{% if post.published_date %}
    {{ post.published_date }}
{% endif %}
```

into these:

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

As you noticed, we added `{% else %}` line here. That means, that if the condition from `{% if post.published_date %}` is not fulfilled (so if there is no `published_date`), then we want to do the line `<a class="btn btn-default" href="{% url "post_publish" pk=post.pk %}">Publish</a>`. Note that we are passing a `pk` variable in the `{% url %}`.

Time to create a URL (in `blog/urls.py`):

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

and finally, a *view* (as always, in `blog/views.py`):

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

Remember, when we created a `Post` model we wrote a method `publish`. It looked like this:

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

Now we can finally use this!

And once again after publishing the post we are immediately redirected to the `post_detail` page!

Congratulations! You are almost there. The last step is adding a delete button!

Delete post

Let's open `blog/templates/blog/post_detail.html` once again and add this line:

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

just under a line with the edit button.

Now we need a URL (`blog/urls.py`):

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

Now, time for a view! Open `blog/views.py` and add this code:

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

The only new thing is to actually delete a blog post. Every Django model can be deleted by `.delete()`. It is as simple as that!

And this time, after deleting a post we want to go to the webpage with a list of posts, so we are using `redirect`.

Let's test it! Go to the page with a post and try to delete it!

Yes, this is the last thing! You completed this tutorial! You are awesome!

Homework: Adding security to your website

You might have noticed that you didn't have to use your password, apart from back when we used the admin interface. You might also have noticed that this means that anyone can add or edit posts in your blog. I don't know about you, but I don't want just anyone to post on my blog. So lets do something about it.

Authorizing add/edit of posts

First lets make things secure. We will protect our `post_new`, `post_edit`, `post_draft_list`, `post_remove` and `post_publish` views so that only logged-in users can access them. Django ships with some nice helpers for that using, the kind of advanced topic, *decorators*. Don't worry about the technicalities now, you can read up on these later. The decorator to use is shipped in Django in the module `django.contrib.auth.decorators` and is called `login_required`.

So edit your `blog/views.py` and add these lines at the top along with the rest of the imports:

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

Then add a line before each of the `post_new`, `post_edit`, `post_draft_list`, `post_remove` and `post_publish` views (decorating them) like the following:

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

Thats it! Now try to access `http://localhost:8000/post/new/`, notice the difference?

> If you just got the empty form, you are probably still logged in from the chapter on the admin-interface. Go to `http://localhost:8000/admin/logout/` to log out, then goto `http://localhost:8000/post/new` again.

You should get one of the beloved errors. This one is quite interesting actually: The decorator we added before will redirect you to the login page. But that isn't available yet, so it raises a "Page not found (404)".

Don't forget to add the decorator from above to `post_edit`, `post_remove`, `post_draft_list` and `post_publish` too.

Horray, we reached part of the goal! Other people can't just create posts on our blog anymore. Unfortunately we can't create posts anymore too. So lets fix that next.

Login users

Now we could try to do lots of magic stuff to implement users and passwords and authentication but doing this kind of stuff correctly is rather complicated. As Django is "batteries included", someone has done the hard work for us, so we will make further use of the authentication stuff provided.

In your `mysite/urls.py` add a url `url(r'^accounts/login/$', 'django.contrib.auth.views.login')`. So the file should now look similar to this:

```
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')),
)
```

Then we need a template for the login page, so create a directory `blog/templates/registration` and a file inside named `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 "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 %}
```

You will see that this also makes use of our base-template for the overall look and feel of your blog.

The nice thing here is that this *just works[TM]*. We don't have to deal with handling of the forms submission nor with passwords and securing them. Only one thing is left here, we should add a setting to `mysite/settings.py`:

```
LOGIN_REDIRECT_URL = '/'
```

Now when the login is accessed directly, it will redirect successful login to the top level index.

