tutorial.djangogirls.org

Django urls · Django Girls Tutorial

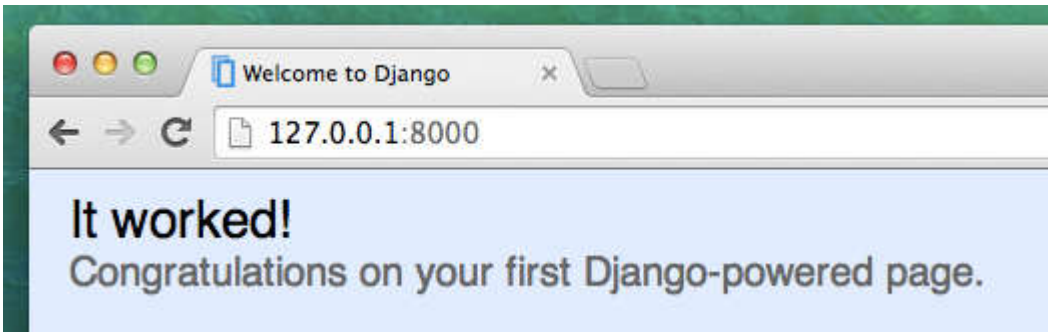
DjangoGirls

6-8 minuti

We're about to build our first webpage: a homepage for your blog! But first, let's learn a little bit about Django URLs.

What is a URL?

A URL is simply a web address. You can see a URL every time you visit a website – it is visible in your browser's address bar. (Yes! `127.0.0.1:8000` is a URL! And `https://djangogirls.org` is also a URL.)



Every page on the Internet needs its own URL. This way your application knows what it should show to a user who opens that URL. In Django we use something called `URLconf` (URL configuration). URLconf is a set of patterns that Django will try to match with the requested URL to find the correct view.

How do URLs work in Django?

Let's open up the `mysite/urls.py` file in your code editor of

choice and see what it looks like:

mysite/urls.py

```
"""mysite URL Configuration

[...]
"""
from django.conf.urls import url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
]
```

As you can see, Django has already put something here for us.

Lines between triple quotes (`'''` or `"""`) are called docstrings – you can write them at the top of a file, class or method to describe what it does. They won't be run by Python.

The admin URL, which you visited in previous chapter, is already here:

mysite/urls.py

```
    url(r'^admin/', admin.site.urls),
```

This line means that for every URL that starts with `admin/`, Django will find a corresponding *view*. In this case we're including a lot of admin URLs so it isn't all packed into this small file – it's more readable and cleaner.

Regex

Do you wonder how Django matches URLs to views? Well, this part is tricky. Django uses `regex`, short for "regular expressions".

Regex has a lot (a lot!) of rules that form a search pattern. Since regexes are an advanced topic, we will not go in detail over how they work.

If you still wish to understand how we created the patterns, here is an example of the process – we will only need a limited subset of the rules to express the pattern we are looking for, namely:

- `^` for the beginning of the text
- `$` for the end of the text
- `\d` for a digit
- `+` to indicate that the previous item should be repeated at least once
- `()` to capture part of the pattern

Anything else in the URL definition will be taken literally.

Now imagine you have a website with the address like `http://www.mysite.com/post/12345/`, where `12345` is the number of your post.

Writing separate views for all the post numbers would be really annoying. With regular expressions, we can create a pattern that will match the URL and extract the number for us: `^post/(\d+)/$`. Let's break this down piece by piece to see what we are doing here:

- **^post/** is telling Django to take anything that has `post/` at the beginning of the url (right after `^`)
- **(\d+)** means that there will be a number (one or more digits) and that we want the number captured and extracted
- **/** tells django that another `/` character should follow
- **$** then indicates the end of the URL meaning that only strings

ending with the `/` will match this pattern

Your first Django URL!

Time to create our first URL! We want 'http://127.0.0.1:8000/' to be the home page of our blog and to display a list of posts.

We also want to keep the `mysite/urls.py` file clean, so we will import URLs from our `blog` application to the main `mysite/urls.py` file.

