

## Django



### A little History

So let's overview!

Django: A "web application framework"

- Released publicly in 2005
- basically a framework abstracted from web application for a newspaper in Kansas.
- immediate popularity and lots of development activity
- 1.0 release in 2008 (essentially modern Django)
- 1.5 released earlier this year

Był fajny i od razu kliknęło, it hit exactly right, RoR was popular at that time and python was the answer to RoR.



### A little History

Same space as competitors like Ruby on Rails, Pyramid etc.

Successfully used on large projects:

- Disqus
- Pinterest
- The Onion
- various parts of mozilla.org
- etc

50K+ downloads of Django version 1.5 via PyPi

4,200+ sites listed on <http://www.djangosites.org/>

Just the general space is to write, quickly write web applications that are backed up by a database. It is widely used, beloved in python community.



Screw you Django unchained!

## So Why Django?

- Documentation
- Python
- full-stack framework (lots of batteries built-in)
- Hits the sweet spot for database backed web applications.

Django pays lots of attention to documentation, have great docs - first creators/users (in a newspaper) had English degrees instead of CS degrees.

It uses Python.

Django does (almost) everything.

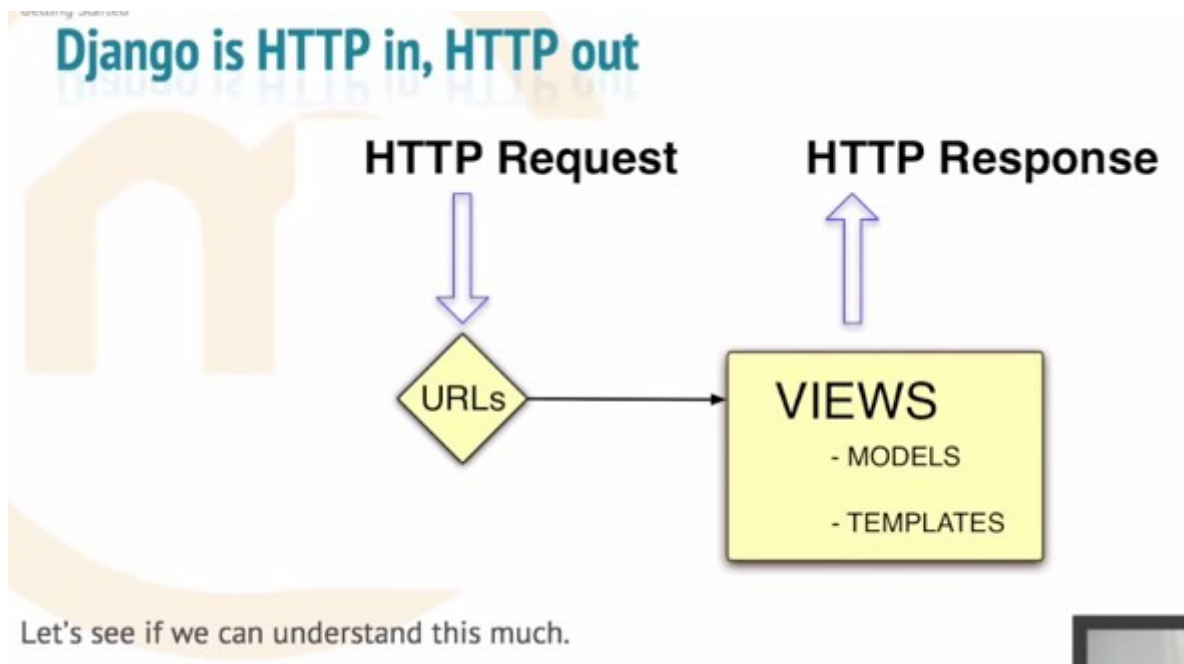
Makes a lot of things easier, faster for database backed web applications (80% something?)

## Getting Started

- Reddit and Pinterest are very different projects.
- But both are at least partly:
 

HTML interfaces to highly dynamic data stored in a database.
- What do we have to know?

Pinterest and Reddit.  
Lots of data floating around, dynamic data.



Django is going to be HTTP in and HTTP out – you gonna need to understand that.  
The diagram has HTTP Requests and HTTP responses and Django is the stuff in yellow.

What do you have to know to write the next Pinterest (well it's gonna be more than this)?  
The minimum you have to know to write a web application, you need to know about:

urls – the request that's coming in, where should it go, and you pick sth,  
views – python callables that actually return the response and  
If you have some really basic web application your view might only use models to tap it to database and templates to produce some html.

10 minnutes



# Introduction - Django Web Development with Python 1 - 19.01.2016

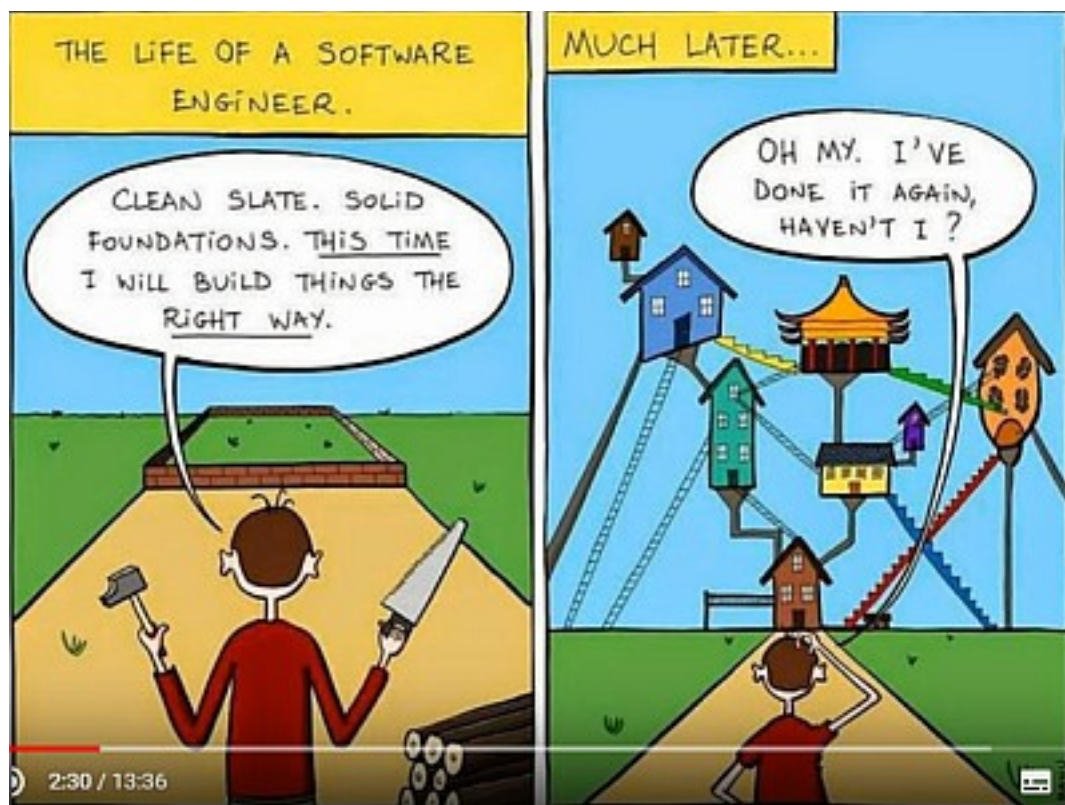
Django a web framework for Python. All the framework is is just a foundation, a set of tools for you to build on top in this is case to make a website.

Firts thing people are going to ask when looking at web development with Python when there are like 20 frameworks for Python, which one do I choose?

For the most part you've got high level and low level frameworks, Django is the highest level framework and the rest are pretty low level frameworks.

We could compare Django to something like Flask. The main difference between these two is Flask allows you a lot of freedom to create things your way, to do them in a custom manner exactly however you want. Django on the other hand forces you to do things the Django way. The trade-off here is under Flask you can do things your way, but your way is almost certainly not the best way and with Django the Django way is almost certainly the best way and if you make an intire website are always doing the Django way overall you're going to have one out most likely, because, honestly, if you're honest with yourself, you're bad at creating systems. Even the extremely talented girls and guys couldn't get it right the first time. It has taken thousand of developers and millions of end users using the Django framework to get it where it is today and it is still under active development.

