**DJANGO**

**DJANGO INSTALLATION**

**Virtual environment**

Before we install Django we will get you to install an extremely useful tool to help keep your coding environment tidy on your computer. It's possible to skip this step, but it's highly recommended. Starting with the best possible setup will save you a lot of trouble in the future!

So, let's create a virtual environment (also called a *virtualenv*). Virtualenv will isolate your Python/Django setup on a per-project basis. This means that any changes you make to one website won't affect any others you're also developing. Neat, right?

All you need to do is find a directory in which you want to create the virtualenv; your home directory, for example. On Windows, it might look like C:\Users\Name\ (where Name is the name of your login).

First we will be using a new directory djangogirls from your home directory:

**command-line1:**

**$ mkdir djangogirls**

**$ cd djangogirls**

To create a new virtualenv, you need to open the command prompt and run python -m venv myvenv. It will look like this:

**command-line 2:**

**C:\Users\Name\djangogirls> python -m venv myvenv**

## Working with virtualenv

The command above will create a directory called myvenv (or whatever name you chose) that contains our virtual environment (basically a bunch of directory and files).

Start your virtual environment by running:

**command-line 3:**

**C:\Users\Name\djangogirls> myvenv\Scripts\activate**

Now we can finally install Django!

## Installing Django

Now that you have your virtualenv started, you can install Django.

Before we do that, we should make sure we have the latest version of pip, the software that we use to install Django:

**command-line 4:**

**(myvenv) ~$ python -m pip install --upgrade pip**

### Installing packages with requirements

A requirements file keeps a list of dependencies to be installed using pip install:

First create a requirements.txt file inside of the djangogirls/ folder, using the code editor that you installed earlier. You do this by opening a new file in the code editor and then saving it as requirements.txt in the djangogirls/ folder. Your directory will look like this:

**djangogirls**

**├── myvenv**

**│ └── ...**

**└───requirements.txt**

In your djangogirls/requirements.txt file you should add the following text:

djangogirls/requirements.txt

**command line 5:**

**Django~=2.2.4**

Now, run **pip install -r requirements.txt** to install Django.

**command-line 6:**

(myvenv) ~$ **pip install -r requirements.txt**

Collecting Django~=2.2.4 (from -r requirements.txt (line 1))

Downloading Django-2.2.4-py3-none-any.whl (7.1MB)

Installing collected packages: Django

Successfully installed Django-2.2.4

**TO CREATE A SMALL BLOG**

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.

**Create project: Windows**

On Windows you should run the following command. **(Don't forget to add the period (or dot) . at the end)**:

**command-line 7:**

**(myvenv) C:\Users\Name\djangogirls> django-admin.exe startproject mysite .**

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**

**│ ├── \_\_init\_\_.py**

**│ ├── settings.py**

**│ ├── urls.py**

**│ └── wsgi.py**

**├── myvenv**

**│ └── ...**

**└── requirements.txt**

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.

## 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.

We’ll 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')**

## 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:**

**python manage.py migrate**

(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 auth.0008\_alter\_user\_username\_max\_length... OK

Applying auth.0009\_alter\_user\_last\_name\_max\_length... 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**

# Django models

What we want to create now is something that will store all the posts in our blog. But to be able to do that we need to talk a little bit about things called objects.

## Objects

There is a concept in programming called object-oriented programming. The idea is that instead of writing everything as a boring sequence of programming instructions, we can model things and define how they interact with each other.

So what is an object? It is a collection of properties and actions. It sounds weird, but we will give you an example.

If we want to model a cat, we will create an object Cat that has some properties such as color, age, mood (like good, bad, or sleepy ;)), and owner (which could be assigned a Person object – or maybe, in case of a stray cat, this property could be empty).

Then the Cat has some actions: purr, scratch, or feed (in which case, we will give the cat some CatFood, which could be a separate object with properties, like taste).

Cat

--------

color

age

mood

owner

purr()

scratch()

feed(cat\_food)

CatFood

--------

taste

So basically the idea is to describe real things in code with properties (called object properties) and actions (called methods).

How will we model blog posts then? We want to build a blog, right?

We need to answer the question: What is a blog post? What properties should it have?

Well, for sure our blog post needs some text with its content and a title, right? It would be also nice to know who wrote it – so we need an author. Finally, we want to know when the post was created and published.

Post

--------

title

text

author

created\_date

published\_date

What kind of things could be done with a blog post? It would be nice to have some method that publishes the post, right?

So we will need a publish method.

Since we already know what we want to achieve, let's start modeling it in Django!

## Django model

Knowing what an object is, we can create a Django model for our blog post.