Go ahead, add a line that will import `blog.urls`. Note that we are using the `include` function here so **you will need** to add that to the import on the first line of the file.

Your `mysite/urls.py` file should now look like this:

mysite/urls.py

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'', include('blog.urls')),
]
```

Django will now redirect everything that comes into 'http://127.0.0.1:8000/' to `blog.urls` and look for further instructions there.

Writing regular expressions in Python is always done with `r` in front of the string. This is a helpful hint for Python that the string may contain special characters that are not meant for Python itself, but for the regular expression instead.

blog.urls

Create a new empty file named `urls.py` in the `blog` directory. All right! Add these first two lines:

blog/urls.py

```
from django.conf.urls import url
from . import views
```

Here we're importing Django's function `url` and all of our `views` from the `blog` application. (We don't have any yet, but we will get to that in a minute!)

After that, we can add our first URL pattern:

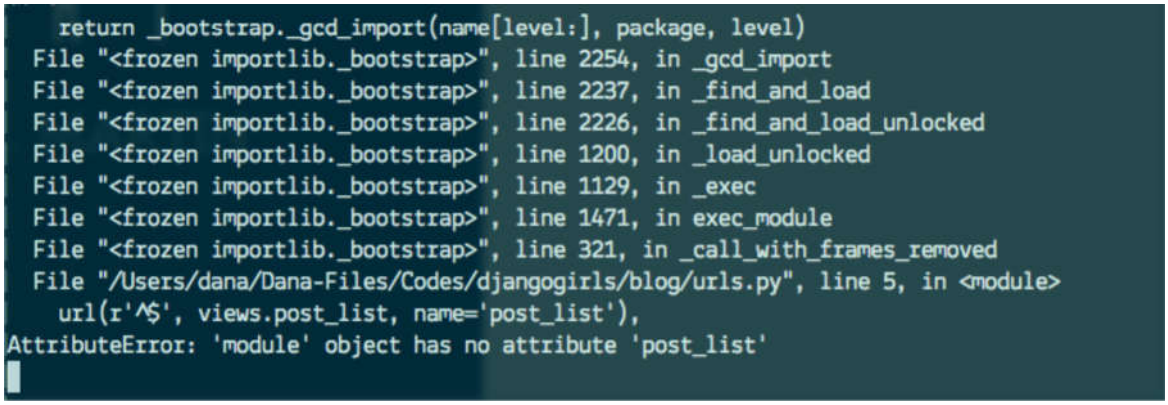
blog/urls.py

```
urlpatterns = [
    url(r'^$', views.post_list,
name='post_list'),
]
```

As you can see, we're now assigning a `view` called `post_list` to the `^$` URL. This regular expression will match `^` (a beginning) followed by `$` (an end) – so only an empty string will match. That's correct, because in Django URL resolvers, 'http://127.0.0.1:8000/' is not a part of the URL. This pattern will tell Django that `views.post_list` is the right place to go if someone enters your website at the 'http://127.0.0.1:8000/' address.

The last part, `name='post_list'`, is the name of the URL that will be used to identify the view. This can be the same as the name of the view but it can also be something completely different. We will be using the named URLs later in the project, so it is important to name each URL in the app. We should also try to keep the names of URLs unique and easy to remember.

If you try to visit http://127.0.0.1:8000/ now, then you'll find some sort of 'web page not available' message. This is because the server (remember typing `runserver`?) is no longer running. Take a look at your server console window to find out why.



Your console is showing an error, but don't worry – it's actually pretty useful: It's telling you that there is **no attribute 'post_list'**. That's the name of the *view* that Django is trying to find and use, but we haven't created it yet. At this stage your `/admin/` will also not work. No worries – we will get there.

> If you want to know more about Django URLconfs, look at the official documentation: https://docs.djangoproject.com/en/1.11/topics/http/urls/

tutorial.djangogirls.org

Django views – time to create! · Django Girls Tutorial

DjangoGirls

2-3 minuti

Time to get rid of the bug we created in the last chapter! :)

A *view* is a place where we put the "logic" of our application. It will request information from the `model` you created before and pass it to a `template`. We'll create a template in the next chapter. Views are just Python functions that are a little bit more complicated than the ones we wrote in the **Introduction to Python** chapter.

Views are placed in the `views.py` file. We will add our *views* to the `blog/views.py` file.

blog/views.py

OK, let's open up this file and see what's in there:

blog/views.py