So using something like Django is very tested, it's going to scale, it's going to be secure and efficient. By scale I mean two things: a webiste scaling that has to do with some back-end things a little bit more, but by scale I usally mean, you know most people refer to scale as how many users can it hold, like can it go from 5000 to 5 mln, but actually when a project scales in my mind where you add abuch of new things and features and stuff like that what ends up happening is this:



One of favourite pictures

You are starting to build a house, everything goes well, then you start adding a little bit of new things and before you even know, you've made a huge mess.

With Django it starts you out right at the gate with something like this:



where you got your main website/your project and you start to build out into what are called apps. The premise with Django is that every website is really just a combination of certain apps, so in this case one might be a forum or a blog or a store, all contained on the same website.

The terminology might get very confusing really fast, especially with something like Django, which is very high-level, very abstracted and it can just be too much sometimes and a little bit overwhelming.

I don't want to focus too much on terminology, but basically you might hear that a lot of times people consider a web app to be a web page or, let's say, phone app to be like the whole thing not to have sub apps, but with Django basically your whole website has these little sub apps, in each app actually could be a standalone website, you might have just a store, a blog or it might be a forum or something like that. **Diagram?**

At the basic level that's what it is. So the beauty of using Django is that it can grow in a way that requires foresight that you probably just don't have when you're initially setting out for a project, but with Django they basically give you that foresight using the certain paradigm, that they kind of forcing on you, but it's probably a good choice.

## GETTING DJANGO

>pip install Django -> you get the latest Django version whatever that happens to be  
>pip install Django==1.9 -> you get the specific Django version

Note: All of the code, all of the Django, the backend, the development server, everything will be the same, no matter what your operating system is, Django is blind to the operating system basically.

<https://www.djangoproject.com/>

## CREATING A PROJECT

Create a new folder, eg Django Project <where django project is located>  
Open terminal in windows, [hold shift + right mouse click and Open Command Window Here]  
> **django-admin startproject <project name>** - name project whatever you want, eg mysite

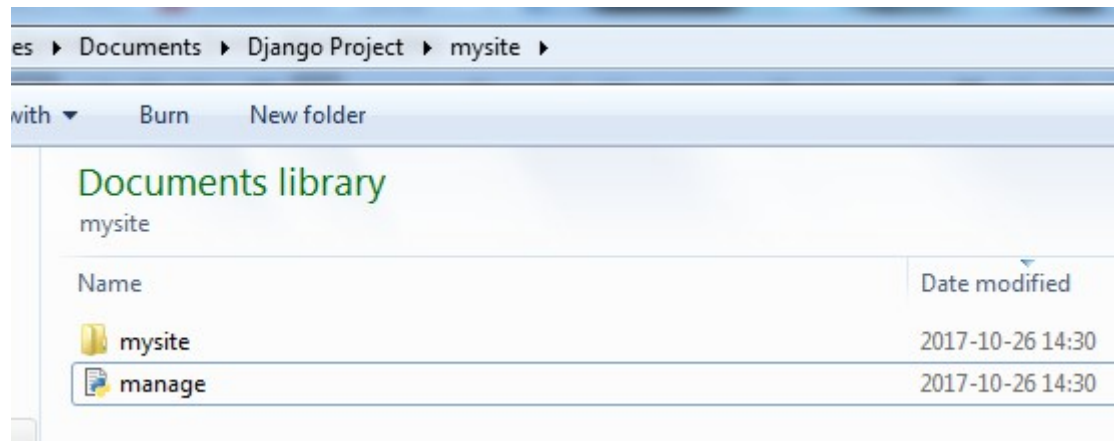
mysite contains another directory mysite. First mysite (higher in hierarchy) is just a container, doesn't matter how you call this. Second mysite folder (lower hierarchy) needs to stay the same, because it's in your settings now. This is going to be the main kind of hub of my website so this



mysite directory is equivalent of that website/project black text there:

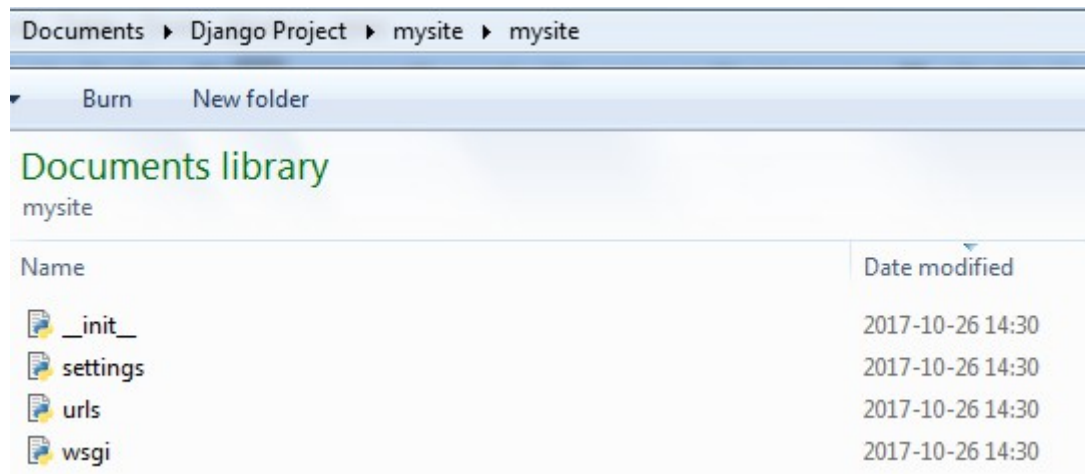


mysite folder (upper):



managy.py some simple stuff here - this is your command line tool, we'll be using manage.py quite a bit, you will actually be reading manage.py very often (?):

mysite directory (lower):



**\_\_init\_\_.py** - just tells Python this is to be treated like a package. We don't touch this.

Tidbit: he put the entire website in the init.py file during the Flask tutorial. That's not going to fly with Django, you gotta do with the right way, ie the Django way.

**settings.py** - settings for your website, **the main hub**. Installed apps are one of the more important things, everytime you add an application, you have to manually add it here. If you download someone else's application, you have to add (install) it in here.

Probably the most important thing here is the **SECRET\_KEY** line. If you put your website up on Github, do not put this here, this is used for encryption, basically where things like your sessions, stuff like that. If someone has access to this key, they can decrypt the session, claim themselves as an admin, re-encrypt and they will be treated like an admin on your website, so don't do that.

**urls.py** - this is the main kind of controller of your website.

Right now we have one url pattern. The url patterns are regular expressions.

^ caret – the beginning of a string

\$ dollar sign – the end of a string

So basically this url pattern just basically says (and?) all this url patterns start after the initial domain name.

Tip: If that all sounds like gibberish to you, don't worry, there's a lot to swallow with Django and just take it very slow and ask questions (below the videos or pythonprograming.net community, stack overflow, reddit).

Basically this file will just point to your apps, so this file is just underneath main website/project and it points to the various apps and as time goes on, you'll see that the apps themselves have their own urls.py files, so that can get kind of confusing, at least that was confusing to me.



## RUNNING (ACTIVATING) A SERVER

> [path to manage.py] **python manage.py runserver**

eg >[C:\Django](#) Project\mysite>python manage.py runserver

If we have more Python versions then we can/should explicitly type a path:

eg >[C:\Django](#) Project\mysite>C:/Python35/python manage.py runserver

Feel free to run it.

You absolutely will see you have unapplied migrations. Don't worry about that migrations, basically everytime you add new models, I suppose, you will need to do make migrations and migrate.

You will notice it actually tells you exactly what you need to do to fix this little error, it says: **"Run 'python manage.py migrate' to apply them."**

[U mnie na screenie tego nie widać bo wcześniej testowo uruchomiłem migrate].