A model in Django is a special kind of object – it is saved in the database. A database is a collection of data. This is a place in which you will store information about users, your blog posts, etc. We will be using a SQLite database to store our data. This is the default Django database adapter – it'll be enough for us right now.

You can think of a model in the database as a spreadsheet with columns (fields) and rows (data).

### Creating an application

To keep everything tidy, we will create a separate application inside our project. It is very nice to have everything organized from the very beginning. To create an application we need to run the following command in the console (from djangogirls directory where manage.py file is):

**myvenv) C:\Users\Name\djangogirls> python manage.py startapp blog**

You will notice that a new blog directory is created and it contains a number of files now. The directories and files in our project should look like this:

djangogirls

├── blog

│ ├── admin.py

│ ├── apps.py

│ ├── \_\_init\_\_.py

│ ├── migrations

│ │ └── \_\_init\_\_.py

│ ├── models.py

│ ├── tests.py

│ └── views.py

├── db.sqlite3

├── manage.py

├── mysite

│ ├── \_\_init\_\_.py

│ ├── settings.py

│ ├── urls.py

│ └── wsgi.py

├── myvenv

│ └── ...

└── requirements.txt

After creating an application, we also need to tell Django that it should use it. We do that in the file mysite/settings.py -- open it in your code editor. We need to find INSTALLED\_APPS and add a line containing 'blog.apps.BlogConfig', just above ]. So the final product should look like this:

mysite/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'blog',

]

### Creating a blog post model

In the blog/models.py file we define all objects called Models – this is a place in which we will define our blog post.

Let's open blog/models.py in the code editor, remove everything from it, and write code like this:

**blog/models.py**

from django.conf import settings

from django.db import models

from django.utils import timezone

class Post(models.Model):

author = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

title = models.CharField(max\_length=200)

text = models.TextField()

created\_date = models.DateTimeField(default=timezone.now)

published\_date = models.DateTimeField(blank=True, null=True)

def publish(self):

self.published\_date = timezone.now()

self.save()

def \_\_str\_\_(self):

return self.title

Double-check that you use two underscore characters (\_) on each side of str. This convention is used frequently in Python and sometimes we also call them "dunder" (short for "double-underscore").

It looks scary, right? But don't worry – we will explain what these lines mean!

All lines starting with from or import are lines that add some bits from other files. So instead of copying and pasting the same things in every file, we can include some parts with from ... import ....

class Post(models.Model): – this line defines our model (it is an object).

* class is a special keyword that indicates that we are defining an object.
* Post is the name of our model. We can give it a different name (but we must avoid special characters and whitespace). Always start a class name with an uppercase letter.
* models.Model means that the Post is a Django Model, so Django knows that it should be saved in the database.

Now we define the properties we were talking about: title, text, created\_date, published\_date and author. To do that we need to define the type of each field (Is it text? A number? A date? A relation to another object, like a User?)

* models.CharField – this is how you define text with a limited number of characters.
* models.TextField – this is for long text without a limit. Sounds ideal for blog post content, right?
* models.DateTimeField – this is a date and time.
* models.ForeignKey – this is a link to another model.

We will not explain every bit of code here since it would take too much time. You should take a look at Django's documentation if you want to know more about Model fields and how to define things other than those described above (<https://docs.djangoproject.com/en/2.2/ref/models/fields/#field-types>).

What about def publish(self):? This is exactly the publish method we were talking about before. def means that this is a function/method and publish is the name of the method. You can change the name of the method if you want. The naming rule is that we use lowercase and underscores instead of spaces. For example, a method that calculates average price could be called calculate\_average\_price.

Methods often return something. There is an example of that in the \_\_str\_\_ method. In this scenario, when we call \_\_str\_\_() we will get a text (**string**) with a Post title.

Also notice that both def publish(self): and def \_\_str\_\_(self): are indented inside our class. Because Python is sensitive to whitespace, we need to indent our methods inside the class. Otherwise, the methods won't belong to the class, and you can get some unexpected behavior.

If something is still not clear about models, feel free to ask your coach! We know it is complicated, especially when you learn what objects and functions are at the same time. But hopefully it looks slightly less magic for you now!

### Create tables for models in your database

The last step here is to add our new model to our database. First we have to make Django know that we have some changes in our model. (We have just created it!) Go to your console window and type python manage.py makemigrations blog. It will look like this:

**command-line**

(myvenv) ~/djangogirls$ python manage.py makemigrations blog

Migrations for 'blog':

blog/migrations/0001\_initial.py:

- Create model Post

**Note:** Remember to save the files you edit. Otherwise, your computer will execute the previous version which might give you unexpected error messages.