```
from django.shortcuts import render
```

Not too much stuff here yet.

Remember that lines starting with `#` are comments – this means that those lines won't be run by Python.

Let's create a *view* as the comment suggests. Add the following minimal view below it:

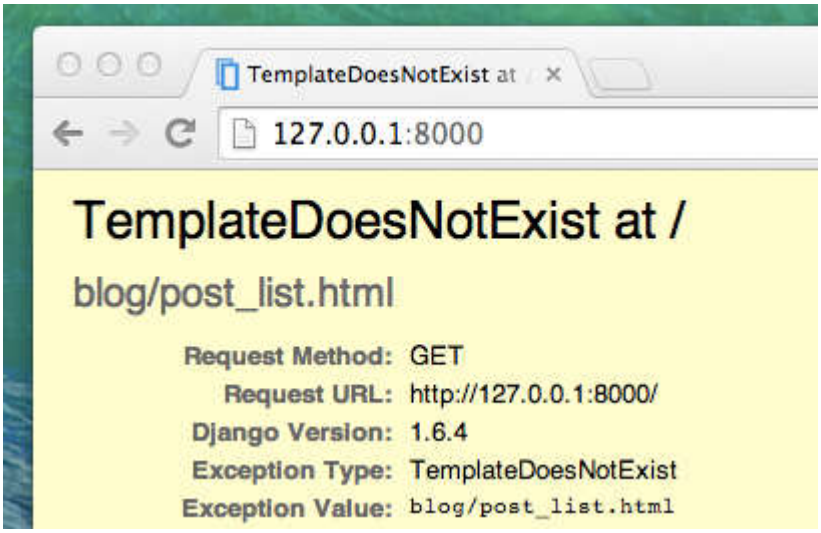
blog/views.py

```
def post_list(request):
    return render(request,
'blog/post_list.html')
```

As you can see, we created a function (`def`) called `post_list` that takes `request` and `return` a function `render` that will render (put together) our template `blog/post_list.html`.

Save the file, go to http://127.0.0.1:8000/ and see what we've got.

Another error! Read what's going on now:



This shows that the server is running again, at least, but it still doesn't look right, does it? Don't worry, it's just an error page, nothing to be scared of! Just like the error messages in the console, these are actually pretty useful. You can read that the *TemplateDoesNotExist*. Let's fix this bug and create a template in the next chapter!

> Learn more about Django views by reading the official documentation: https://docs.djangoproject.com/en/1.11/topics/http/views/

tutorial.djangogirls.org

Dynamic data in templates · Django Girls Tutorial

DjangoGirls

4-5 minuti

We have different pieces in place: the `Post` model is defined in `models.py`, we have `post_list` in `views.py` and the template added. But how will we actually make our posts appear in our HTML template? Because that is what we want to do – take some content (models saved in the database) and display it nicely in our template, right?

This is exactly what *views* are supposed to do: connect models and templates. In our `post_list` *view* we will need to take the models we want to display and pass them to the template. In a *view* we decide what (model) will be displayed in a template.

OK, so how will we achieve this?

We need to open our `blog/views.py`. So far `post_list` *view* looks like this:

blog/views.py

```
from django.shortcuts import render

def post_list(request):
    return render(request, 'blog/post_list.html',
{})
```

Remember when we talked about including code written in different files? Now is the moment when we have to include the model we have written in `models.py`. We will add the line `from .models import Post` like this:

blog/views.py