**So the errors and stuff you are going to see here are super helpful errors.**

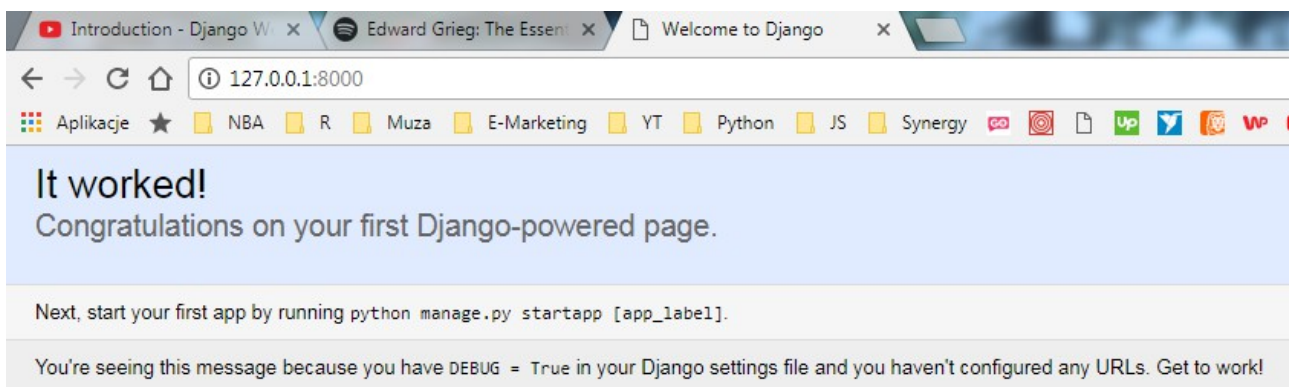


```
Administrator: C:\Windows\system32\cmd.exe - python manage.py runserver

C:\Users\Mikołaj\Documents\Django Project\mysite>python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
October 30, 2017 - 09:46:41
Django version 1.11.6, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[30/Oct/2017 09:46:57] "GET / HTTP/1.1" 200 1716
[30/Oct/2017 09:50:46] "GET / HTTP/1.1" 200 1716
```

Ok, so go head over to your browser and go to the following url:  
127.0.0.1:8000



Basically 127.0.0.1 is just localhost, so that is your local IP, so this is obviously not accessible outside of your home network and actually. Well, if you put computer's local IP, as far as I know, the local IP will correspond, but don't worry about that. Anyway basis (?) is on port :8000.

```
Administrator: C:\Windows\system32\cmd.exe - python manage.py runserver

C:\Users\Mikołaj\Documents\Django Project\mysite>python
Performing system checks...

System check identified no issues (0 silenced).
October 30, 2017 - 09:46:41
Django version 1.11.6, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[30/Oct/2017 09:46:57] "GET / HTTP/1.1" 200 1716
[30/Oct/2017 09:50:46] "GET / HTTP/1.1" 200 1716
```

So this is your development server, so Django comes shipped with this awesome development server. Basically over here you're running the server and you can see all the requests. So you can actually see here we made a request to hear the responses at 200 which is successful and then we did not get a favicon so that was returning a 404 for us, don't worry about that (u mnie tego błędu nie ma, sentdex nie jest pewien dlaczego u niego takie coś wyskoczyło, maybe there's like a cache or something).

Read the errors, see what it says, they are very useful errors, otherwise it should look just like this.

```
Next, start your first app by running python manage.py startapp [app_label].
```

```
You're seeing this message because you have DEBUG = True in your Django settings file and you haven't configured any URLs. Get to work!
```

# Creating App - Django Web Development with Python 2 - 20.01.2016

In this tutorial we will be talking about how to add another app basically, because our website right now doesn't really have anything. We have got our kind of main hub going on, but we don't have any apps and as we said before a website is a combination of apps, so we need at least one app.

Tip: To break running server click Ctrl+C (Ctrl+Break also works for me)

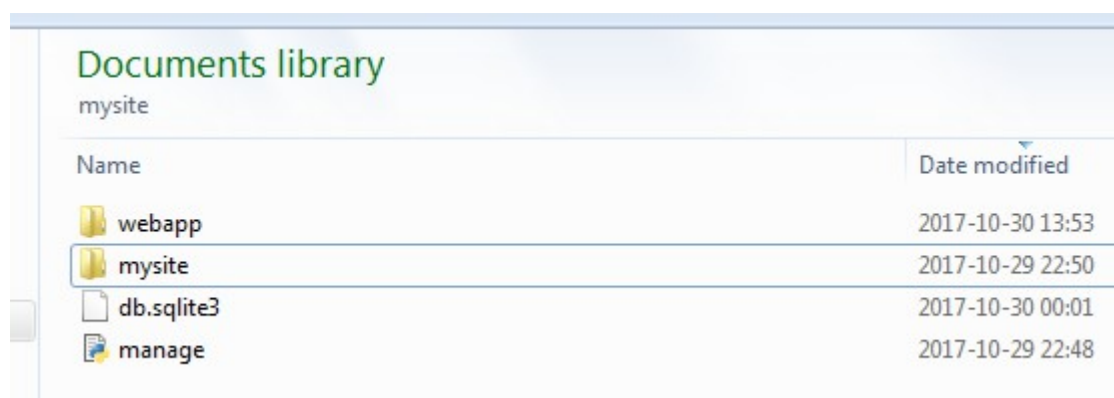
## STARTING A NEW APP

> [path to manage.py] python manage.py startapp [webapp name]

eg >C:\Django Project\mysite>python manage.py startapp webapp

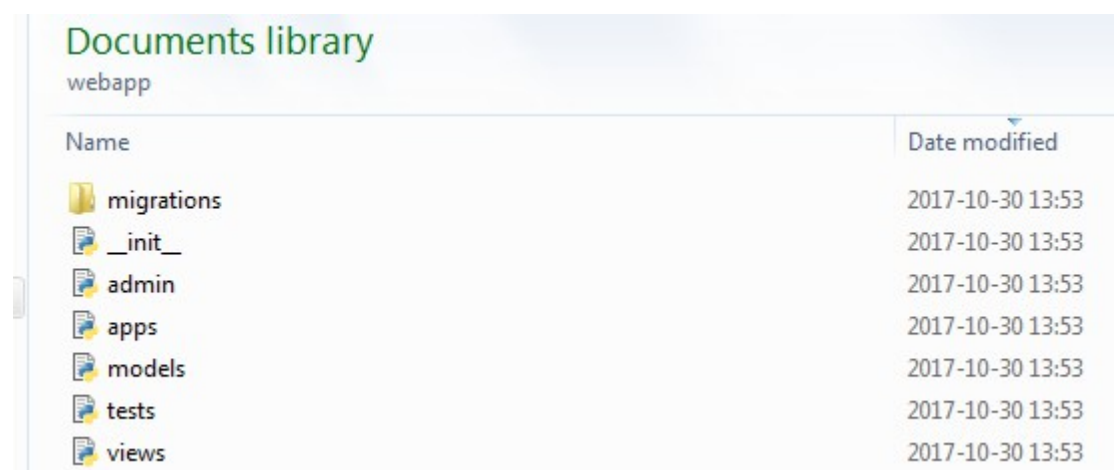
You can call any name that you want. The idea here is in any interest of keeping things as simple as possible yet also useful, the first thing we are going to do is just build like a personal website and it will have some about me page and then like a blog that you can post on, something like that. And then from there we can go on to make a much more complex website possibly, but for now we'll keep it pretty simple, yet actually end up with something somewhat useful hopefully for you in the end.

mysite/mysite is a central hub



Name	Date modified
webapp	2017-10-30 13:53
mysite	2017-10-29 22:50
db.sqlite3	2017-10-30 00:01
manage	2017-10-29 22:48

Let's go inside the webapp folder:



Name	Date modified
migrations	2017-10-30 13:53
__init__	2017-10-30 13:53
admin	2017-10-30 13:53
apps	2017-10-30 13:53
models	2017-10-30 13:53
tests	2017-10-30 13:53
views	2017-10-30 13:53

So we've got a few extra things that we didn't quite have in that original mysite.

