**Django project Setup**

The names of some files and directories are very important for Django. You should not rename the files that we are about to create. Moving them to a different place is also not a good idea.

**Django needs to maintain a certain structure to be able to find important things.**

Remember to run everything in the virtualenv. If you don't see a prefix (myvenv) in your console, you need to activate your virtualenv. We explained how to do that in the **Django installation** chapter in the **Working with virtualenv** part.

**Typing myvenv\Scripts\activate on Windows or source myvenv/bin/activate on Mac OS X or Linux will do this for you.**

**Windows**

django-admin.py is a script that will create the directories and files for you. You should now have a directory structure which looks like this:

djangogirls

├───manage.py

└───mysite

settings.py : db conf, version

urls.py : navigation of web page / website

wsgi.py

\_\_init\_\_.py

Note: in your directory structure, you will also see your venv directory that we created before.

manage.py is a script that helps with management of the site. With it we will be able (amongst other things) to start a web server on our computer without installing anything else.

**The settings.py file contains the configuration of your website.**

Remember when we talked about a mail carrier checking where to deliver a letter? urls.py file contains a list of patterns used by urlresolver.

**Changing settings**

Let's make some changes in mysite/settings.py. Open the file using the code editor you installed earlier.

**Note**: Keep in mind that settings.py is a regular file, like any other. You can open it from inside the code editor, using the "file -> open" menu actions. This should get you the usual window in which you can navigate to your settings.py file and select it. Alternatively, you can open the file by navigating to the djangogirls folder on your desktop and right-clicking on it. Then, select your code editor from the list. Selecting the editor is important as you might have other programs installed that can open the file but will not let you edit it.

In settings.py, find the line that contains TIME\_ZONE and modify it to choose your own timezone. For example:

mysite/settings.py

TIME\_ZONE = 'Europe/Berlin'

We'll also need to add a path for static files. (We'll find out all about static files and CSS later in the tutorial.) Go down to the *end* of the file, and just underneath the STATIC\_URL entry, add a new one called STATIC\_ROOT:

mysite/settings.py

STATIC\_URL = '/static/'

STATIC\_ROOT = os.path.join(BASE\_DIR, 'static')

When DEBUG is True and ALLOWED\_HOSTS is empty, the host is validated against ['localhost', '127.0.0.1', '[::1]']. This won't match our hostname on PythonAnywhere once we deploy our application so we will change the following setting:

mysite/settings.py

ALLOWED\_HOSTS = ['127.0.0.1', '<your\_username>.pythonanywhere.com']

**Note**: If you're using a Chromebook, add this line at the bottom of your settings.py file: MESSAGE\_STORAGE = 'django.contrib.messages.storage.session.SessionStorage'

**Set up a database**

There's a lot of different database software that can store data for your site. We'll use the default one, sqlite3.

This is already set up in this part of your mysite/settings.py file:

mysite/settings.py

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3', …. Driver / package

'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'), … location (folder, ip address, computer name) , dbname

}

}

To create a database for our blog, let's run the following in the console: python manage.py migrate (we need to be in the djangogirls directory that contains the manage.py file). If that goes well, you should see something like this:

command-line

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

Operations to perform:

Apply all migrations: auth, admin, contenttypes, sessions

Running migrations:

Rendering model states... DONE

Applying contenttypes.0001\_initial... OK

Applying auth.0001\_initial... OK

Applying admin.0001\_initial... OK

Applying admin.0002\_logentry\_remove\_auto\_add... OK

Applying contenttypes.0002\_remove\_content\_type\_name... OK

Applying auth.0002\_alter\_permission\_name\_max\_length... OK

Applying auth.0003\_alter\_user\_email\_max\_length... OK

Applying auth.0004\_alter\_user\_username\_opts... OK

Applying auth.0005\_alter\_user\_last\_login\_null... OK

Applying auth.0006\_require\_contenttypes\_0002... OK

Applying auth.0007\_alter\_validators\_add\_error\_messages... OK

Applying sessions.0001\_initial... OK

**Starting the web server**

You need to be in the directory that contains the manage.py file (the djangogirls directory). In the console, we can start the web server by running python manage.py runserver:

command-line

(myvenv) ~/djangogirls$ python manage.py runserver

If you are on a Chromebook, use this command instead:

Cloud 9

(myvenv) ~/djangogirls$ python manage.py runserver 0.0.0.0:8080

If you are on Windows and this fails with UnicodeDecodeError, use this command instead:

command-line

(myvenv) ~/djangogirls$ python manage.py runserver 0:8000

Now all you need to do is check that your website is running. Open your browser (Firefox, Chrome, Safari, Internet Explorer or whatever you use) and enter this address:

browser

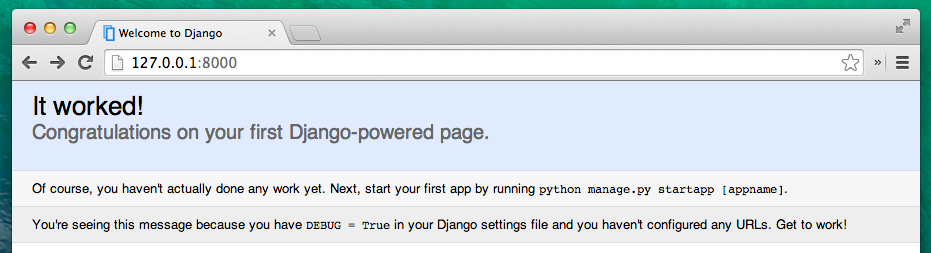
http://127.0.0.1:8000/

If you're using a Chromebook, you'll always visit your test server by accessing:

browser

https://django-girls-<your cloud9 username>.c9users.io

Congratulations! You've just created your first website and run it using a web server! Isn't that awesome?



While the web server is running, you won't see a new command-line prompt to enter additional commands. The terminal will accept new text but will not execute new commands. This is because the web server continuously runs in order to listen for incoming requests.

We reviewed how web servers work in the **How the Internet works** chapter.

To type additional commands while the web server is running, open a new terminal window and activate your virtualenv. To stop the web server, switch back to the window in which it's running and press CTRL+C - Control and C keys together (on Windows, you might have to press Ctrl+Break).

# First Django project!

The first step is to start a new Django project. Basically, this means that we'll run some scripts provided by Django that will create the skeleton of a Django project for us. This is just a bunch of directories and files that we will use later.

The names of some files and directories are very important for Django. You should not rename the files that we are about to create. Moving them to a different place is also not a good idea. Django needs to maintain a certain structure to be able to find important things.

Remember to run everything in the virtualenv. If you don't see a prefix (myvenv) in your console, you need to activate your virtualenv. We explained how to do that in the **Django installation** chapter in the **Working with virtualenv** part. Typing myvenv\Scripts\activate on Windows or source myvenv/bin/activate on Mac OS X or Linux will do this for you.

## Windows

django-admin.py is a script that will create the directories and files for you. You should now have a directory structure which looks like this:

djangogirls

├───manage.py

└───mysite

settings.py

urls.py

wsgi.py

\_\_init\_\_.py

**Note**: in your directory structure, you will also see your venv directory that we created before.

manage.py is a script that helps with management of the site. With it we will be able (amongst other things) to start a web server on our computer without installing anything else.

The settings.py file contains the configuration of your website.

Remember when we talked about a mail carrier checking where to deliver a letter? urls.py file contains a list of patterns used by urlresolver.

Let's ignore the other files for now as we won't change them. The only thing to remember is not to delete them by accident!

## Changing settings

Let's make some changes in mysite/settings.py. Open the file using the code editor you installed earlier.