```
from django.shortcuts import render
from .models import Post
```

The dot before `models` means *current directory* or *current application*. Both `views.py` and `models.py` are in the same directory. This means we can use `.` and the name of the file (without `.py`). Then we import the name of the model (`Post`).

But what's next? To take actual blog posts from the `Post` model we need something called `QuerySet`.

QuerySet

You should already be familiar with how QuerySets work. We talked about them in Django ORM (QuerySets) chapter.

So now we want published blog posts sorted by `published_date`, right? We already did that in QuerySets chapter!

blog/views.py

```
Post.objects.filter(published_date__lte=timezone.now()).order_by('pub
```

Now we put this piece of code inside the `blog/views.py` file by adding it to the function `def post_list(request)`, but don't forget to first add `from django.utils import timezone`:

blog/views.py

```
from django.shortcuts import render
from django.utils import timezone
from .models import Post

def post_list(request):
    posts =
Post.objects.filter(published_date__lte=timezone.now()).order_by('pub
    return render(request, 'blog/post_list.html',
{})
```

The last missing part is passing the `posts` QuerySet to the template context. Don't worry – we will cover how to display it in a later chapter.

Please note that we create a *variable* for our QuerySet: `posts`. Treat this as the name of our QuerySet. From now on we can refer to it by this name.

In the `render` function we have one parameter `request` (everything we receive from the user via the Internet) and another giving the template file (`'blog/post_list.html'`). The last parameter, `{}`, is a place in which we can add some things for the template to use. We need to give them names (we will stick to `'posts'` right now). :) It should look like this: `{'posts': posts}`. Please note that the part before `:` is a string; you need to wrap it with quotes: `''`.

So finally our `blog/views.py` file should look like this:

blog/views.py

```
from django.shortcuts import render
from django.utils import timezone
from .models import Post

def post_list(request):
    posts =
Post.objects.filter(published_date__lte=timezone.now()).order_by('pub
```

```
    return render(request, 'blog/post_list.html',
{'posts': posts})
```

That's it! Time to go back to our template and display this QuerySet!

Want to read a little bit more about QuerySets in Django? You should look here: https://docs.djangoproject.com/en/1.11/ref/models/querysets/

tutorial.djangogirls.org

Extend your application · Django Girls Tutorial

DjangoGirls

7-9 minuti

We've already completed all the different steps necessary for the creation of our website: we know how to write a model, url, view and template. We also know how to make our website pretty.

Time to practice!

The first thing we need in our blog is, obviously, a page to display one post, right?

We already have a `Post` model, so we don't need to add anything to `models.py`.

Create a template link to a post's detail

We will start with adding a link inside `blog/templates/blog/post_list.html` file. So far it should look like this:

blog/templates/blog/post_list.html

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

{% block content %}
    {% for post in posts %}
        <div class="post">
            <div class="date">
```

```
                {{ post.published_date }}
            </div>
            <h1><a href="">{{ post.title }}</a>
</h1>
            <p>{{ post.text|linebreaksbr }}</p>
        </div>
    {% endfor %}
{% endblock %}
```

We want to have a link from a post's title in the post list to the post's detail page. Let's change `<h1><a href="">{{ post.title }}</a></h1>` so that it links to the post's detail page:

blog/templates/blog/post_list.html

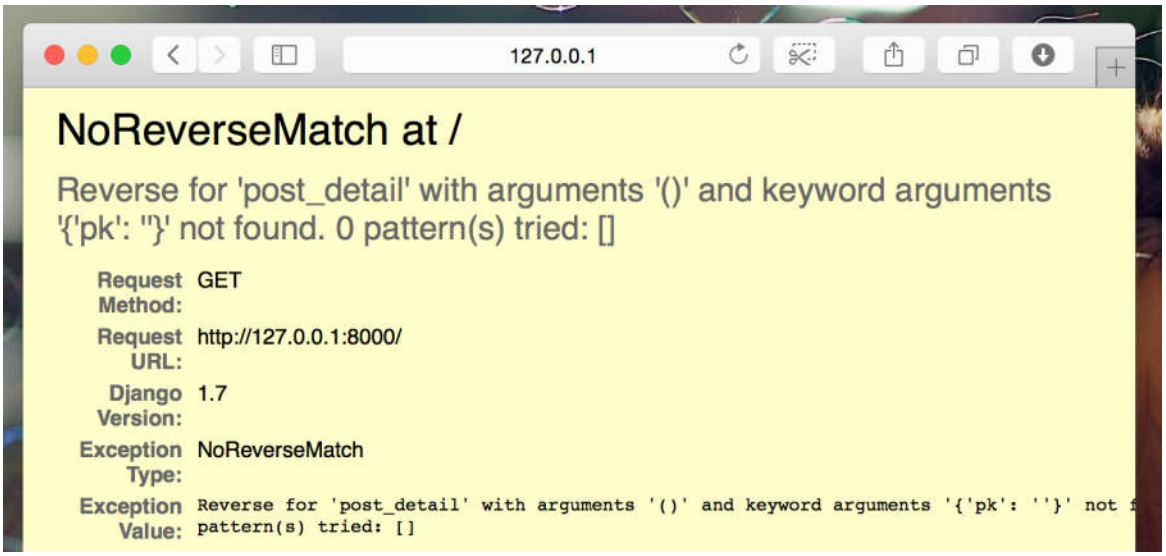
```
<h1><a href="{% url 'post_detail' pk=post.pk
%}">{{ post.title }}</a></h1>
```

Time to explain the mysterious `{% url 'post_detail' pk=post.pk %}`. As you might suspect, the `{% %}` notation means that we are using Django template tags. This time we will use one that will create a URL for us!

The `post_detail` part means that Django will be expecting a URL in `blog/urls.py` with name=post_detail

And how about `pk=post.pk`? `pk` is short for primary key, which is a unique name for each record in a database. Because we didn't specify a primary key in our `Post` model, Django creates one for us (by default, a number that increases by one for each record, i.e. 1, 2, 3) and adds it as a field named `pk` to each of our posts. We access the primary key by writing `post.pk`, the same way we access other fields (`title`, `author`, etc.) in our `Post` object!

Now when we go to http://127.0.0.1:8000/ we will have an error (as expected, since we do not yet have a URL or a *view* for `post_detail`). It will look like this:



Create a URL to a post's detail

Let's create a URL in `urls.py` for our `post_detail` *view*!

We want our first post's detail to be displayed at this **URL**: http://127.0.0.1:8000/post/1/

Let's make a URL in the `blog/urls.py` file to point Django to a *view* named `post_detail`, that will show an entire blog post. Add the line `url(r'^post/(?P<pk>\d+)/$', views.post_detail, name='post_detail'),` to the `blog/urls.py` file. The file should look like this:

blog/urls.py

```
from django.conf.urls import url
from . import views

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

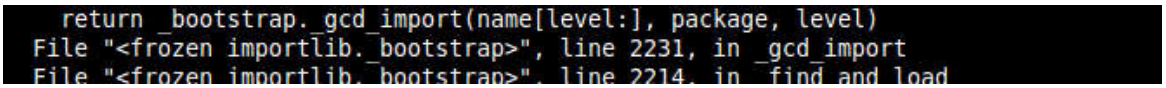
```
    views.post_detail, name='post_detail'),
]
```

This part `^post/(?P<pk>\d+)/$` looks scary, but no worries – we will explain it for you:

- it starts with `^` again – "the beginning".
- `post/` just means that after the beginning, the URL should contain the word **post** and a */*. So far so good.
- `(?P<pk>\d+)` – this part is trickier. It means that Django will take everything that you place here and transfer it to a view as a variable called `pk`. (Note that this matches the name we gave the primary key variable back in `blog/templates/blog/post_list.html`!) `\d` also tells us that it can only be a digit, not a letter (so everything between 0 and 9). `+` means that there needs to be one or more digits there. So something like `http://127.0.0.1:8000/post//` is not valid, but `http://127.0.0.1:8000/post/1234567890/` is perfectly OK!
- `/` – then we need a */* again.
- `$` – "the end"!

That means if you enter `http://127.0.0.1:8000/post/5/` into your browser, Django will understand that you are looking for a *view* called `post_detail` and transfer the information that `pk` equals `5` to that *view*.

OK, we've added a new URL pattern to `blog/urls.py`! Let's refresh the page: http://127.0.0.1:8000/ Boom! The server has stopped running again. Have a look at the console – as expected, there's yet another error!