`__init__.py` – same reason, it's treated as its own app

`admin.py` – administration stuff, we will leave that for now

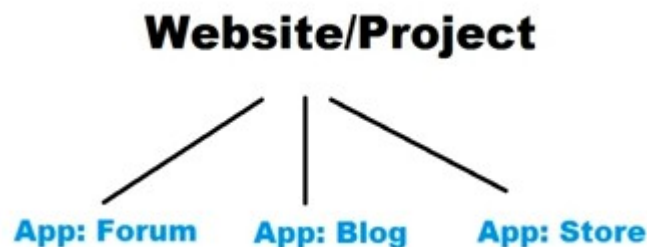
`apps.py` – right now and probably in the nearest future this will never be edited

`models.py` – this is what contains database information as well as just some simple metadata

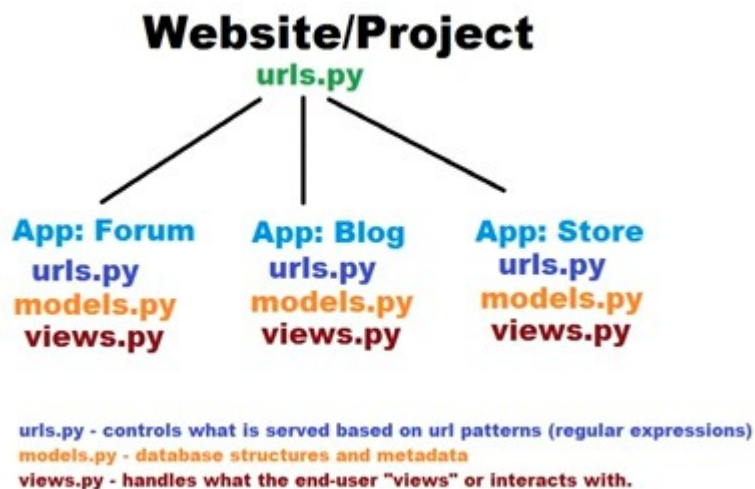
`tests.py` - **historyjka**

`views.py` – this is basically what the user sees, Django is a Model View Controller, you've got your models, your views and `urls.py` is your controller.

Let me pull up a picture here, our previous picture was just this:

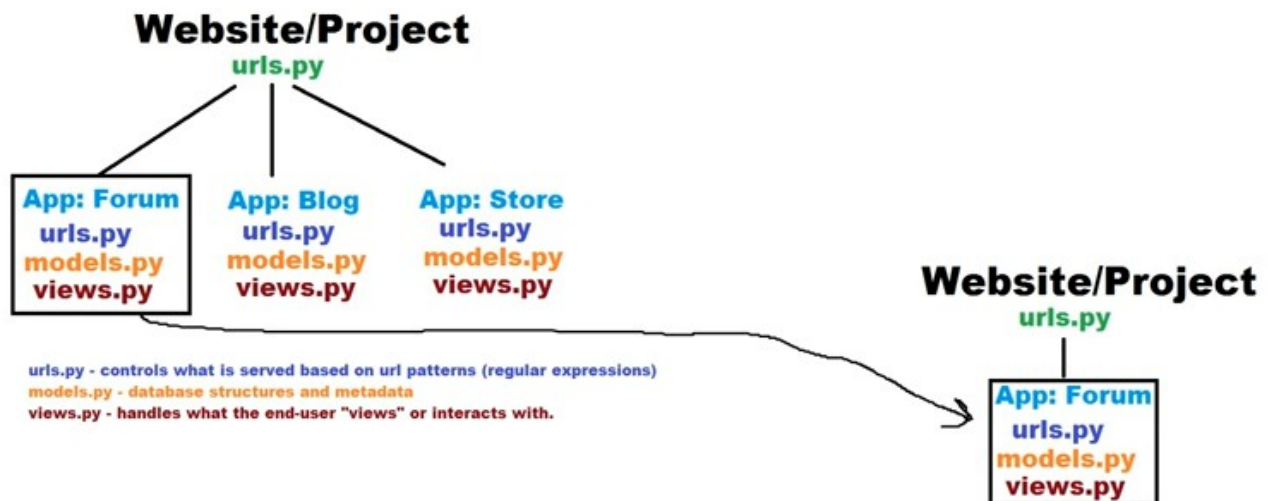


We can now kind of populate that a bit more and you'll see why in this tutorial:



So the main website has its own `urls.py`, you've seen it, but it currently doesn't really do much. So you have got your `urls.py` which controls and it links to all of the apps in your project and each app will have, at least, doesn't currently have a `urls.py` by default, but it will have it. So the you've got your models, views, so every app will consist of that and just below it you can see `urls.py` just controls what's served, models is your database structure and your metadata, and then views is basically what the end-user winds up seeing.

The reason why we do things this way, rather than, say, the flask way, at least at the basic, you could do a Model View Controller paradigm with Flask. Nothing stops you from doing it, you just have to do it yourself, okay? But what Django does is it just has it built in place for you. So the reason for that is so you can do something like this:



You have got your website and it has an app of forum, but maybe you want another website that is just a forum or something, you can literally take that exact app and move it right on over to your other website and on that same kind of thread you can download other people's apps and use them really quick, really easy. So that's why this Django paradigm just makes a whole lot of sense to use.

## APP INSTALLATION

So moving along we need to go ahead and get started making our own, basically connecting this app, so everytime you start an app, what do we do from there. Well, first of all if we come over here to our settings, the first thing that you might want to do is just add web app. There are a couple things that you can do here, the easiest thing that you can say is just "webapp", just add it that way. Just add "webapp" comma, and you are good to go.

mysite/mysite/settings.py

```
# Application definition

INSTALLED_APPS = [
    'webapp',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

These other things are just defaults, you could remove some of these I suppose, but you might as well leave them. These are things that you are just almost certain to use. And there are other apps that come with Django that are not just default installed, so there's a lot of stuff that comes with Django. In fact, the unofficial mascot of Django is a pony.



Eventually just became a saying: "No, you can't have a pony".

In Django's defence, they have a lot of features, they have pretty much honored quite a bit of those feature requests, so that's why we have such an impressive high level framework that we do.

Anyways, we've installed webapp and now what we want to do, while we are in that mysite directory, we got to go to the urls.py, so this is that main controller, so it leads to everything. As you can see currently the only thing it leads to is the starting of admin and there is no end.

```
"""
from django.conf.urls import url
from django.contrib import admin

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

So remember how I said that the url patterns generally, at least at the most basic level, are going to consist with a caret and a dollar sign if you want to end it. There is no dollar here, so there is no necessary ending to this url. All we know for sure it's going to point to admin.site.urls. You can kind of think of admin.site as it's own app, because that's exactly what it is.

This url's just saying 'hey, if the pattern begins with 'admin/', but doesn't end with anything yet, it could end, but it doesn't have to end here, let's check out the admin.site.urls.py file and see what it says as far as where we go from there. So now we are going to add our own:

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

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

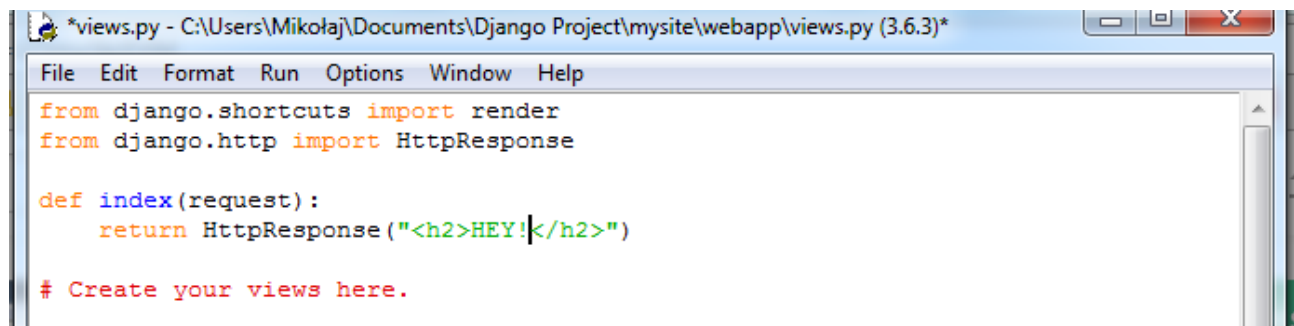
It's a url, it's not to admin. We include the following file and it'll be that urls.py that doesn't exist yet, but we're about to make it. So what it will do is when someone goes to your website/webapp, it's then going to say 'ok, cool, now that we are here, we need to consult webapp.urls to figure out

what view we need to serve up for the user'. Now, we're using this keyword 'include' which we don't actually have yet, so we need to import this 'include'. That's all here.