**Note**: Keep in mind that settings.py is a regular file, like any other. You can open it from inside the code editor, using the "file -> open" menu actions. This should get you the usual window in which you can navigate to your settings.py file and select it. Alternatively, you can open the file by navigating to the djangogirls folder on your desktop and right-clicking on it. Then, select your code editor from the list. Selecting the editor is important as you might have other programs installed that can open the file but will not let you edit it.

In settings.py, find the line that contains TIME\_ZONE and modify it to choose your own timezone. For example:

mysite/settings.py

TIME\_ZONE = 'Europe/Berlin'

We'll also need to add a path for static files. (We'll find out all about static files and CSS later in the tutorial.) Go down to the end of the file, and just underneath the STATIC\_URL entry, add a new one called STATIC\_ROOT:

mysite/settings.py

STATIC\_URL = '/static/'

STATIC\_ROOT = os.path.join(BASE\_DIR, 'static')

When DEBUG is True and ALLOWED\_HOSTS is empty, the host is validated against ['localhost', '127.0.0.1', '[::1]']. This won't match our hostname on PythonAnywhere once we deploy our application so we will change the following setting:

mysite/settings.py

ALLOWED\_HOSTS = ['127.0.0.1', '<your\_username>.pythonanywhere.com']

**Note**: If you're using a Chromebook, add this line at the bottom of your settings.py file: MESSAGE\_STORAGE = 'django.contrib.messages.storage.session.SessionStorage'

## Set up a database

There's a lot of different database software that can store data for your site. We'll use the default one, sqlite3.

This is already set up in this part of your mysite/settings.py file:

mysite/settings.py

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),

}

}

To create a database for our blog, let's run the following in the console: python manage.py migrate (we need to be in the djangogirls directory that contains the manage.py file). If that goes well, you should see something like this:

command-line

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

Operations to perform:

Apply all migrations: auth, admin, contenttypes, sessions

Running migrations:

Rendering model states... DONE

Applying contenttypes.0001\_initial... OK

Applying auth.0001\_initial... OK

Applying admin.0001\_initial... OK

Applying admin.0002\_logentry\_remove\_auto\_add... OK

Applying contenttypes.0002\_remove\_content\_type\_name... OK

Applying auth.0002\_alter\_permission\_name\_max\_length... OK

Applying auth.0003\_alter\_user\_email\_max\_length... OK

Applying auth.0004\_alter\_user\_username\_opts... OK

Applying auth.0005\_alter\_user\_last\_login\_null... OK

Applying auth.0006\_require\_contenttypes\_0002... OK

Applying auth.0007\_alter\_validators\_add\_error\_messages... OK

Applying sessions.0001\_initial... OK

And we're done! Time to start the web server and see if our website is working!

## Starting the web server

You need to be in the directory that contains the manage.py file (the djangogirls directory). In the console, we can start the web server by running python manage.py runserver:

command-line

(myvenv) ~/djangogirls$ python manage.py runserver

If you are on a Chromebook, use this command instead:

Cloud 9

(myvenv) ~/djangogirls$ python manage.py runserver 0.0.0.0:8080

If you are on Windows and this fails with UnicodeDecodeError, use this command instead:

command-line

(myvenv) ~/djangogirls$ python manage.py runserver 0:8000

Now all you need to do is check that your website is running. Open your browser (Firefox, Chrome, Safari, Internet Explorer or whatever you use) and enter this address:

browser

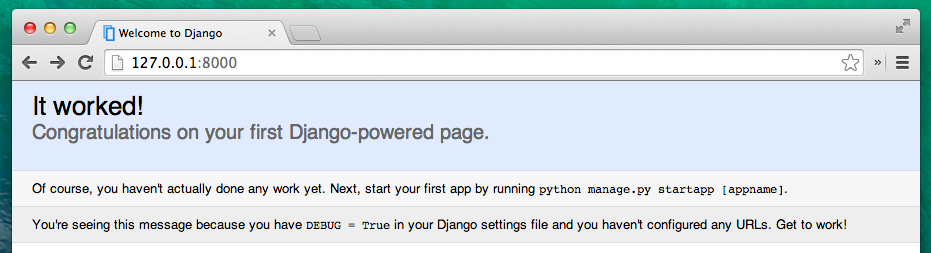
http://127.0.0.1:8000/

If you're using a Chromebook, you'll always visit your test server by accessing:

browser

https://django-girls-<your cloud9 username>.c9users.io

Congratulations! You've just created your first website and run it using a web server! Isn't that awesome?



While the web server is running, you won't see a new command-line prompt to enter additional commands. The terminal will accept new text but will not execute new commands. This is because the web server continuously runs in order to listen for incoming requests.

We reviewed how web servers work in the **How the Internet works** chapter.

To type additional commands while the web server is running, open a new terminal window and activate your virtualenv. To stop the web server, switch back to the window in which it's running and press CTRL+C - Control and C keys together (on Windows, you might have to press Ctrl+Break).

# Django admin

To add, edit and delete the posts we've just modeled, we will use Django admin.

Let's open the blog/admin.py file and replace its contents with this:

blog/admin.py

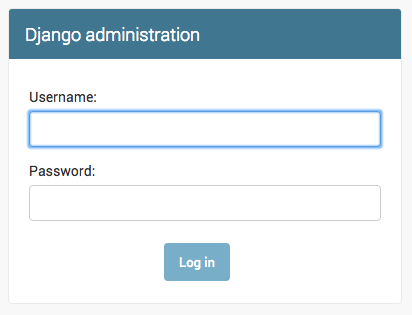
from django.contrib import admin

from .models import Post

admin.site.register(Post)

As you can see, we import (include) the Post model defined in the previous chapter. To make our model visible on the admin page, we need to register the model with admin.site.register(Post).

OK, time to look at our Post model. Remember to run python manage.py runserver in the console to run the web server. Go to your browser and type the address <http://127.0.0.1:8000/admin/>. You will see a login page like this:



To log in, you need to create a superuser - a user account that has control over everything on the site. Go back to the command line, type python manage.py createsuperuser, and press enter.

Remember, to write new commands while the web server is running, open a new terminal window and activate your virtualenv. We reviewed how to write new commands in the **Your first Django project!** chapter, in the **Starting the web server** section.

When prompted, type your username (lowercase, no spaces), email address, and password. Don't worry that you can't see the password you're typing in – that's how it's supposed to be. Just type it in and press enter to continue. The output should look like this (where the username and email should be your own ones):

command-line

(myvenv) ~/djangogirls$ python manage.py createsuperuser

Username: admin

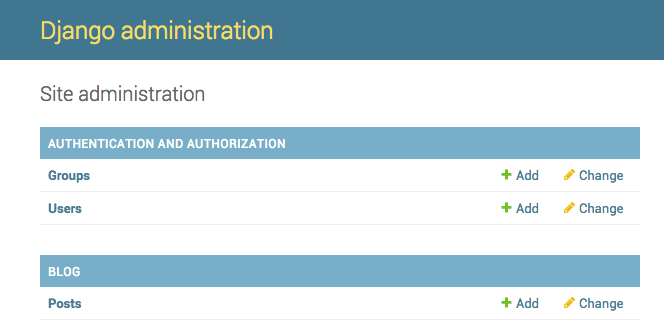
Email address: admin@admin.com

Password:

Password (again):