Django prepared a migration file for us that we now have to apply to our database. Type python manage.py migrate blog and the output should be as follows:

**command-line**

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

Operations to perform:

Apply all migrations: blog

Running migrations:

Applying blog.0001\_initial... OK

# **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 in the code editor and replace its contents with this:

**blog/admin.py**

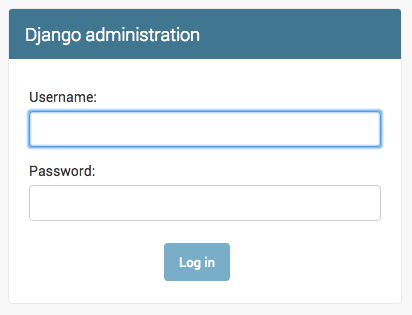
from django.contrib import admin

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.

**(myvenv) C:\Users\Name\djangogirls> python manage.py createsuperuser**

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.** Type it in and press enter to continue. The output should look like this (where the username and email should be your own ones):

Username: ola

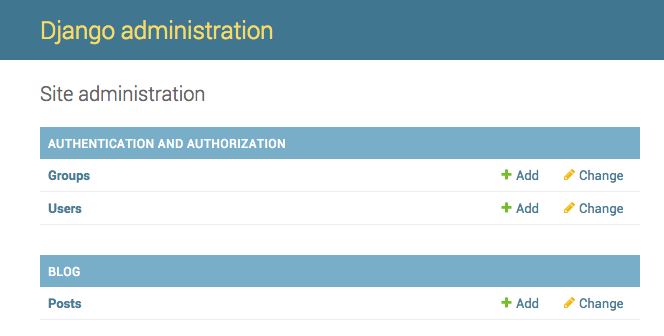
Email address: ola@example.com

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 –- it's only visible to you on your local computer -- you can copy-paste some text from this tutorial to save time. :)

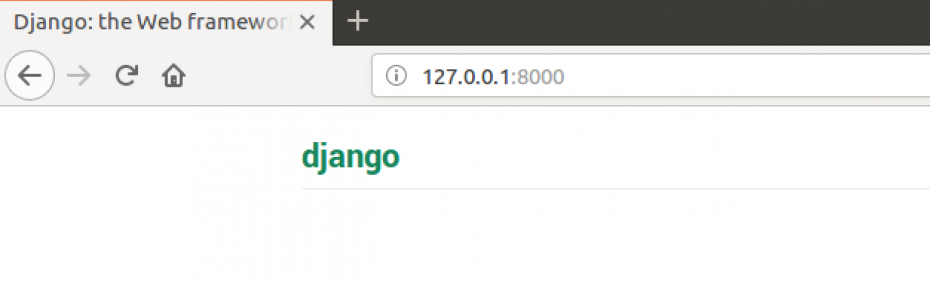
Make sure that at least two or three posts (but not all) have the publish date set. It will be helpful later.

# 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 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 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.contrib import admin

from django.urls import path

urlpatterns = [

path('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 the previous chapter, is already here:

**mysite/urls.py**

**path('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.

## 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. You will also need to change the from django.urls… line because we are using the include function here, so you will need to add that import to the line.

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

**mysite/urls.py**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('', include('blog.urls')),

]

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

## blog.urls

Create a new empty file named urls.py in the blog directory, and open it in the code editor. All right! Add these first two lines:

**blog/urls.py**

from django.urls import path

from . import views

Here we're importing Django's function path 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 = [

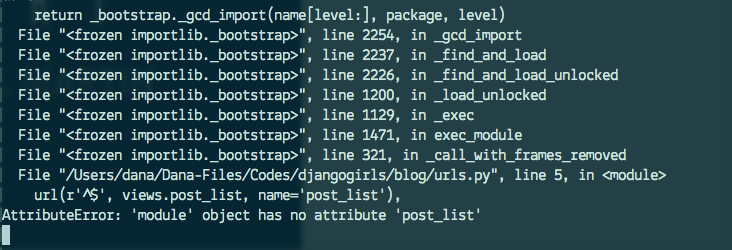
path('', views.post\_list, name='post\_list'),

]

As you can see, we're now assigning a view called post\_list to the root URL. This URL pattern will match an empty string and the Django URL resolver will ignore the domain name (i.e., <http://127.0.0.1:8000/>) that prefixes the full URL path. 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 see a different error message, try restarting your web server. To do that, in the console window that is running the web server, stop it by pressing Ctrl+C (the Control and C keys together). On Windows, you might have to press Ctrl+Break. Then you need to restart the web server by running a python manage.py runserver command.