Now we need to go into that new app, that web app and the first thing we want to do is we are going to **serve a view** here. And in here we have a few things. We actually have render, but we're not going to use render here, we're going to use a simple response, so just for now we are going to say:

```
from django.shortcuts import render
from django.http import HttpResponse
```

and then we're going to define 'index' and then **the parameter** here is 'request' and what we want to return here is a simple HTTP response and here you can just write in HTML, this is not how you would normally do it, but it's just what we are going to do just to make it very short, it'll be just a big 'HEY' in the heading two tags.



```
*views.py - C:\Users\Mikołaj\Documents\Django Project\mysite\webapp\views.py (3.6.3)*
File Edit Format Run Options Window Help
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h2>HEY!</h2>")

# Create your views here.
```

So that's our view for this index request, so that's the view, now how do we control that view to show? Well, we use urls.py for this. So that's our views and now we need a urls.py. We don't actually have one, so I'm just going to just take view.py, copy/paste and call it 'urls', and edit.

What we are going to say here for this is going to be:

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

that's how we specify url pattern and then from period '.' import views. Period is a relative import, basically we're just importing from a current package. Ok, so that's how we can keep things very dynamic, so we're just importing views locally **fdkfhkds [unknown word]**, so they're relatively not locally [niejasne explanation].

Url pattern is going to be a list in our case:

```
urlpatterns = [] -> s na końcu!!!!
```

and then the url pattern that we want to reference is just going to be:

```
url(r'^$', views.index, name='index')
```

the url will be, it's a regular expression (r) and it starts (^) and ends (\$). So this would be basically an 'index', there's nothing here, it's just start&end and that's that. If that is the url pattern it's going to return the views.index, so that is here: views is the views.py file (we've just created), index is this function (inside the file). What is it (function) doing? It's returning this response (hey in html). And then we will give a name to views.index, for now it needs a namespace, so name is 'index' ok?



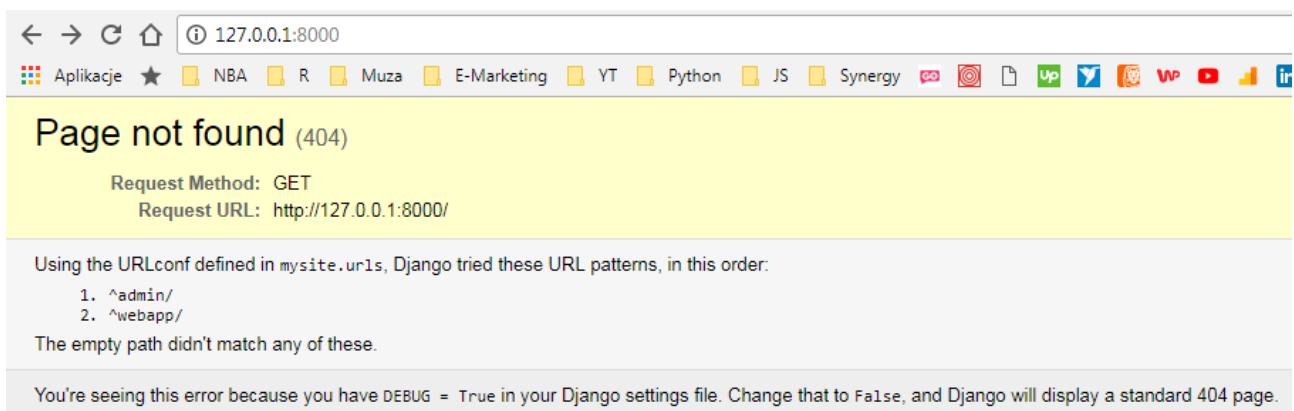
```
urls.py - C:\Users\Mikołaj\Documents\Django Project\mysite\webapp\urls.py (3.6.3)
File Edit Format Run Options Window Help
from django.conf.urls import url
from . import views

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

```

So once we've done that, we should be all set. I might be forgetting something and if I am forgetting something, we'll get an error and then we can explain why those errors are so useful.

So we'll bring back over the terminal, the server is not running, so run server. We can have some unapplied migrations, but probably it's ok.

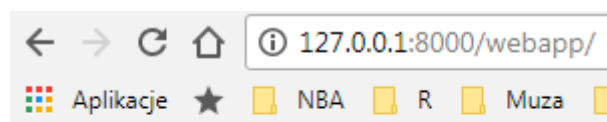


So we visited the homepage, that basically is the index of the website and it says 404 page not found. Then it goes through and tells you exactly what it looked for. It looked for the following things to be in the url: either looked for the starting of the url to be 'admin/' or the starting of the url to be 'webapp/', which if we go into mysite/mysite/ and open that urls.py file:

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

again this is that main hub, that's exactly what it did right, it went through these url patterns and said 'hey does this url have admin/? nope, does it have webapp/? nope, so it returns a 404.

Now what if we say okay: <http://127.0.0.1:8000/webapp/> (instead of <http://127.0.0.1:8000/>), sure enough we get HEEEEEEYY!! (sorry headphone users):



**HEY!**

We get hey and we actually find out that indeed we actually did not get any errors, so we actually did it right the first time through (yay!) and we got that message.

So what happened in this case was it did the url began with webapp/, so it said 'ok webapp/' when it

saw that it says 'ok we need to include webapp.urls', so it said 'all right that's fine Danny', so it went back here and went to webapp, it went to urls, opened that up, read it and it's said 'ok what's going on here, well this is just it begins and it ends', so what happened it's returning this views.index with the name of 'index', so then it went here to views.index and it return the following view which was this and just HTTP response with 'hey' (ok, you're welcome, I didn't yell this time).

SO, that's just a really basic example of, you know, starting an application and connecting that application, because again, **that's how the Django app actually works, it just connects these applications based on the urls** and initially when I started out with Django I was somewhat confused, because to me it just doesn't really make that much sense to have multiple urls.py and what makes it nice though, is you could take now..., let's say your web app is a really epic application, you could take the web app and include it in another website and web app can be like, for example, like let's say you included another application so the word on mysite/urls.py and instead of it being corresponding to 'webapp/', you want it to correspond to a 'ggg/' or something like this:

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

You could save that and then what you could do is you could come over to your new app and now we can refresh and actually let's just go 'ggg' (in the browser url address) and you'll see that it corresponds there:



So you can include this in any of your other packages, you just have to click and drag it in, right? And if it has models, you have to migrate, but we are not going to worry about that right now, we'll talk about that very soon, but if it has database needs, you have to migrate, but if it doesn't all you have to do is basically you just bring it in and then you have to come over to your main mysite, your settings and install the application, but let's be serious, it's not really installing, it's just, you know, you type in the word and boom, it's done.

Hopefully that was a bite-sized chunk of Django, if you have questions, comments, concerns, whatever, feel free to leave them below, I'll be happy to help you out where I can, otherwise stay tuned for the next tutorial and thanks for watching.