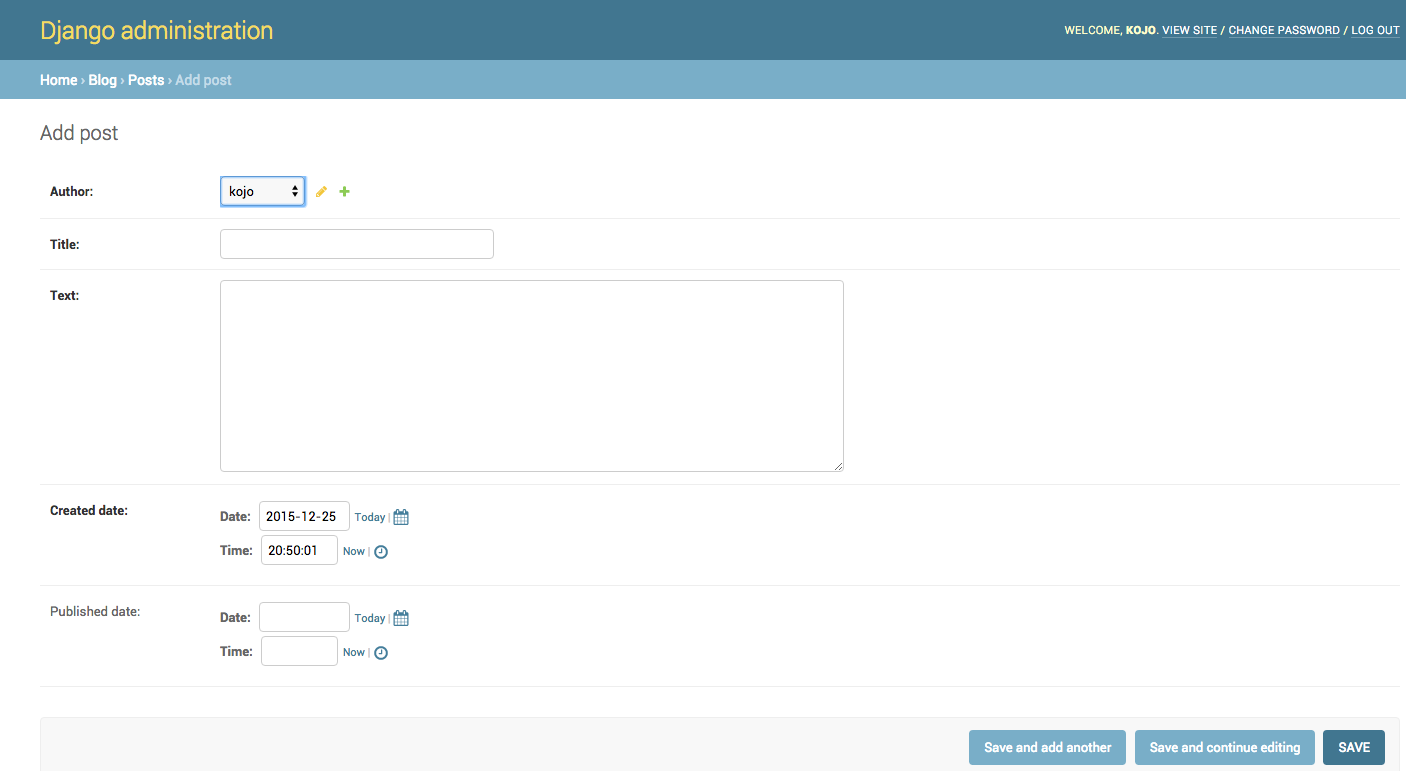
Superuser created successfully.

Return to your browser. Log in with the superuser's credentials you chose; you should see the Django admin dashboard.



Go to Posts and experiment a little bit with it. Add five or six blog posts. Don't worry about the content – you can simply copy-paste some text from this tutorial to save time. :)

Make sure that at least two or three posts (but not all) have the publish date set. It will be helpful later.



This is probably a good moment to grab a coffee (or tea) or something to eat to re-energize yourself. You created your first Django model – you deserve a little break!

# Deploy!

**Note** The following chapter can be sometimes a bit hard to get through. Persist and finish it; deployment is an important part of the website development process. This chapter is placed in the middle of the tutorial so that your mentor can help with the slightly trickier process of getting your website online. This means you can still finish the tutorial on your own if you run out of time.

Until now, your website was only available on your computer. Now you will learn how to deploy it! Deploying is the process of publishing your application on the Internet so people can finally go and see your app. :)

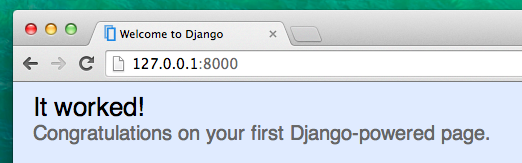
These three places will be important to you. Your local computer will be the place where you do development and testing. When you're happy with the changes, you will place a copy of your program on GitHub. Your website will be on PythonAnywhere and you will update it by getting a new copy of your code from GitHub.

# Django URLs

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 blog/urls.py. 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 = [

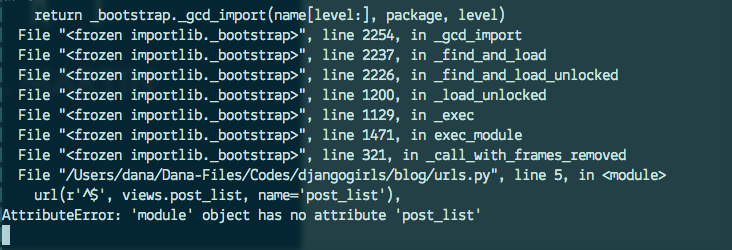
url(r'^$', views.post\_list, name='post\_list'),

]

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.



# Django views – time to create!

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

# Create your views here.

Not too much stuff here yet.

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

The simplest view can look like this:

blog/views.py

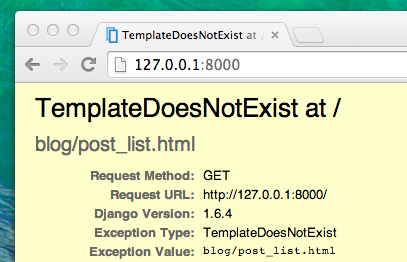
def post\_list(request):

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!

# Introduction to HTML

What's a template, you may ask?

A template is a file that we can re-use to present different information in a consistent format – for example, you could use a template to help you write a letter, because although each letter might contain a different message and be addressed to a different person, they will share the same format.

A Django template's format is described in a language called HTML (that's the HTML we mentioned in the first chapter, **How the Internet works**).

## What is HTML?

HTML is a simple code that is interpreted by your web browser – such as Chrome, Firefox or Safari – to display a web page for the user.

HTML stands for "HyperText Markup Language". **HyperText** means it's a type of text that supports hyperlinks between pages. **Markup** means we have taken a document and marked it up with code to tell something (in this case, a browser) how to interpret the page. HTML code is built with **tags**, each one starting with < and ending with >. These tags represent markup **elements**.

## Your first template!

Creating a template means creating a template file. Everything is a file, right? You have probably noticed this already.

Templates are saved in blog/templates/blog directory. So first create a directory called templates inside your blog directory. Then create another directory called blog inside your templates directory:

blog

└───templates

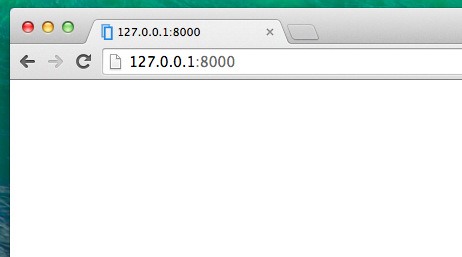
└───blog

(You might wonder why we need two directories both called blog – as you will discover later, this is simply a useful naming convention that makes life easier when things start to get more complicated.)

And now create a post\_list.html file (just leave it blank for now) inside the blog/templates/blog directory.

See how your website looks now: <http://127.0.0.1:8000/>

If you still have an error TemplateDoesNotExist, try to restart your server. Go into command line, stop the server by pressing Ctrl+C (Control and C keys together) and start it again by running a python manage.py runserver command.