# **Django views – time to create!**

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 in our code editor 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.

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 will return the value it gets from calling another 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 .

# **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.

## What is HTML?

HTML is a 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

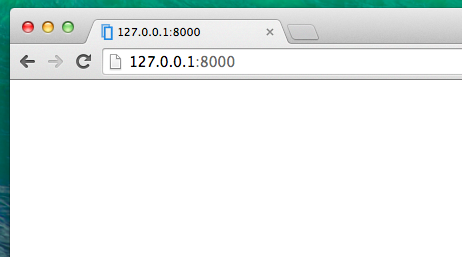
└───blog

(You might wonder why we need two directories both called blog – as you will discover later, this is 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 to the 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.

Open the new file in the code editor, and add the following:

blog/templates/blog/post\_list.html

<html>

<body>

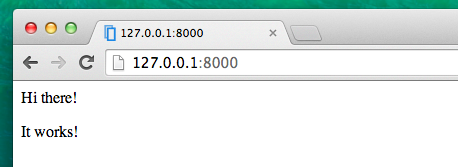
<p>Hi there!</p>

<p>It works!</p>

</body>

</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>

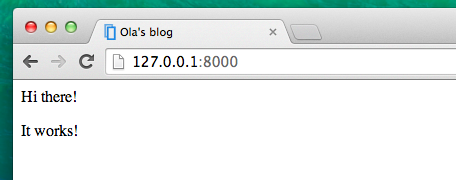
<p>Hi there!</p>

<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 and there's no closing tag)
* <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 blog posts 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 blog post.

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**.

# **Django ORM and QuerySets**

In this chapter you'll learn how Django connects to the database and stores data in it. Let's dive in!

## What is a QuerySet?

A QuerySet is, in essence, a list of objects of a given Model. QuerySets allow you to read the data from the database, filter it and order it.

It's easiest to learn by example. Let's try this, shall we?

## Django shell

Open up your local console (not on PythonAnywhere) and type this command:

**command-line**

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

The effect should be like this:

**command-line**

(InteractiveConsole)

>>>

You're now in Django's interactive console. It's just like the Python prompt, but with some additional Django magic. :) You can use all the Python commands here too.

### All objects

Let's try to display all of our posts first. You can do that with the following command:

command-line

>>> Post.objects.all()

Traceback (most recent call last):

File "<console>", line 1, in <module>

NameError: name 'Post' is not defined

Oops! An error showed up. It tells us that there is no Post. It's correct – we forgot to import it first!

**command-line**

>>> from blog.models import Post

We import the model Post from **blog.models**. Let's try displaying all posts again:

**command-line**

>>> Post.objects.all()

<QuerySet [<Post: my post title>, <Post: another post title>]>

This is a list of the posts we created earlier! We created these posts using the Django admin interface. But now we want to create new posts using Python, so how do we do that?

### **Create object**

This is how you create a new Post object in database:

**command-line**

>>> Post.objects.create(author=me, title='Sample title', text='Test')

But we have one missing ingredient here: me. We need to pass an instance of User model as an author. How do we do that?

Let's import User model first:

**command-line**

>>> from django.contrib.auth.models import User

What users do we have in our database? Try this:

**command-line**

>>> User.objects.all()

<QuerySet [<User: ola>]>

This is the superuser we created earlier! Let's get an instance of the user now (adjust this line to use your own username):

**command-line**

>>> me = User.objects.get(username='ola')

As you can see, we now get a User with a username that equals 'ola'. Neat!

Now we can finally create our post:

**command-line**

>>> Post.objects.create(author=me, title='Sample title', text='Test')

<Post: Sample title>

Hurray! Wanna check if it worked?

**command-line**

>>> Post.objects.all()

<QuerySet [<Post: my post title>, <Post: another post title>, <Post: Sample title>]>

There it is, one more post in the list!

### **Add more posts**

You can now have a little fun and add more posts to see how it works. Add two or three more and then go ahead to the next part.

### **Filter objects**

A big part of QuerySets is the ability to filter them. Let's say we want to find all posts that user ola authored. We will use filter instead of all in Post.objects.all(). In parentheses we state what condition(s) a blog post needs to meet to end up in our queryset. In our case, the condition is that author should be equal to me. The way to write it in Django is author=me. Now our piece of code looks like this:

**command-line**

>>> Post.objects.filter(author=me)

<QuerySet [<Post: Sample title>, <Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>]>

Or maybe we want to see all the posts that contain the word 'title' in the title field?