```
  File "<frozen importlib._bootstrap>", line 2203, in _find_and_load_unlocked
  File "<frozen importlib._bootstrap>", line 1200, in _load_unlocked
  File "<frozen importlib._bootstrap>", line 1129, in _exec
  File "<frozen importlib._bootstrap>", line 1448, in exec_module
  File "<frozen importlib._bootstrap>", line 321, in _call_with_frames_removed
  File "/home/hel/code/djangogirls/workthrough/blog/urls.py", line 6, in <module>
    url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail, name='post_detail'),
AttributeError: 'module' object has no attribute 'post_detail'
```

Do you remember what the next step is? Of course: adding a view!

Add a post's detail view

This time our *view* is given an extra parameter, `pk`. Our *view* needs to catch it, right? So we will define our function as `def post_detail(request, pk):`. Note that we need to use exactly the same name as the one we specified in urls (`pk`). Omitting this variable is incorrect and will result in an error!

Now, we want to get one and only one blog post. To do this, we can use querysets, like this:

blog/views.py

```
Post.objects.get(pk=pk)
```

But this code has a problem. If there is no `Post` with the given `primary key` (`pk`) we will have a super ugly error!

We don't want that! But, of course, Django comes with something

that will handle that for us: `get_object_or_404`. In case there is no `Post` with the given `pk`, it will display much nicer page, the `Page Not Found 404` page.

The good news is that you can actually create your own `Page not found` page and make it as pretty as you want. But it's not super important right now, so we will skip it.

OK, time to add a *view* to our `views.py` file!

In `blog/urls.py` we created a URL rule named `post_detail` that refers to a view called `views.post_detail`. This means that Django will be expecting a view function called `post_detail` inside `blog/views.py`.

We should open `blog/views.py` and add the following code near the other `from` lines:

blog/views.py