No error anymore! Congratulations :) However, your website isn't actually publishing anything except an empty page, because your template is empty too. We need to fix that.

Add the following to your template file:

blog/templates/blog/post\_list.html

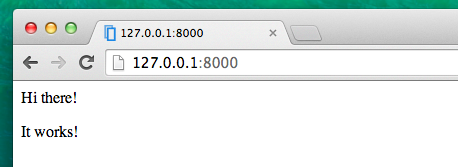
<html>

<p>Hi there!</p>

<p>It works!</p>

</html>

So how does your website look now? Visit it to find out: <http://127.0.0.1:8000/>



It worked! Nice work there :)

* The most basic tag, <html>, is always the beginning of any web page and </html> is always the end. As you can see, the whole content of the website goes between the beginning tag <html> and closing tag </html>
* <p> is a tag for paragraph elements; </p> closes each paragraph

## Head and body

Each HTML page is also divided into two elements: **head** and **body**.

* **head** is an element that contains information about the document that is not displayed on the screen.
* **body** is an element that contains everything else that is displayed as part of the web page.

We use <head> to tell the browser about the configuration of the page, and <body> to tell it what's actually on the page.

For example, you can put a web page title element inside the <head>, like this:

blog/templates/blog/post\_list.html

<html>

<head>

<title>Ola's blog</title>

</head>

<body>

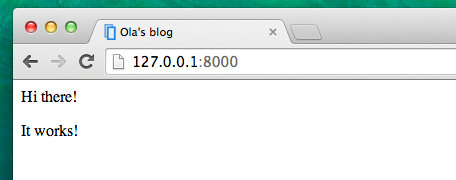
<p>Hi there!</p>

<p>It works!</p>

</body>

</html>

Save the file and refresh your page.



Notice how the browser has understood that "Ola's blog" is the title of your page? It has interpreted <title>Ola's blog</title> and placed the text in the title bar of your browser (it will also be used for bookmarks and so on).

Probably you have also noticed that each opening tag is matched by a closing tag, with a /, and that elements are nested (i.e. you can't close a particular tag until all the ones that were inside it have been closed too).

It's like putting things into boxes. You have one big box, <html></html>; inside it there is <body></body>, and that contains still smaller boxes: <p></p>.

You need to follow these rules of closing tags, and of nesting elements – if you don't, the browser may not be able to interpret them properly and your page will display incorrectly.

## Customize your template

You can now have a little fun and try to customize your template! Here are a few useful tags for that:

* <h1>A heading</h1> for your most important heading
* <h2>A sub-heading</h2> for a heading at the next level
* <h3>A sub-sub-heading</h3> …and so on, up to <h6>
* <p>A paragraph of text</p>
* <em>text</em> emphasizes your text
* <strong>text</strong> strongly emphasizes your text
* <br /> goes to another line (you can't put anything inside br)
* <a href="https://djangogirls.org">link</a> creates a link
* <ul><li>first item</li><li>second item</li></ul> makes a list, just like this one!
* <div></div> defines a section of the page

Here's an example of a full template, copy and paste it into blog/templates/blog/post\_list.html:

blog/templates/blog/post\_list.html

<html>

<head>

<title>Django Girls blog</title>

</head>

<body>

<div>

<h1><a href="">Django Girls Blog</a></h1>

</div>

<div>

<p>published: 14.06.2014, 12:14</p>

<h2><a href="">My first post</a></h2>

<p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus.</p>

</div>

<div>

<p>published: 14.06.2014, 12:14</p>

<h2><a href="">My second post</a></h2>

<p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut f.</p>

</div>

</body>

</html>

We've created three div sections here.

* The first div element contains the title of our blog – it's a heading and a link
* Another two div elements contain our blogposts with a published date, h2 with a post title that is clickable and two ps (paragraph) of text, one for the date and one for our blogpost.

It gives us this effect:



Yaaay! But so far, our template only ever displays exactly **the same information** – whereas earlier we were talking about templates as allowing us to display **different** information in the **same format**.

What we really want to do is display real posts added in our Django admin – and that's where we're going next.

## One more thing: deploy!

It'd be good to see all this out and live on the Internet, right? Let's do another PythonAnywhere deploy:

### Commit, and push your code up to Github

First off, let's see what files have changed since we last deployed (run these commands locally, not on PythonAnywhere):

command-line

$ git status

Make sure you're in the djangogirls directory and let's tell git to include all the changes within this directory:

command-line

$ git add --all .

--all means that git will also recognize if you've deleted files (by default, it only recognizes new/modified files). Also remember (from chapter 3) that . means the current directory.

Before we upload all the files, let's check what git will be uploading (all the files that git will upload should now appear in green):

command-line

$ git status

We're almost there, now it's time to tell it to save this change in its history. We're going to give it a "commit message" where we describe what we've changed. You can type anything you'd like at this stage, but it's helpful to type something descriptive so that you can remember what you've done in the future.

command-line

$ git commit -m "Changed the HTML for the site."

Make sure you use double quotes around the commit message.

Once we've done that, we upload (push) our changes up to GitHub:

command-line

$ git push

command-line

$ cd ~/my-first-blog

$ git pull

[...]

And watch your code get downloaded. If you want to check that it's arrived, you can hop over to the **Files tab** and view your code on PythonAnywhere.

* Finally, hop on over to the [Web tab](https://www.pythonanywhere.com/web_app_setup/) and hit **Reload** on your web app.

Your update should be live! Go ahead and refresh your website in the browser. Changes should be visible. :)

# Dynamic data in templates

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](https://tutorial.djangogirls.org/en/django_orm/).

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('published\_date')

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('published\_date')

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('published\_date')

return render(request, 'blog/post\_list.html', {'posts': posts})

**Django templates**

Time to display some data! Django gives us some helpful built-in **template tags** for that.

**What are template tags?**

You see, in HTML, you can't really write Python code, because browsers don't understand it. They know only HTML. We know that HTML is rather static, while Python is much more dynamic.

**Django template tags** allow us to transfer Python-like things into HTML, so you can build dynamic websites faster and easier. Cool!

**Display post list template**

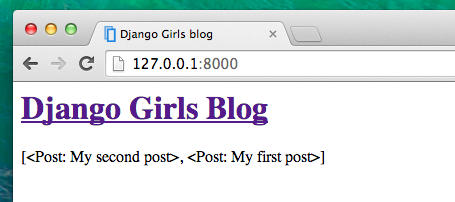
In the previous chapter we gave our template a list of posts in the posts variable. Now we will display it in HTML.

To print a variable in Django templates, we use double curly brackets with the variable's name inside, like this:

blog/templates/blog/post\_list.html

{{ posts }}

Try this in your blog/templates/blog/post\_list.html template. Replace everything from the second <div> to the third </div> with {{ posts }}. Save the file, and refresh the page to see the results:



As you can see, all we've got is this:

blog/templates/blog/post\_list.html

<QuerySet [<Post: My second post>, <Post: My first post>]>

This means that Django understands it as a list of objects. Remember from **Introduction to Python** how we can display lists? Yes, with for loops! In a Django template you do them like this:

blog/templates/blog/post\_list.html

{% for post in posts %}

{{ post }}

{% endfor %}

Try this in your template.



It works! But we want the posts to be displayed like the static posts we created earlier in the **Introduction to HTML** chapter. You can mix HTML and template tags. Our body will look like this:

blog/templates/blog/post\_list.html

<div>

<h1><a href="/">Django Girls Blog</a></h1>

</div>

{% for post in posts %}

<div>

<p>published: {{ post.published\_date }}</p>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endfor %}

Everything you put between {% for %} and {% endfor %} will be repeated for each object in the list. Refresh your page:



Have you noticed that we used a slightly different notation this time ({{ post.title }} or {{ post.text }})? We are accessing data in each of the fields defined in our Post model. Also, the |linebreaksbr is piping the posts' text through a filter to convert line-breaks into paragraphs.