**command-line**

>>> Post.objects.filter(title\_\_contains='title')

<QuerySet [<Post: Sample title>, <Post: 4th title of post>]>

There are two underscore characters (\_) between title and contains. Django's ORM uses this rule to separate field names ("title") and operations or filters ("contains"). If you use only one underscore, you'll get an error like "FieldError: Cannot resolve keyword title\_contains".

You can also get a list of all published posts. We do this by filtering all the posts that have published\_date set in the past:

**command-line**

>>> from django.utils import timezone

>>> Post.objects.filter(published\_date\_\_lte=timezone.now())

<QuerySet []>

Unfortunately, the post we added from the Python console is not published yet. But we can change that! First get an instance of a post we want to publish:

**command-line**

>>> post = Post.objects.get(title="Sample title")

And then publish it with our publish method:

**command-line**

>>> post.publish()

Now try to get list of published posts again (press the up arrow key three times and hit enter):

**command-line**

>>> Post.objects.filter(published\_date\_\_lte=timezone.now())

<QuerySet [<Post: Sample title>]>

### Ordering objects

QuerySets also allow you to order the list of objects. Let's try to order them by created\_date field:

**command-line**

>>> Post.objects.order\_by('created\_date')

<QuerySet [<Post: Sample title>, <Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>]>

We can also reverse the ordering by adding - at the beginning:

**command-line**

>>> Post.objects.order\_by('-created\_date')

<QuerySet [<Post: 4th title of post>, <Post: My 3rd post!>, <Post: Post number 2>, <Post: Sample title>]>

### Complex queries through method-chaining

As you saw, some methods on Post.objects return a QuerySet. The same methods can in turn also be called on a QuerySet, and will then return a new QuerySet. Thus, you can combine their effect by **chaining** them together:

>>> Post.objects.filter(published\_date\_\_lte=timezone.now()).order\_by('published\_date')

<QuerySet [<Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>, <Post: Sample title>]>

This is really powerful and lets you write quite complex queries.

Cool! You're now ready for the next part! To close the shell, type this:

**command-line**

>>> exit()

# **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 in our code editor. 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')

So, let's open the blog/views.py file in the code editor, and add this piece of code 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', {})

To display our QuerySet on our blog's post list, we have two things left to do:

1. Pass the posts QuerySet to the template context, by changing the render function call. We'll do this now.
2. Modify the template to display the posts QuerySet. We'll cover this 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})

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

# **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. Cool!

## Display post list template

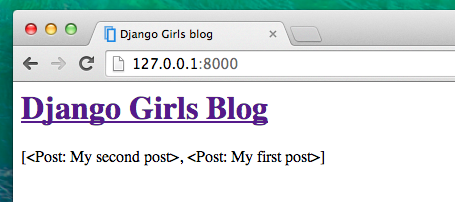
In the previous chapter we gave our template a list of posts in the posts variable. Now we will display it in HTML.

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. Open it up in the code editor, and 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>

<h2><a href="">{{ post.title }}</a></h2>

<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.

# **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, open up your .html file in the code editor and add this to the <head> section:

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. So 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 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

│ └── templates

└── 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. There is a recommendation for a free CSS course at the end of this page if you would like to learn more.

But let's do at least a little. Maybe we could change the color of our headers? To understand colors, computers use special codes. These codes start with # followed by 6 letters (A–F) and numbers (0–9). For example, the code for blue is #0000FF. You can find the color codes for many colors here: <http://www.colorpicker.com/>. You may also use [predefined colors](http://www.w3schools.com/colors/colors_names.asp), such as red and green.

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

blog/static/css/blog.css

h1 a, h2 a {

color: #FF8833;

}

h1 a is a CSS Selector. This means we're applying our styles to any a element inside of an h1 element; the h2 a selector does the same thing for h2 elements. 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 #C25100, which is a dark orange. Or you can put your own color here, but make sure it has good contrast against a white background!

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">

You can read more about [CSS Selectors at w3schools](http://www.w3schools.com/cssref/css_selectors.asp).

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

blog/templates/blog/post\_list.html

{% load static %}

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 be overriden by 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 static %}

<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>

<h2><a href="">{{ post.title }}</a></h2>

<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, h2 a {

color: #C25100;

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>

<h2><a href="">{{ post.title }}</a></h2>

<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, copy and paste it into your blog/static/css/blog.css file:

blog/static/css/blog.css

.page-header {

background-color: #C25100;

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 h2 a, .post h2 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>

<h2><a href="">{{ post.title }}</a></h2>

<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>

<h2><a href="">{{ post.title }}</a></h2>

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

</div>

{% endfor %}

</div>

</div>

</div>

Save those files and refresh your website.