Improving the layout

So now we made sure that only authorized users (ie. us) can add, edit or publish posts. But still everyone gets to view the buttons to add or edit posts, lets hide these for users that aren't logged in. For this we need to edit the templates, so lets start with the base template from `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-plus"></span></a>
        <a href="{% url "post_draft_list" %}" class="top-menu"><span class="glyphicon glyphicon-edit"></span></a>
        {% else %}
        <a href="{% url "django.contrib.auth.views.login" %}" class="top-menu"><span class="glyphicon glyphicon-lock"><
        {% 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>
```

You might recognize the pattern here. There is an if-condition inside the template that checks for authenticated users to show the edit buttons. Otherwise it shows a login button.

Homework: Edit the template `blog/templates/blog/post_detail.html` to only show the edit buttons for authenticated users.

More on authenticated users

Lets add some nice sugar to our templates while we are at it. First we will add some stuff to show that we are logged in. Edit `blog/templates/blog/base.html` like this:

```
<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 "django.contrib.auth.views.logout" %}">Log out
    {% else %}
    <a href="{% url "django.contrib.auth.views.login" %}" class="top-menu"><span class="glyphicon glyphicon-lock"></spa
    {% endif %}
    <h1><a href="{% url "blog.views.post_list" %}">Django Girls</a></h1>
</div>
```

This adds a nice "Hello <username>" to remind us who we are and that we are authenticated. Also this adds a link to log out of the blog. But as you might notice this isn't working yet. Oh nooz, we broke the internetz! Lets fix it!

We decided to rely on django to handle login, lets see if Django can also handle logout for us. Check https://docs.djangoproject.com/en/1.8/topics/auth/default/ and see if you find something.

Done reading? You should by now think about adding a url (in `mysite/urls.py`) pointing to the `django.contrib.auth.views.logout` view. Like this:

```
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')),
)
```

Thats it! If you followed all of the above until this point (and did the homework), you now have a blog where you

- need a username and password to log in,
- need to be logged in to add/edit/publish(/delete) posts

- and can log out again

Homework: create comment model

Now we only have Post model, what about receiving some feedback from your readers?

Creating comment blog model

Let's open `blog/models.py` and append this piece of code to the end of file:

```
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
```

You can go back to **Django models** chapter in tutorial if you need to remind yourself what each of field types means.

In this chapter we have new type of field:

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

And `related_name` option in `models.ForeignKey` allow us to have access to comments from post model.

Create tables for models in your database

Now it's time to add our comment model to database. To do this we have to let know Django that we made changes in our model. Type `python manage.py makemigrations blog`. Just like this:

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

You can see that this command created for us another migration file in `blog/migrations/` directory. Now we need to apply those changes with `python manage.py migrate blog`. It should look like this:

```
(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
```

Our Comment model exists in database now. It would be nice if we had access to it in our admin panel.

Register comment model in admin panel

To register model in admin panel, go to `blog/admin.py` and add line:

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

Don't forget to import Comment model, file should look like this:

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

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

If you type `python manage.py runserver` in command prompt and go to http://127.0.0.1:8000/admin/ in your browser, you should have access to list, add and remove comments. Don't hesitate to play with it!

Make our comments visible

Go to `blog/templated/blog/post_detail.html` file and add those lines before `{% endblock %}` tag:

```
<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 %}
```

Now we can see the comments section on pages with post details.

But it can look a little bit better, add some css to `static/css/blog.css`:

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

We can also let know about comments on post list page, go to `blog/templates/blog/post_list.html` file and add line:

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

After that our template should look like this:

```
{% 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.comments.count }}</a>
        </div>
    {% endfor %}
{% endblock content %}
```

Let your readers write comments

Right now we can see comments on our blog, but we cannot add them, let's change that!

Go to `blog/forms.py` and add those lines to the end of the file:

```
class CommentForm(forms.ModelForm):

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

Don't forget to import Comment model, change line:

```
from .models import Post
```

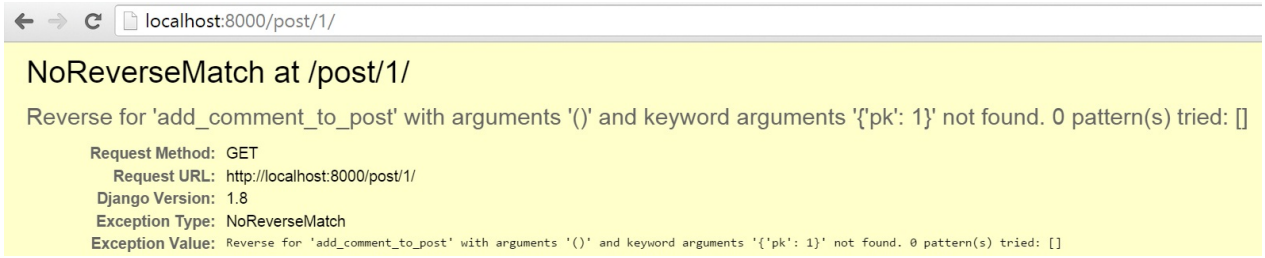
into:

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

Now go to `blog/templates/blog/post_detail.html` and before line `{% for comment in post.comments.all %}` add:

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

Go to post detail page and you should see error:



Let's fix this! Go to `blog/urls.py` and add this pattern to `urlpatterns`:

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

Now you should see this error:

To fix this, add this piece of code to `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})
```

Don't forget about imports at the beginning of the file:

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

Now you should see:

Like error mentions, template does not exist, create one as `blog/templates/blog/add_comment_to_post.html` and add those lines:

```
{% 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 %}
```

Yay! Now your readers can let you know what they think below your posts!

Moderating your comments

Not all of our comments should be displayed. Blog owner should have option to approve or delete comments. Let's do something about it.

Go to `blog/templates/blog/post_detail.html` and change lines:

```
{% 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 %}
```

to:

```
{% 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 glyph
                <a class="btn btn-default" href="{% url "comment_approve" pk=comment.pk %}"><span class="glyphicon glyp
            {% endif %}
        </div>
        <strong>{{ comment.author }}</strong>
        <p>{{ comment.text|linebreaks }}</p>
    </div>
    {% endif %}
{% empty %}
    <p>No comments here yet :(</p>
{% endfor %}
```

You should see `NoReverseMatch`, because no url matches `comment_remove` and `comment_approve` patterns.

Add url patterns to `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'),
```

Now you should see `AttributeError`. To get rid of it, create more views in `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)
```

And of course fix imports.

Everything works, but there is one misconception. In our post list page under posts we see number of all comments attached, but we want to have number of approved comments there.

Go to `blog/templates/blog/post_list.html` and change line:

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

to:

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

And also add this method to Post model in `blog/models.py`:

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

Now your comment feature is finished! Congrats! :-)

PostgreSQL installation

> Part of this chapter is based on tutorials by Geek Girls Carrots (http://django.carrots.pl/).
>
> Parts of this chapter is based on the django-marcador tutorial licensed under Creative Commons Attribution-ShareAlike 4.0 International License. The django-marcador tutorial is copyrighted by Markus Zapke-Gründemann et al.

Windows

The easiest way to install Postgres on Windows is using a program you can find here: http://www.enterprisedb.com/products-services-training/pgdownload#windows

Choose the newest version available for your operating system. Download the installer, run it and then follow the instructions available here: http://www.postgresqltutorial.com/install-postgresql/. Take note of the installation directory as you will need it in the next step (typically, it's `C:\Program Files\PostgreSQL\9.3`).

Mac OS X

The easiest way is to download the free Postgres.app and install it like any other application on your operating system.

Download it, drag to the Applications directory and run by double clicking. That's it!

You'll also have to add the Postgres command line tools to your `PATH` variable, what is described here.

Linux

Installation steps vary from distribution to distribution. Below are the commands for Ubuntu and Fedora, but if you're using a different distro take a look at the PostgreSQL documentation.

Ubuntu

Run the following command:

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

Fedora

Run the following command:

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

Create database

Next up, we need to create our first database, and a user that can access that database. PostgreSQL lets you create as many databases and users as you like, so if you're running more than one site you should create a database for each one.

Windows

If you're using Windows, there's a couple more steps we need to complete. For now it's not important for you to understand the configuration we're doing here, but feel free to ask your coach if you're curious as to what's going on.

1. Open the Command Prompt (Start menu → All Programs → Accessories → Command Prompt)
2. Run the following by typing it in and hitting return: `setx PATH "%PATH%;C:\Program Files\PostgreSQL\9.3\bin"`. You can paste things into the Command Prompt by right clicking and selecting `Paste`. Make sure that the path is the same one you noted during installation with `\bin` added at the end. You should see the message `SUCCESS: Specified value was saved.`.
3. Close and then reopen the Command Prompt.

Create the database

First, let's launch the Postgres console by running `psql`. Remember how to launch the console?

> On Mac OS X you can do this by launching the `Terminal` application (it's in Applications → Utilities). On Linux, it's probably under Applications → Accessories → Terminal. On Windows you need to go to Start menu → All Programs → Accessories → Command Prompt. Furthermore, on Windows, `psql` might require logging in using the username and password you chose during installation. If `psql` is asking you for a password and doesn't seem to work, try `psql -U <username> -W` first and enter the password later.

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

Our `$` now changed into `#`, which means that we're now sending commands to PostgreSQL. Let's create a user:

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

Replace `name` with your own name. You shouldn't use accented letters or whitespace (e.g. `bożena maria` is invalid - you need to convert it into `bozena_maria`).

Now it's time to create a database for your Django project:

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

Remember to replace `name` with the name you've chosen (e.g. `bozena_maria`).

Great - that's databases all sorted!

Updating settings

Find this part in your `mysite/settings.py` file:

```
DATABASES = {
    'default': {
```

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

And replace it with this:

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

Remeber to change `name` to the user name that you created earlier in this chapter.

Installing PostgreSQL package for Python

First, install Heroku Toolbelt from https://toolbelt.heroku.com/ While we will need this mostly for deploying your site later on, it also includes Git, which might come in handy already.

Next up, we need to install a package which lets Python talk to PostgreSQL - this is called `psycopg2`. The installation instructions differ slightly between Windows and Linux/OS X.

Windows

For Windows, download the pre-built file from http://www.stickpeople.com/projects/python/win-psycopg/

Make sure you get the one corresponding to your Python version (3.4 should be the last line) and to the correct architecture (32 bit in the left column or 64 bit in the right column).

Rename the downloaded file and move it so that it's now available at `C:\psycopg2.exe`.

Once that's done, enter the following command in the terminal (make sure your `virtualenv` is activated):

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

Linux and OS X

Run the following in your console:

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

If that goes well, you'll see something like this

```
Downloading/unpacking psycopg2
Installing collected packages: psycopg2
```

```
  Successfully installed psycopg2
  Cleaning up...
```

Once that's completed, run `python -c "import psycopg2"`. If you get no errors, everything's installed successfully.

Domain

PythonAnywhere gave you a free domain, but maybe you don't want to have ".pythonanywhere.com" at the end of your blog URL. Maybe you want your blog to just live at "www.infinite-kitten-pictures.org" or "www.3d-printed-steam-engine-parts.com" or "www.antique-buttons.com" or "www.mutant-unicornz.net", or whatever it'll be.

Here we'll talk a bit about where to get a domain, and how to hook it up to your web app on PythonAnywhere. However, you should know that most domains cost money, and PythonAnywere also charges a monthly fee to use your own domain name -- it's not much money in total, but this is probably something you only want to do if you're really committed!

Where to register a domain?

A typical domain costs around $15 a year. There are cheaper and more expensive options, depending on the provider. There are a lot of companies that you can buy a domain from: a simple google search will give hundreds of options.

Our favourite one is I want my name. They advertise as "painless domain management" and it really is painless.

You can also get domains for free. dot.tk is one place to get one, but you should be aware that free domains sometimes feel a bit cheap -- if your site is going to be for a professional business, you might want to think about paying for a "proper" domain that ends in `.com`.

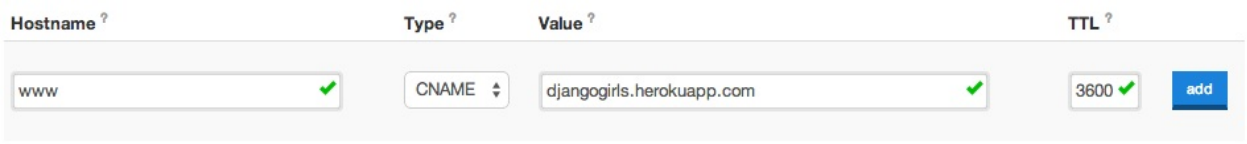
How to point your domain at PythonAnywhere

If you went through *iwantmyname.com*, click `Domains` in the menu and choose your newly purchased domain. Then locate and click on the `manage DNS records` link:

Now you need to locate this form:

And fill it in with the following details:

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



Click the Add button and Save changes at the bottom.

> **Note** If you used a different domain provider, the exact UI for finding your DNS / CNAME settings will be different, but your objective is the same: to set up a CNAME that points your new domain at `yourusername.pythonanywhere.com`.

It can take a few minutes for your domain to start working, so be patient!

Configure the domain via a web app on PythonAnywhere.

You also need to tell PythonAnywhere that you want to use your custom domain.

Go to the PythonAnywhere Accounts page and upgrade your account. The cheapest option (a "Hacker" plan) is fine to start with, you can always upgrade it later when you get super-famous and have millions of hits.