**One more thing**

It'd be good to see if your website will still be working on the public Internet, right? Let's try deploying to PythonAnywhere again. Here's a recap of the steps…

* First, push your code to Github

command-line

$ git status

[...]

$ git add --all .

$ git status

[...]

$ git commit -m "Modified templates to display posts from database."

[...]

$ git push

PythonAnywhere command-line

$ cd my-first-blog

$ git pull

[...]

* Finally, hop on over to the Web tab and hit **Reload** on your web app. Your update should be live! If the blog posts on your PythonAnywhere site don't match the posts appearing on the blog hosted on your local server, that's OK. The databases on your local computer and Python Anywhere don't sync with the rest of your files.

# CSS – make it pretty!

Our blog still looks pretty ugly, right? Time to make it nice! We will use CSS for that.

## What is CSS?

Cascading Style Sheets (CSS) is a language used for describing the look and formatting of a website written in a markup language (like HTML). Treat it as make-up for our web page. ;)

But we don't want to start from scratch again, right? Once more, we'll use something that programmers released on the Internet for free. Reinventing the wheel is no fun, you know.

## Let's use Bootstrap!

Bootstrap is one of the most popular HTML and CSS frameworks for developing beautiful websites: <https://getbootstrap.com/>

It was written by programmers who worked for Twitter. Now it's developed by volunteers from all over the world!

## Install Bootstrap

To install Bootstrap, you need to add this to your <head> in your .html file:

blog/templates/blog/post\_list.html

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">

This doesn't add any files to your project. It just points to files that exist on the Internet. Just go ahead, open your website and refresh the page. Here it is!



Looking nicer already!

## Static files in Django

Finally we will take a closer look at these things we've been calling **static files**. Static files are all your CSS and images. Their content doesn't depend on the request context and will be the same for every user.

### Where to put static files for Django

Django already knows where to find the static files for the built-in "admin" app. Now we just need to add some static files for our own app, blog.

We do that by creating a folder called static inside the blog app:

djangogirls

├── blog

│ ├── migrations

│ └── static

└── mysite

Django will automatically find any folders called "static" inside any of your apps' folders. Then it will be able to use their contents as static files.

## Your first CSS file!

Let's create a CSS file now, to add your own style to your web page. Create a new directory called css inside your static directory. Then create a new file called blog.css inside this css directory. Ready?

djangogirls

└─── blog

└─── static

└─── css

└─── blog.css

Time to write some CSS! Open up the blog/static/css/blog.css file in your code editor.

We won't be going too deep into customizing and learning about CSS here. It's pretty easy and you can learn it on your own after this workshop. There is a recommendation for a free course to learn more at the end of this page.

In your blog/static/css/blog.css file you should add the following code:

blog/static/css/blog.css

h1 a {

color: #FCA205;

}

h1 a is a CSS Selector. This means we're applying our styles to any a element inside of an h1 element. So when we have something like <h1><a href="">link</a></h1>, the h1 a style will apply. In this case, we're telling it to change its color to #FCA205, which is orange. Of course, you can put your own color here!

In a CSS file we determine styles for elements in the HTML file. The first way we identify elements is with the element name. You might remember these as tags from the HTML section. Things like a, h1, and body are all examples of element names. We also identify elements by the attribute class or the attribute id. Class and id are names you give the element by yourself. Classes define groups of elements, and ids point to specific elements. For example, you could identify the following tag by using the tag name a, the class external\_link, or the id link\_to\_wiki\_page:

<a href="https://en.wikipedia.org/wiki/Django" class="external\_link" id="link\_to\_wiki\_page">

We also need to tell our HTML template that we added some CSS. Open the blog/templates/blog/post\_list.html file and add this line at the very beginning of it:

blog/templates/blog/post\_list.html

{% load staticfiles %}

We're just loading static files here. :) Between the <head> and </head> tags, after the links to the Bootstrap CSS files, add this line:

blog/templates/blog/post\_list.html

<link rel="stylesheet" href="{% static 'css/blog.css' %}">

The browser reads the files in the order they're given, so we need to make sure this is in the right place. Otherwise the code in our file may override code in Bootstrap files. We just told our template where our CSS file is located.

Your file should now look like this:

blog/templates/blog/post\_list.html

{% load staticfiles %}

<html>

<head>

<title>Django Girls blog</title>

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">

<link rel="stylesheet" href="{% static 'css/blog.css' %}">

</head>

<body>

<div>

<h1><a href="/">Django Girls Blog</a></h1>

</div>

{% for post in posts %}

<div>

<p>published: {{ post.published\_date }}</p>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endfor %}

</body>

</html>

OK, save the file and refresh the site!



Nice work! Maybe we would also like to give our website a little air and increase the margin on the left side? Let's try this!

blog/static/css/blog.css

body {

padding-left: 15px;

}

Add that to your CSS, save the file and see how it works!



Maybe we can customize the font in our header? Paste this into your <head> in blog/templates/blog/post\_list.html file:

blog/templates/blog/post\_list.html

<link href="//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext" rel="stylesheet" type="text/css">

As before, check the order and place before the link to blog/static/css/blog.css. This line will import a font called Lobster from Google Fonts (<https://www.google.com/fonts>).

Find the h1 a declaration block (the code between braces { and }) in the CSS file blog/static/css/blog.css. Now add the line font-family: 'Lobster'; between the braces, and refresh the page:

blog/static/css/blog.css

h1 a {

color: #FCA205;

font-family: 'Lobster';

}



Great!

As mentioned above, CSS has a concept of classes. These allow you to name a part of the HTML code and apply styles only to this part, without affecting other parts. This can be super helpful! Maybe you have two divs that are doing something different (like your header and your post). A class can help you make them look different.

Go ahead and name some parts of the HTML code. Add a class called page-header to your div that contains your header, like this:

blog/templates/blog/post\_list.html

<div class="page-header">

<h1><a href="/">Django Girls Blog</a></h1>

</div>

And now add a class post to your div containing a blog post.

blog/templates/blog/post\_list.html

<div class="post">

<p>published: {{ post.published\_date }}</p>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

We will now add declaration blocks to different selectors. Selectors starting with . relate to classes. There are many great tutorials and explanations about CSS on the Web that can help you understand the following code. For now, just copy and paste it into your blog/static/css/blog.css file:

blog/static/css/blog.css

.page-header {

background-color: #ff9400;

margin-top: 0;

padding: 20px 20px 20px 40px;

}

.page-header h1, .page-header h1 a, .page-header h1 a:visited, .page-header h1 a:active {

color: #ffffff;

font-size: 36pt;

text-decoration: none;

}

.content {

margin-left: 40px;

}

h1, h2, h3, h4 {

font-family: 'Lobster', cursive;

}

.date {

color: #828282;

}

.save {

float: right;

}

.post-form textarea, .post-form input {

width: 100%;

}

.top-menu, .top-menu:hover, .top-menu:visited {

color: #ffffff;

float: right;

font-size: 26pt;

margin-right: 20px;

}

.post {

margin-bottom: 70px;

}

.post h1 a, .post h1 a:visited {

color: #000000;

}

Then surround the HTML code which displays the posts with declarations of classes. Replace this:

blog/templates/blog/post\_list.html

{% for post in posts %}

<div class="post">

<p>published: {{ post.published\_date }}</p>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endfor %}

in the blog/templates/blog/post\_list.html with this:

blog/templates/blog/post\_list.html

<div class="content container">

<div class="row">

<div class="col-md-8">

{% for post in posts %}

<div class="post">

<div class="date">

<p>published: {{ post.published\_date }}</p>

</div>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endfor %}

</div>

</div>

</div>

Save those files and refresh your website.