```
from django.shortcuts import render,
get_object_or_404
```

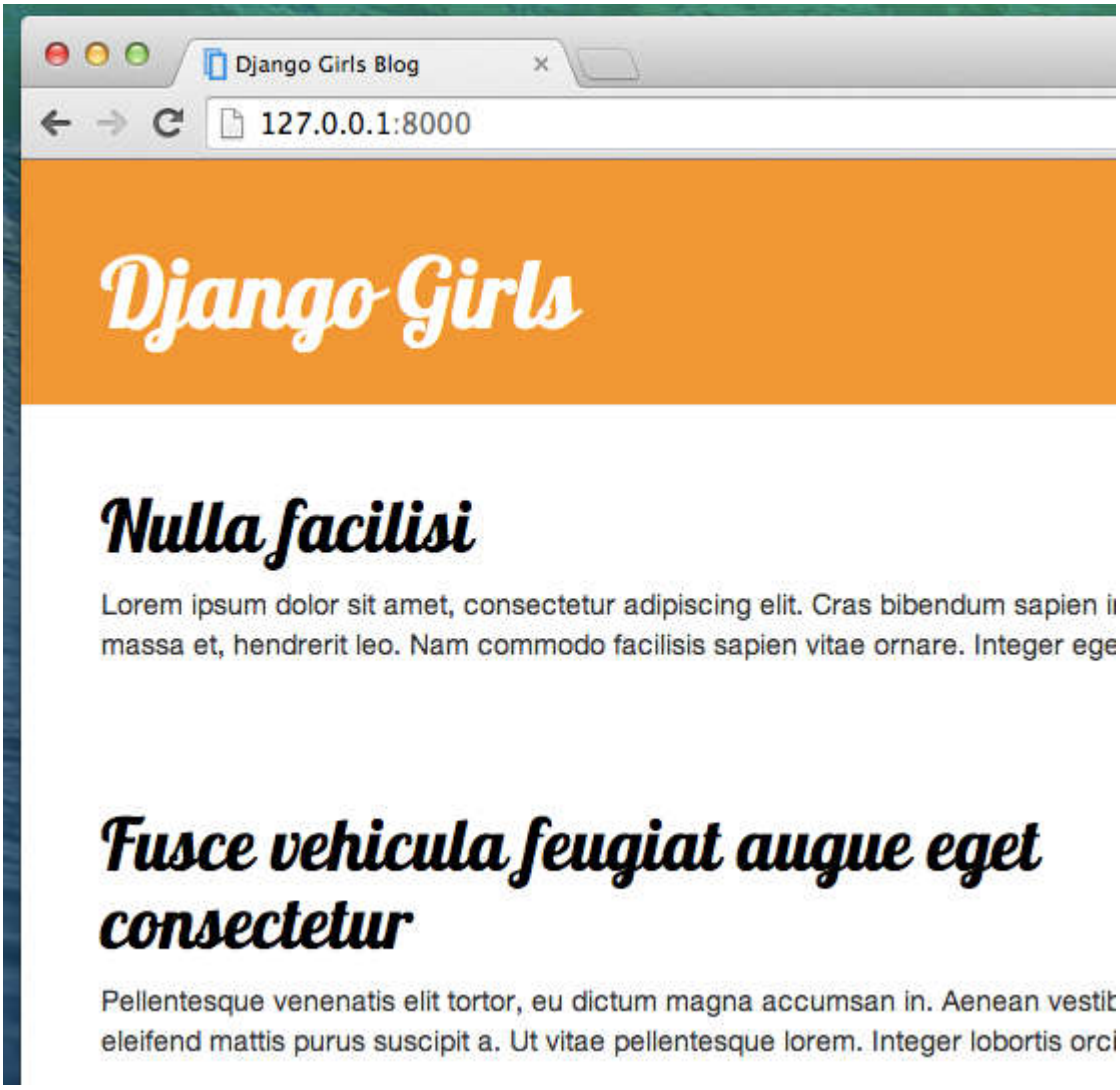
And at the end of the file we will add our *view*:

blog/views.py

```
def post_detail(request, pk):
    post = get_object_or_404(Post, pk=pk)
    return render(request,
```

```
'blog/post_detail.html', {'post': post})
```

Yes. It is time to refresh the page: http://127.0.0.1:8000/



It worked! But what happens when you click a link in blog post title?

Oh no! Another error! But we already know how to deal with it,

right? We need to add a template!

Create a template for the post details

We will create a file in `blog/templates/blog` called `post_detail.html`.

It will look like this:

blog/templates/blog/post_detail.html

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

{% block content %}
    <div class="post">
        {% if post.published_date %}
            <div class="date">
                {{ post.published_date }}
            </div>
        {% endif %}
        <h1>{{ post.title }}</h1>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endblock %}
```

Once again we are extending `base.html`. In the `content` block we want to display a post's published_date (if it exists), title and text. But we should discuss some important things, right?

`{% if ... %} ... {% endif %}` is a template tag we can use when we want to check something. (Remember `if ... else ..` from **Introduction to Python** chapter?) In this scenario we want to check if a post's `published_date` is not empty.

OK, we can refresh our page and see if

`TemplateDoesNotExist` is gone now.

Yay! It works!

One more thing: deploy time!

It'd be good to see if your website will still be working on PythonAnywhere, right? Let's try deploying again.

command-line

```
$ git status
$ git add --all .
$ git status
$ git commit -m "Added view and template for detailed blog post as well as CSS for the site."
$ git push
```

Then, in a PythonAnywhere Bash console:

command-line

```
$ cd my-first-blog
$ git pull
```

```
[...]
```

Finally, hop on over to the Web tab and hit **Reload**.

And that should be it! Congrats :)