Next, go over to the Web tab and note down a couple of things:

- Copy the **path to your virtualenv** and put it somewhere safe
- Click through to your **wsgi config file**, copy the contents, and paste them somewhere safe.

Next, **Delete** your old web app. Don't worry, this doesn't delete any of your code, it just switches off the domain at *yourusername.pythonanywhere.com*. Next, create a new web app, and follow these steps:

- Enter your new domain name
- Choose "manual configuration"
- Pick Python 3.4
- And we're done!

When you get taken back to the web tab.

- Paste in the virtualenv path you saved earlier
- Click through to the wsgi configuration file, and paste in the contents from your old config file

Hit reload web app, and you should find your site is live on its new domain!

If you run into any problems, hit the "Send feedback" link on the PythonAnywhere site, and one of their friendly admins will be there to help you in no time.

Deploy to Heroku (as well as PythonAnyhere)

It's always good for a developer to have a couple of different deployment options under their belt. Why not try deploying your site to Heroku, as well as PythonAnywhere?

Heroku is also free for small applications that don't have too many visitors, but it's a bit more tricky to get deployed.

We will be following this tutorial: https://devcenter.heroku.com/articles/getting-started-with-django, but we pasted it here so it's easier for you.

The `requirements.txt` file

If you didn't create one before, we need to create a `requirements.txt` file to tell Heroku what Python packages need to be installed on our server.

But first, Heroku needs us to install a few new packages. Go to your console with `virtualenv` activated and type this:

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

After the installation is finished, go to the `djangogirls` directory and run this command:

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

This will create a file called `requirements.txt` with a list of your installed packages (i.e. Python libraries that you are using, for example Django :)).

> **Note**: `pip freeze` outputs a list of all the Python libraries installed in your virtualenv, and the `>` takes the output of `pip freeze` and puts it into a file. Try running `pip freeze` without the `> requirements.txt` to see what happens!

Open this file and add the following line at the bottom:

```
psycopg2==2.5.4
```

This line is needed for your application to work on Heroku.

Procfile

Another thing Heroku wants is a Procfile. This tells Heroku which commands to run in order to start our website. Open up your code editor, create a file called `Procfile` in `djangogirls` directory and add this line:

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

This line means that we're going to be deploying a `web` application, and we'll do that by running the command `gunicorn mysite.wsgi` (`gunicorn` is a program that's like a more powerful version of Django's `runserver` command).

Then save it. Done!

The `runtime.txt` file

We also need to tell Heroku which Python version we want to use. This is done by creating a `runtime.txt` in the `djangogirls` directory using your editor's "new file" command, and putting the following text (and nothing else!) inside:

```
python-3.4.2
```

`mysite/local_settings.py`

Because it's more restrictive than PythonAnywhere, Heroku wants to use different settings from the ones we use on our locally (on our computer). Heroku wants to use Postgres while we use SQLite for example. That's why we need to create a separate file for settings that will only be available for our local environment.

Go ahead and create `mysite/local_settings.py` file. It should contain your `DATABASE` setup from your `mysite/settings.py` file. Just like that:

```
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
```

Then just save it! :)

mysite/settings.py

Another thing we need to do is modify our website's `settings.py` file. Open `mysite/settings.py` in your editor and add the following lines at the end of the file:

```
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
```

It'll do necessary configuration for Heroku and also it'll import all of your local settings if `mysite/local_settings.py` exists.

Then save the file.

mysite/wsgi.py

Open the `mysite/wsgi.py` file and add these lines at the end:

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

All right!

Heroku account

You need to install your Heroku toolbelt which you can find here (you can skip the installation if you've already installed it during setup): https://toolbelt.heroku.com/

> When running the Heroku toolbelt installation program on Windows make sure to choose "Custom Installation" when being asked which components to install. In the list of components that shows up after that please additionally check the checkbox in front of "Git and SSH".
>
> On Windows you also must run the following command to add Git and SSH to your command prompt's `PATH`: `setx PATH "%PATH%;C:\Program Files\Git\bin"`. Restart the command prompt program afterwards to enable the change.
>
> After restarting your command prompt, don't forget to go to your `djangogirls` folder again and activate your virtualenv! (Hint: Check the Django installation chapter)

Please also create a free Heroku account here: https://id.heroku.com/signup/www-home-top

Then authenticate your Heroku account on your computer by running this command:

```
$ heroku login
```

In case you don't have an SSH key this command will automatically create one. SSH keys are required to push code to the Heroku.

Git commit

Heroku uses git for its deployments. Unlike PythonAnywhere, you can push to Heroku directly, without going via Github. But we need to tweak a couple of things first.

Open the file named `.gitignore` in your `djangogirls` directory and add `local_settings.py` to it. We want git to ignore `local_settings`, so it stays on our local computer and doesn't end up on Heroku.

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

And we commit our changes

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

Pick an application name

We'll be making your blog available on the Web at `[your blog's name].herokuapp.com`, so we need to choose a name that nobody else has taken. This name doesn't need to be related to the Django `blog` app or to `mysite` or anything we've created so far. The name can be anything you want, but Heroku is quite strict as to what characters you can use: you're only allowed to use simple lowercase letters (no capital letters or accents), numbers, and dashes (`-`).

Once you've thought of a name (maybe something with your name or nickname in it), run this command, replacing `djangogirlsblog` with your own application name:

```
$ heroku create djangogirlsblog
```

> **Note**: Remember to replace `djangogirlsblog` with the name of your application on Heroku.

If you can't think of a name, you can instead run

```
$ heroku create
```

and Heroku will pick an unused name for you (probably something like `enigmatic-cove-2527`).

If you ever feel like changing the name of your Heroku application, you can do so at any time with this command (replace `the-new-name` with the new name you want to use):

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

> **Note**: Remember that after you change your application's name, you'll need to visit `[the-new-name].herokuapp.com` to see your site.

Deploy to Heroku!

That was a lot of configuration and installing, right? But you only need to do that once! Now you can deploy!

When you ran `heroku create`, it automatically added the Heroku remote for our app to our repository. Now we can do a simple git push to deploy our application:

```
$ git push heroku master
```

> **Note**: This will probably produce a *lot* of output the first time you run it, as Heroku compiles and installs psycopg. You'll know it's succeeded if you see something like `https://yourapplicationname.herokuapp.com/ deployed to Heroku` near the end of the output.

Visit your application

You've deployed your code to Heroku, and specified the process types in a `Procfile` (we chose a `web` process type earlier). We can now tell Heroku to start this `web process`.

To do that, run the following command:

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

This tells Heroku to run just one instance of our `web` process. Since our blog application is quite simple, we don't need too much power and so it's fine to run just one process. It's possible to ask Heroku to run more processes (by the way, Heroku calls these processes "Dynos" so don't be surprised if you see this term) but it will no longer be free.

We can now visit the app in our browser with `heroku open`.

```
$ heroku open
```

> **Note**: you will see an error page! We'll talk about that in a minute.

This will open a url like https://djangogirlsblog.herokuapp.com/ in your browser, and at the moment you will probably see an error page.

The error you saw was because we when we deployed to Heroku, we created a new database and it's empty. We need to run the `migrate` and `createsuperuser` commands, just like we did on PythonAnywhere. This time, they come via a special command-line on our own computer, `heroku run`:

```
$ heroku run python manage.py migrate

$ heroku run python manage.py createsuperuser
```

The command prompt will ask you to choose a username and a password again. These will be your login details on your live website's admin page.

Refresh it in your browser, and there you go! You now know how to deploy to two different hosting platforms. Pick your favourite :)