# emplate extending

Another nice thing Django has for you is **template extending**. What does this mean? It means that you can use the same parts of your HTML for different pages of your website.

Templates help when you want to use the same information or layout in more than one place. You don't have to repeat yourself in every file. And if you want to change something, you don't have to do it in every template, just one!

## Create a base template

A base template is the most basic template that you extend on every page of your website.

Let's create a base.html file in blog/templates/blog/:

blog

└───templates

└───blog

base.html

post\_list.html

Then open it up and copy everything from post\_list.html to base.html file, like this:

blog/templates/blog/base.html

{% load staticfiles %}

<html>

<head>

<title>Django Girls blog</title>

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">

<link href='//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext' rel='stylesheet' type='text/css'>

<link rel="stylesheet" href="{% static 'css/blog.css' %}">

</head>

<body>

<div class="page-header">

<h1><a href="/">Django Girls Blog</a></h1>

</div>

<div class="content container">

<div class="row">

<div class="col-md-8">

{% 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 %}

</div>

</div>

</div>

</body>

</html>

Then in base.html, replace your whole <body> (everything between <body> and </body>) with this:

blog/templates/blog/base.html

<body>

<div class="page-header">

<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>

You might notice this replaced everything from {% for post in posts %} to {% endfor %} with:

blog/templates/blog/base.html

{% block content %}

{% endblock %}

But why? You just created a block! You used the template tag {% block %} to make an area that will have HTML inserted in it. That HTML will come from another template that extends this template (base.html). We will show you how to do this in a moment.

Now save base.html and open your blog/templates/blog/post\_list.html again. You're going to remove everything above {% for post in posts %} and below {% endfor %}. When you're done, the file will look like this:

blog/templates/blog/post\_list.html

{% 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 %}

We want to use this as part of our template for all the content blocks. Time to add block tags to this file!

You want your block tag to match the tag in your base.html file. You also want it to include all the code that belongs in your content blocks. To do that, put everything between {% block content %} and {% endblock content %}. Like this:

blog/templates/blog/post\_list.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 %}

Only one thing left. We need to connect these two templates together. This is what extending templates is all about! We'll do this by adding an extends tag to the beginning of the file. 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 %}

**Extend your application**

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 content %}

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

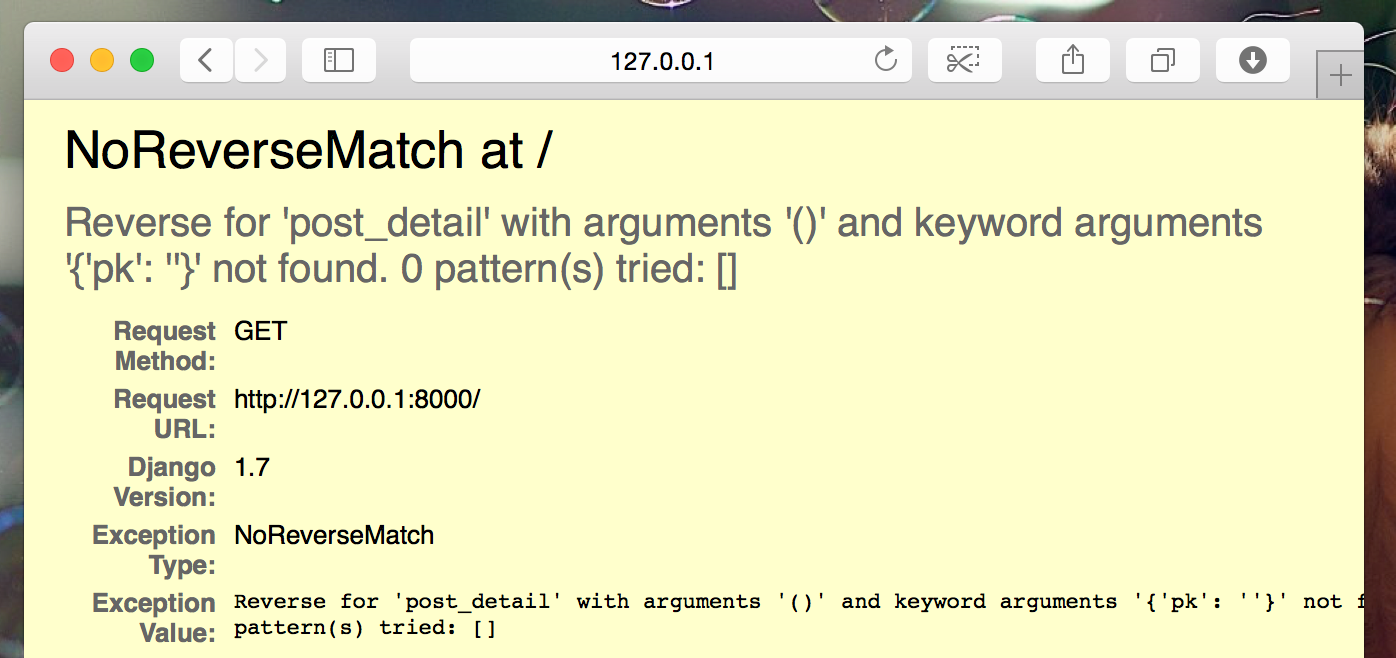
<h1><a href="{% url 'post\_detail' pk=post.pk %}">{{ post.title }}</a></h1>

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!

blog.views.post\_detail is a path to a post\_detail *view* we want to create. Please note: blog is the name of our application (the directory blog), views is from the name of the views.py file and the last bit – post\_detail – is the name of the *view*.

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 don't 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'),

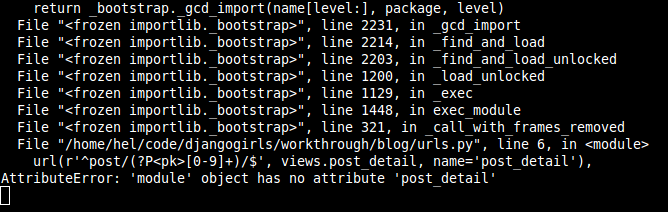
]

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!



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!

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*:

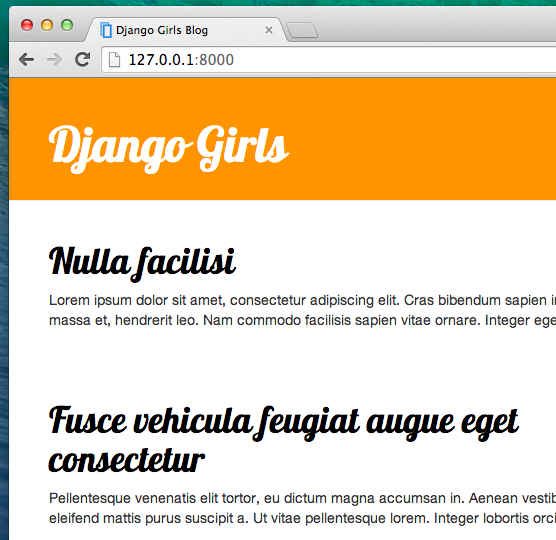
blog/views.py

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

command-line

$ cd my-first-blog

$ git pull

[...]

**Django Forms**

The final thing we want to do on our website is create a nice way to add and edit blog posts. Django's admin is cool, but it is rather hard to customize and make pretty. With forms we will have absolute power over our interface – we can do almost anything we can imagine!

The nice thing about Django forms is that we can either define one from scratch or create a ModelForm which will save the result of the form to the model.

This is exactly what we want to do: we will create a form for our Post model.

Like every important part of Django, forms have their own file: forms.py.

We need to create a file with this name in the blog directory.

blog

└── forms.py

OK, let's open it and type the following code:

blog/forms.py

from django import forms

from .models import Post

class PostForm(forms.ModelForm):

class Meta:

model = Post

fields = ('title', 'text',)

We need to import Django forms first (from django import forms) and, obviously, our Post model (from .models import Post).

PostForm, as you probably suspect, is the name of our form. We need to tell Django that this form is a ModelForm (so Django will do some magic for us) – forms.ModelForm is responsible for that.

Next, we have class Meta, where we tell Django which model should be used to create this form (model = Post).

Finally, we can say which field(s) should end up in our form. In this scenario we want only title and text to be exposed – author should be the person who is currently logged in (you!) and created\_date should be automatically set when we create a post (i.e. in the code), right?

And that's it! All we need to do now is use the form in a *view* and display it in a template.

So once again we will create a link to the page, a URL, a view and a template.

**Link to a page with the form**

It's time to open blog/templates/blog/base.html. We will add a link in div named page-header:

blog/templates/blog/base.html

<a href="{% url 'post\_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>

Note that we want to call our new view post\_new. The class "glyphicon glyphicon-plus" is provided by the bootstrap theme we are using, and will display a plus sign for us.

After adding the line, your HTML file should now look like this:

blog/templates/blog/base.html

{% load staticfiles %}

<html>

<head>

<title>Django Girls blog</title>

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">

<link href='//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext' rel='stylesheet' type='text/css'>

<link rel="stylesheet" href="{% static 'css/blog.css' %}">

</head>

<body>

<div class="page-header">

<a href="{% url 'post\_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>

<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>

</html>

After saving and refreshing the page <http://127.0.0.1:8000> you will obviously see a familiar NoReverseMatch error, right?

**URL**

We open blog/urls.py and add a line:

blog/urls.py

url(r'^post/new/$', views.post\_new, name='post\_new'),

And the final code will 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'),

url(r'^post/new/$', views.post\_new, name='post\_new'),

]

After refreshing the site, we see an AttributeError, since we don't have the post\_new view implemented. Let's add it right now.

**post\_new view**

Time to open the blog/views.py file and add the following lines with the rest of the from rows:

blog/views.py

from .forms import PostForm

And then our *view*:

blog/views.py

def post\_new(request):

form = PostForm()

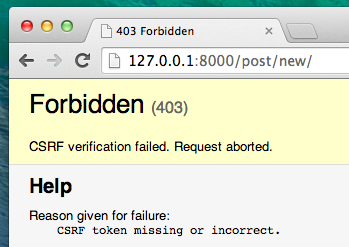
return render(request, 'blog/post\_edit.html', {'form': form})

To create a new Post form, we need to call PostForm() and pass it to the template. We will go back to this *view*, but for now, let's quickly create a template for the form.

**Template**

We need to create a file post\_edit.html in the blog/templates/blog directory. To make a form work we need several things:

* We have to display the form. We can do that with (for example) a simple {{ form.as\_p }}.
* The line above needs to be wrapped with an HTML form tag: <form method="POST">...</form>.
* We need a Save button. We do that with an HTML button: <button type="submit">Save</button>.
* And finally, just after the opening <form ...> tag we need to add {% csrf\_token %}. This is very important, since it makes your forms secure! If you forget about this bit, Django will complain when you try to save the form:



OK, so let's see how the HTML in post\_edit.html should look:

blog/templates/blog/post\_edit.html

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

{% block content %}

<h1>New post</h1>

<form method="POST" class="post-form">{% csrf\_token %}

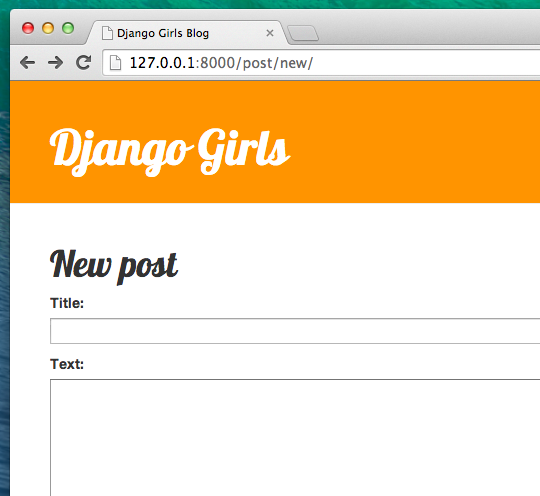
{{ form.as\_p }}

<button type="submit" class="save btn btn-default">Save</button>

</form>

{% endblock %}

Time to refresh! Yay! Your form is displayed!



But, wait a minute! When you type something in the title and text fields and try to save it, what will happen?

Nothing! We are once again on the same page and our text is gone… and no new post is added. So what went wrong?

The answer is: nothing. We need to do a little bit more work in our *view*.

**Saving the form**

Open blog/views.py once again. Currently all we have in the post\_new view is the following:

blog/views.py

def post\_new(request):

form = PostForm()

return render(request, 'blog/post\_edit.html', {'form': form})

When we submit the form, we are brought back to the same view, but this time we have some more data in request, more specifically in request.POST (the naming has nothing to do with a blog "post"; it's to do with the fact that we're "posting" data). Remember how in the HTML file, our <form> definition had the variable method="POST"? All the fields from the form are now in request.POST. You should not rename POST to anything else (the only other valid value for method is GET, but we have no time to explain what the difference is).

So in our *view* we have two separate situations to handle: first, when we access the page for the first time and we want a blank form, and second, when we go back to the *view* with all form data we just typed. So we need to add a condition (we will use if for that):

blog/views.py

if request.method == "POST":

[...]

else:

form = PostForm()

It's time to fill in the dots [...]. If method is POST then we want to construct the PostForm with data from the form, right? We will do that as follows:

blog/views.py

form = PostForm(request.POST)

Easy! The next thing is to check if the form is correct (all required fields are set and no incorrect values have been submitted). We do that with form.is\_valid().

We check if the form is valid and if so, we can save it!

blog/views.py

if form.is\_valid():

post = form.save(commit=False)

post.author = request.user

post.published\_date = timezone.now()

post.save()

Basically, we have two things here: we save the form with form.save and we add an author (since there was no author field in the PostForm and this field is required). commit=False means that we don't want to save the Post model yet – we want to add the author first. Most of the time you will use form.save() without commit=False, but in this case, we need to supply it. post.save() will preserve changes (adding the author) and a new blog post is created!

Finally, it would be awesome if we could immediately go to the post\_detail page for our newly created blog post, right? To do that we need one more import:

blog/views.py

from django.shortcuts import redirect

Add it at the very beginning of your file. And now we can say, "go to the post\_detail page for the newly created post":

blog/views.py

return redirect('post\_detail', pk=post.pk)

post\_detail is the name of the view we want to go to. Remember that this *view* requires a pk variable? To pass it to the views, we use pk=post.pk, where post is the newly created blog post!

OK, we've talked a lot, but we probably want to see what the whole *view* looks like now, right?

blog/views.py

def post\_new(request):

if request.method == "POST":

form = PostForm(request.POST)

if form.is\_valid():

post = form.save(commit=False)

post.author = request.user

post.published\_date = timezone.now()

post.save()

return redirect('post\_detail', pk=post.pk)

else:

form = PostForm()

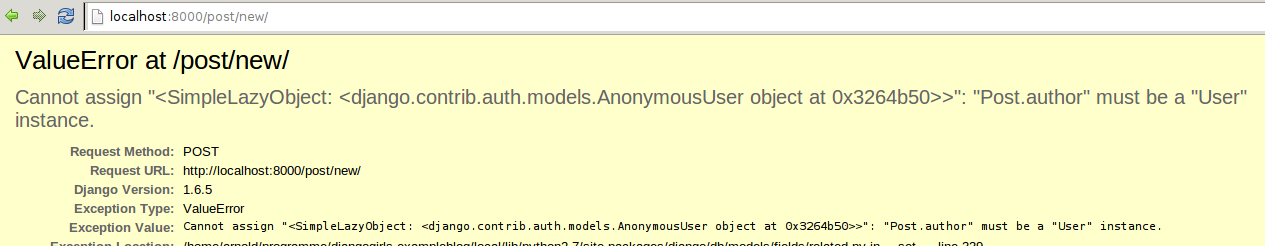
return render(request, 'blog/post\_edit.html', {'form': form})

Let's see if it works. Go to the page <http://127.0.0.1:8000/post/new/>, add a title and text, save it… and voilà! The new blog post is added and we are redirected to the post\_detail page!

You might have noticed that we are setting the publish date before saving the post. Later on, we will introduce a *publish button* in **Django Girls Tutorial: Extensions**.

That is awesome!

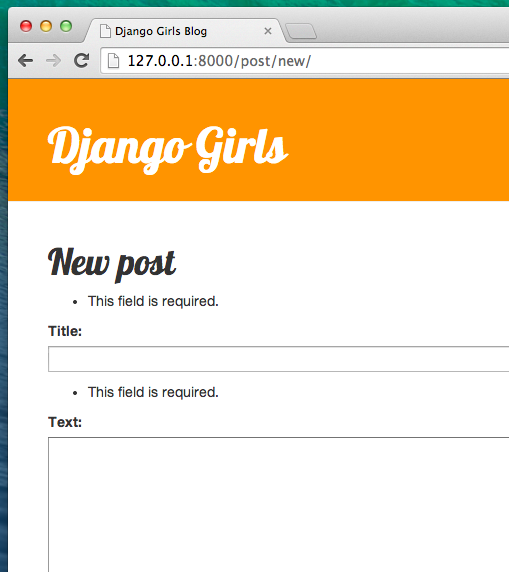
As we have recently used the Django admin interface, the system currently thinks we are still logged in. There are a few situations that could lead to us being logged out (closing the browser, restarting the DB, etc.). If, when creating a post, you find that you are getting errors referring to the lack of a logged-in user, head to the admin page <http://127.0.0.1:8000/admin> and log in again. This will fix the issue temporarily. There is a permanent fix awaiting you in the **Homework: add security to your website!** chapter after the main tutorial.



**Form validation**

Now, we will show you how cool Django forms are. A blog post needs to have title and text fields. In our Post model we did not say that these fields (as opposed to published\_date) are not required, so Django, by default, expects them to be set.

Try to save the form without title and text. Guess what will happen!



Django is taking care to validate that all the fields in our form are correct. Isn't it awesome?

**Edit form**

Now we know how to add a new form. But what if we want to edit an existing one? This is very similar to what we just did. Let's create some important things quickly. (If you don't understand something, you should ask your coach or look at the previous chapters, since we covered all these steps already.)

Open blog/templates/blog/post\_detail.html and add the line

blog/templates/blog/post\_detail.html

<a class="btn btn-default" href="{% url 'post\_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>

so that the template 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 %}

<a class="btn btn-default" href="{% url 'post\_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>

<h1>{{ post.title }}</h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endblock %}

In blog/urls.py we add this line:

blog/urls.py

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

We will reuse the template blog/templates/blog/post\_edit.html, so the last missing thing is a *view*.

Let's open blog/views.py and add this at the very end of the file:

blog/views.py

def post\_edit(request, pk):

post = get\_object\_or\_404(Post, pk=pk)

if request.method == "POST":

form = PostForm(request.POST, instance=post)

if form.is\_valid():

post = form.save(commit=False)

post.author = request.user

post.published\_date = timezone.now()

post.save()

return redirect('post\_detail', pk=post.pk)

else:

form = PostForm(instance=post)

return render(request, 'blog/post\_edit.html', {'form': form})

This looks almost exactly the same as our post\_new view, right? But not entirely. For one, we pass an extra pk parameter from urls. Next, we get the Post model we want to edit with get\_object\_or\_404(Post, pk=pk) and then, when we create a form, we pass this post as an instance, both when we save the form…

blog/views.py

form = PostForm(request.POST, instance=post)

…and when we've just opened a form with this post to edit:

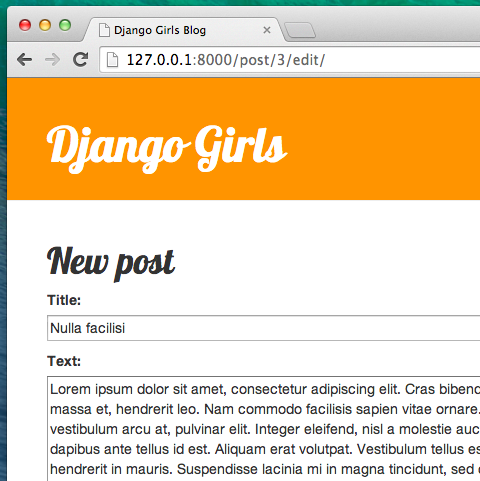
blog/views.py

form = PostForm(instance=post)

OK, let's test if it works! Let's go to the post\_detail page. There should be an edit button in the top-right corner:



When you click it you will see the form with our blog post:



Feel free to change the title or the text and save the changes!

Congratulations! Your application is getting more and more complete!

If you need more information about Django forms, you should read the documentation: <https://docs.djangoproject.com/en/1.10/topics/forms/>

**Security**

Being able to create new posts just by clicking a link is awesome! But right now, anyone who visits your site will be able to make a new blog post, and that's probably not something you want. Let's make it so the button shows up for you but not for anyone else.

In blog/templates/blog/base.html, find our page-header div and the anchor tag you put in there earlier. It should look like this:

blog/templates/blog/base.html

<a href="{% url 'post\_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>

We're going to add another {% if %} tag to this, which will make the link show up only for users who are logged into the admin. Right now, that's just you! Change the <a> tag to look like this:

blog/templates/blog/base.html

{% if user.is\_authenticated %}

<a href="{% url 'post\_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>

{% endif %}

This {% if %} will cause the link to be sent to the browser only if the user requesting the page is logged in. This doesn't protect the creation of new posts completely, but it's a good first step. We'll cover more security in the extension lessons.

Remember the edit icon we just added to our detail page? We also want to add the same change there, so other people won't be able to edit existing posts.

Open blog/templates/blog/post\_detail.html and find this line:

blog/templates/blog/post\_detail.html

<a class="btn btn-default" href="{% url 'post\_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>

Change it to this:

blog/templates/blog/post\_detail.html

{% if user.is\_authenticated %}

<a class="btn btn-default" href="{% url 'post\_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>

{% endif %}

Since you're likely logged in, if you refresh the page, you won't see anything different. Load the page in a different browser or an incognito window, though, and you'll see that the link doesn't show up, and the icon doesn't display either!

**One more thing: deploy time!**

Let's see if all this works on PythonAnywhere. Time for another deploy!

* First, commit your new code, and push it up to Github:

command-line

$ git status

$ git add --all .

$ git status

$ git commit -m "Added views to create/edit blog post inside the site."

$ git push

command-line

$ cd my-first-blog

$ git pull

[...]