# Jinja Templating - Django Web Development with Python 3 -

## 21.01.2016

In this tutorial we're gonna be talking about templating. The idea of templating is pretty simple to understand. The concept behind it is consider you've got a website, most websites look very similar page to page, this is because they use, what we call in HTML circles, headers and footers, okay? And so the idea there is that you've got nav bars let's say and the nav bar is something you include in your header file and this way on every page the navigation bar is identical. Then when it comes time to change the navigation bar, you just have to edit one file rather than editing a hundred files or even more, right? So we use templates to do that and in for Python and in with major Python frameworks we use Jinja.

So Jinja is a templating framework built with Python and working with Python in mind, it is not Python in HTML. There are a lot of things in Python that you just can't do with Jinja, but Jinja can do a lot of logic that at least get us pretty far with writing dynamic and logical ish HTML and there are actually some things in Jinja that you can do that you can't do in Python just based on the fact that we are writing in HTML.

So anyways, let's go ahead get started. The first thing we are going to do is, this project, the idea here is to create a personal website, about you website, so we're going to start a new app, because 'webapp' doesn't really fit that bill.

1. Starting a new 'personal' app
2. Installing the 'personal' app
3. Link that sends people to the 'personal' app (mysite/urls.py):

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

In most cases you'll be able to just fall, ..., in most cases you'll probably follow this structure that I'm about to show you. So I will take one of the previous lines that's including another app, copy/paste, done. And then instead of 'webapp.urls' it's 'personal.urls' that doesn't exist yet, we will make it, don't worry.

And we've got the path, the path ('ggg') really, the personal website is the crux of our website, so what url path should link to it? Well, the index path, so basically the string that goes caret dollar sign ('^\$'), again that means that's the beginning of the string, the string ends, so the index page, right, if there is no other link assigned to the path, right, it's just the homepage, we want to include personal urls.

So this includes personal.urls, so that's the next thing we need to make.

Tip: If some app is not installed anymore, we can delete url pattern related to that app (or even remove the folder of that app, 'webapp' in this case. I left all of the 'webapp' untouched).

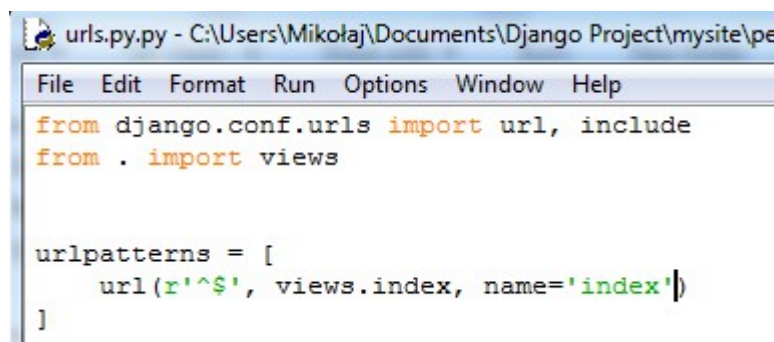
So we go to the mysite/personal folder. We don't have a urls.py in /personal and in fact we're going to create a new urls.py in the /personal and copy and paste from the other urls.py (mysite/urls.py to be exact).

We do have to make quite a few modifications, but it's going to be ok everyone.

So we can't leave this: 'from django.conf.urls import **url**, include', we actually don't need that url, but we'll leave it there. And the next thing we want to have is: we don't need the admin, so we're going to delete that ('from django.contrib import admin'), because we're deleting that let's make sure we get rid of that: 'url(r'^admin/', admin.site.urls)'.

And now what we need to do is in the local url.py file, instead of including urls like we have in the main one, what we add typically we're going to be doing is returning views, so what we need to do is remember that we go from period ('.') or relative import views: 'from . import views'.

So once you've got that and rather than the homepage, basically that url is going to say: 'hey, let's check out the other url.py and what we're saying here is if the url remains 'caret dollar sign' as in it's a homepage which it absolutely must be, we're going to return the 'views.index', because that's what it corresponds to and then, don't forget your comma, the namespace will be just 'index'. So at this point you should be questioning 'hey, do I have a views.index, right?', because we're trying to return that so save that (and exit):

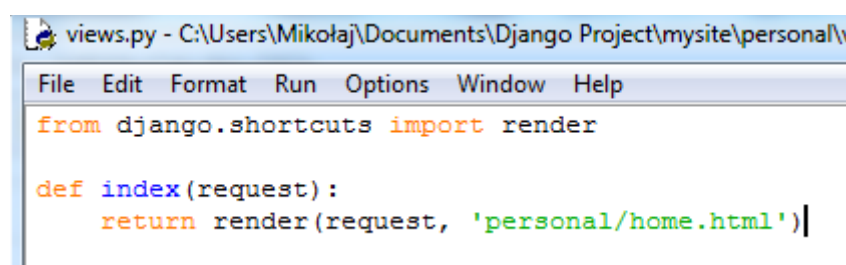


```
urls.py - C:\Users\Mikołaj\Documents\Django Project\mysite\pe
File Edit Format Run Options Window Help
from django.conf.urls import url, include
from . import views

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

Now head into **views.py** and we look and nope, we don't have a 'views.index', so let's go ahead and create one. So we're going to say 'define index', because we have to. The obligatory parameter here is always request, you have got to pass it. I think it's just to be explicit, you know, it'd be easier if didn't have to like keep passing that through, but with Django explicit is nice. Anyway, a lot of stuff is magical, we are going to return and before we return that HTTP response that's really not going to be your main method of returning anything and as you might even be able to guess that the main one is actually a 'render' right?, so if you're actually using views.py later I'll show you actually a lot of people don't even use views.py and I'll show you why, but if you are using views.py chances are you are going to be returning a 'render', so 'return render' and you've got to pass the obligatory request and then you pass the location of the HTML template you want to render, so we're going to return 'personal/home.html'.

Ok, that's all we are going to do, you can also pass a dictionary and the dictionary will be then passed to your template, which you can then use Jinja to pull apart your dictionary and do all kinds of cool stuff, but for now we'll keep it simple.



```
views.py - C:\Users\Mikołaj\Documents\Django Project\mysite\personal\w
File Edit Format Run Options Window Help
from django.shortcuts import render

def index(request):
    return render(request, 'personal/home.html')
```

Let's head to '/personal' here and what's happening? Are we going to put 'personal.html' file right

here? Noooo. So 'return' is going to..., or 'render' is going to want to return (I was thinking of Flask, the return render template I suppose I was thinking of Flask actually, anyway back on topic). This is going to render and the place that render looks is actually in a templates directory, so '/mysite/personal' and then you're going to have a new folder here called 'templates'. And here is the kicker, every app is probably going to have templates right?, that corresponds to that app and you actually can in templates you can put your code right or your html files right in here ('/personal'), that's totally acceptable, Django will find them. The problem is, consider you got your personal app website might have it, for example, a header and a footer file, nowadays the header and footer is actually all contained in the header, but you might have a homepage right?, so for example your forum might have a home, your blog might have a home.html, so then these would create conflicting names, so what Django does is it takes all your apps and then for templates what it does is it takes all your apps takes all your templates and loads them all and basically has access to all of them as if they were all stored in a single templates directory. So if you have two templates that have the identical name, this can be a problem, so you have two ways to get around this:

- never call any template the same as any other template, this can be a challenge though to remember to do,
- have a principle where in this templates directory, you have another directory that is again '/personal'. This can seem redundant, but it is going to save you. So that's what you should do, feel free to not do that if you don't want to, you just have to add that path basically when you're going to return the files, but that's going to save you and also it's part of the official documentation, so follow it.

Now we are here: /mysite/personal/templates/personal.

Anyway, in 'personal' we need a new file, a header.html file.

<html document explanation here>

If you're familiar with HTML, you've probably never seen this before:

```
{% block content %}  
{% endblock %}
```

This is Jinja. So as you can see there's really nothing here that we're saying 'hey use Jinja'. This is just used okay? So Django, Flask, whatever is just going to recognize this as Jinja. So what we have got 'block content', 'block body', what's going on here is we'll the next thing we do is define a file that's going to basically fill in between these tags, basically these are logic tags, so when you see like a curly braces and a percentage sign folowed by the closing that is Jinja logic. Conversely, the following: '{{var}}' (?) is the Jinja variable. We'll talk about both and you'll see both as we progress, but for now we've got 'block content' and 'end block', so we can now define something that will actually fill in these spaces basically, so let's save header.html.

We create a new file called home.html next to header.html and we're going to define what will go between the block content and the end block. To do that we're going to start off with some logic and the logic first will be that this 'extends' and then you give the path to what it is extending, starting with templates, so starting with the templates directory. This is going to extend 'personal/header.html'. Again, we just have one app, but later we are going to have many apps and other apps can extend this template ("personal/header.html"), even though it's not included in the templates directory of that app. It doesn't matter. Django is going to load them all and consider them all at '/templates'. It's very cool. Another thing that it will do is the same thing like for the static files, it's going to do the same thing, which I'll show you later on, so this is basically absolutely necessary that Django does this of course, but it's also very kind of confusing initially, like how is it possibly knowing how to get to these, anyway...

So this extends that file ("personal/header.html"), so when you extend a file, it's going to be looking

for 'okay, what is that block body or block content end block?'. So we're going to literally take this (from header.html):

```
{% block content %}
{% endblock %}
```

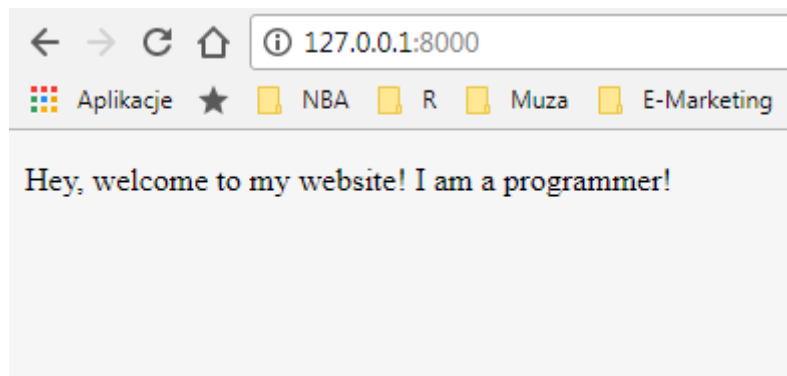
copy, come over here (home.html), paste and what we now put between ({% block content %} and {% endblock %} in home.html) is going to be simply placed between here ({% block content %} {% endblock %} in header.html). And really when you extend something, the page that you return or render is this page (home.html), so it loads up this page which is home and then it extends THIS header page, so you could load the header page, but really you're always loading the page that you were actually extending to, I suppose (tu w I suppose chodziło o zwrot extending to, czy można tak użyć w zdaniu).

We can just put some simple paragraph text in here and save that. home.html looks like:

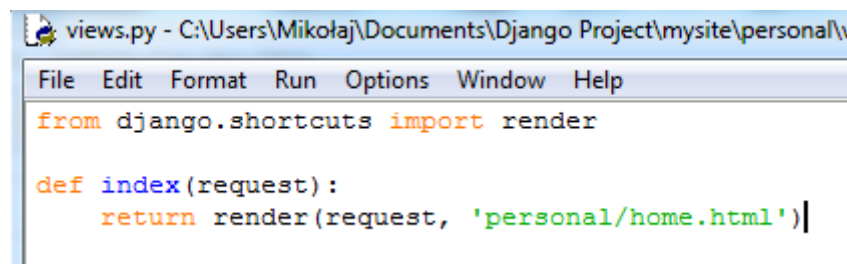
```
{% extends "personal/header.html" %}

{% block content %}
<p>Hey, welcome to my website! I am a programmer!</p>
{% endblock %}
```

Now what we have is hopefully everything we need, we'll go ahead and start server, see if we have any errors (there is one warning).



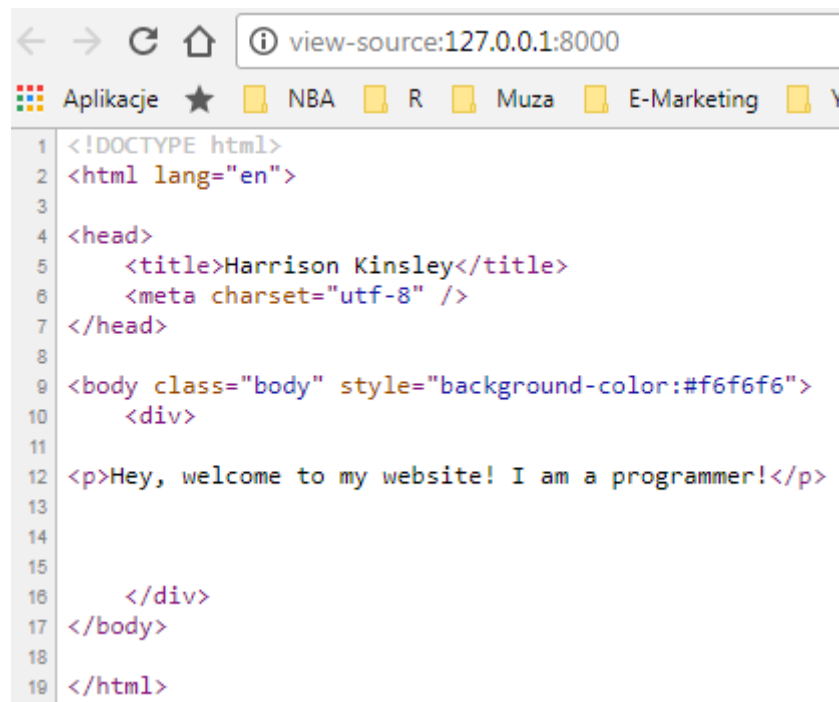
Let me go back to:



So the views.py that we're rendering, that it corresponds to this home page basically, again is actually 'personal/home.html' not header, but if view source (CTRL+U in Chrome), we'll see that we got all that other information. In fact, we don't even have the Jinja logic, we just have the HTML



that was generated by the Jinja logic, ok?, so that's pretty cool.



```
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5     <title>Harrison Kinsley</title>
6     <meta charset="utf-8" />
7 </head>
8
9 <body class="body" style="background-color:#f6f6f6">
10     <div>
11
12     <p>Hey, welcome to my website! I am a programmer!</p>
13
14
15
16     </div>
17 </body>
18
19 </html>
```

Now, the next thing I want us to go ahead and do is cover one more way of including HTML in another HTML, so for example, one thing that I like to do is like, for example, with 'extends' this can be somewhat restrictive, because like what if you want to include multiple HTML snippets in one HTML file. That can get kind of messy, especially because like which one gets included, like are they included in order, or...? So another method that we have is instead of using the 'extends' logic, we can use logic that is 'includes'.

So again we add some logic and this logic is going to be 'include' and then in quotes, again you start in '/templates' so '/templates' is assumed, then we're going to say '/personal' and then we could say a file we want to include, but sorry guys we're going to make another directory and I call this directory 'includes'. The reason for this is 'includes' is generally very short snippets of code and I like to keep things as organized as possible here. A lot of times you start off like with maybe like 5 templates and as your website grows, soon enough you have like 25 templates and then with 'includes' you might have another hundred includes or something, so I always like to have includes in a separate directory.

So we're going to say:

```
{%include "personal/include/htmlsnippet.html"}
```

htmlsnippet.html doesn't exist yet, so we have to create it(copy/paste home.html). And you'll see that actually with includes it's not so restrictive and the reason it's not so restrictive is we can do this BAM (this part at the beginning of the file is deleted: `{% extends "personal/header.html" %}`). So now it's not only going to extend a certain page or be included with a certain page, instead it can be included anywhere. And then we're not going to do this like infinite loop of including, so get rid of that too.

We add new content in the paragraph and the htmlsnippet.html finally looks like:

```
{% block content %}
```

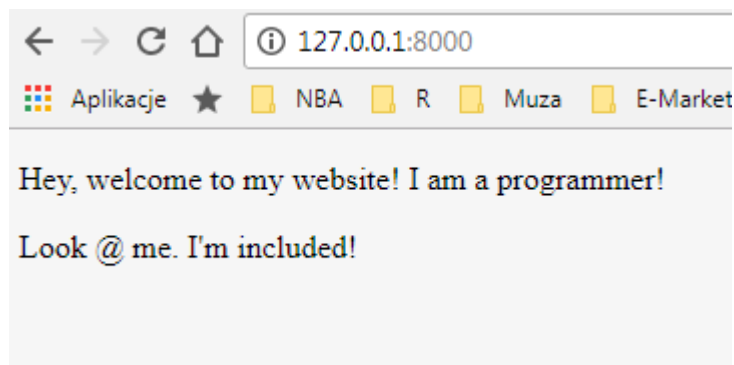


```
<p>Look @ me. I'm included!</p>
{% endblock %}
```

So then you can include this on many pages (now showing home.html), you can include it multiple times, even like you could include and include again, it's not a big deal.

So generally with 'extends' you're going to extend the pages that will be extended or pages that are going to be present all over your website. 'include' are things that are present in a lot of places, but not necessarily like everywhere. So you would extend a page that is like your header and footer of your website, something that's like there all the time. That's why you would be explicit, but with HTML snippets you just would basically only include them, because you don't actually want to extend a page and then be restricted by the "restrictiveness" of extends.

Let's come back to our website now and refresh it:



In the source code you can see that, again there's no Jinja logic, it's just the **text that was rendered**. So there's like the basics of Jinja templating, the extreme basics, we haven't covered variables and really that much Jinja logic, but as you can see, you can probably start to do some pretty cool stuff even with just this information.

In the next tutorial we're gonna be talking about is a little bit more of like styling, because right now our website obviously, I mean this is very ugly website, you wouldn't actually share this website with anybody. I'm not promising that what you will have at the very end is something you'll want to share, but it at least be a whole lot better than this.

So the next tutorial we're gonna be talking about HTML and CSS, luckily for us programmers, there is a lovely, lovely thing called Bootstrap. We will be incorporating Bootstrap into our project, so stay tuned for that.

If you have questions, comments, concerns, whatever leave them below. Otherwise as always thanks for watching, thanks for all the support and subscriptions and until next time.

## Bootstrap HTML CSS - Django Web Development with Python 4 – 22.01.2016

In this video what we're talking about is styling. We can break websites into a lot of aspects, but generally you've got more of the backend which sometimes is confused with sort of server backend, but really what we mean by backend is something that isn't presented to the user and then you've got frontend, this is the actual user interface, so the UI/UX. Generally there's quite the dichotomy here and people that are good at the backend or the website logic let's call it, are not generally very good at the user interface. So this can be kind of a challenge if you're a programmer, who wants to get into web development, but you are not a designer, you don't know much about HTML, CSS and all that. Luckily, there is Bootstrap.

Bootstrap is, it's hard to explain it right out of the gate, but basically it provides you with mainly a cascading style sheet or a CSS and some JavaScript includes that you can use on your website. And then it also gives you HTML snippets that you can make use of on your website. And basically what it does for you is it allows you to have a nice, responsive, decent looking website very fast.

So some examples of my websites that are bootstrap would be like:

- sentdex.com – this is all bootstrap, I didn't really, I wrote some of the HTML, but all the CSS is, it's just BS
- pythonprogramming.net – it's all BS ok?

So once you start to learn BS, you begin to see that like a lot of websites are using BS, you'll start seeing it everywhere.

Let's get started. Go to: <http://getbootstrap.com/> and download it.

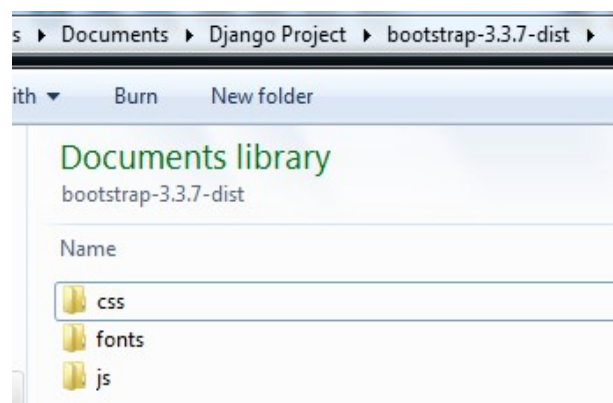
One thing to note: BS has content delivery network or CDN. If you're not familiar with a CDN ([i dalej historia...](#))

So you can use a CDN, in this tutorial we are running the code locally, because it allows to show us how to handle static files, which I mean like all your files you're not going to be able to have a CDN, unless you make the CDN and if you're watching this tutorial, chance are you have no idea how to do it anyway.

So first thing we're going to do is talk about static files and where to store them. With CDN in mind, we're going to actually download BS instead.

I downloaded it from: <http://getbootstrap.com/docs/3.3/getting-started/#download>

Unzip the folder 'bootstrap-3.3.7-dist.zip' and inside this distribution you have 3 folders: CSS, fonts and js.



Now, you can have a static directory just like 'templates' a static directory per apps, as you can imagine there may be some images that are specific to that app, also your website's main app is also probably going to be where you reference all of your CSS okay?, but you could reference or at least your main CSS page let's say, you might have custom CSS per app, but your main app CSS probably in your main website, so every app is probably not going to have a separate CSS if it's all with the same website anyways.

So we're going to go into '/personal', make a new directory and call it 'static'. We copy/paste 3 BS folders that we downloaded:

- css – the main one we are interested in is 'bootstrap.min.css', that's the one we're actually going to reference. The other ones you can have here, it's not going to make a big difference, because we're not referencing them anyways.
- fonts – this is mainly for glyph icons, which we'll talk about here in probably a moment,
- js – some js stuff, so various like popovers and maybe like notification, kind of dialogues, the modals, stuff like this are going to be using js, drop-down menus. So if you want to reference that, you have to reference these, but we'll talk about it here in a moment.

### **We have a 'static' directory. 'Templates' requires it, 'static' requires it.**

If you're old-school Django you may have actually had to hard code, where to find 'static'. If you're using a later version, you probably don't, but just to make sure you're referencing the 'static' directory, you go to the main hub directory and 'settings.py'. Make sure that you have the settings directory, it's probably at the very bottom, right here:

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.11/howto/static-files/

STATIC_URL = '/static/'
```

You can also visit the link above for some information, but you'll get it from me anyway.

**So just like templates, when you reference a static file, it's going to automatically assume it's in '/static' and just like templates you have a '/static' directory for all of your installed apps, Django is just going to mesh them all together. So for the same reason, you may want to lead them off with a 'personal' directory.**

Now, that's basically all we need now. We've got that in now how might we actually reference those files, so first of all let me talk real briefly about BS. Generally what I recommend people do, the first time you come to see BS and then also everytime you start a project I usually go to:

<https://getbootstrap.com/> (i dalej tips)

We're going to go into our website and talk about how we can at least include our static files in our CSS page. We go to: 'personal/templates/personal/header.html'.

So again, the header file is what's going to encase basically all of our other HTML files for the most part. Generally your CSS is going to be loaded in the <head> of your header, let's say.

We need to load in all available, possible static files using Jinja logic:

```
{% load staticfiles %}
```

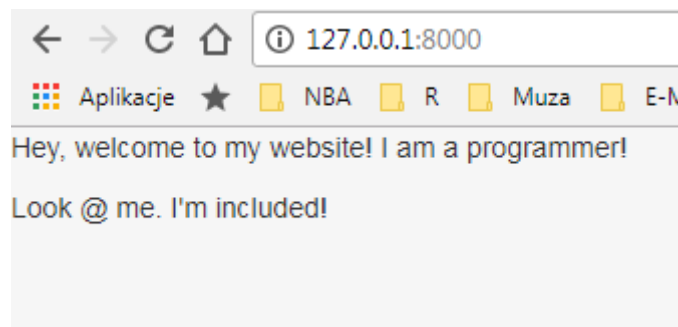
It tells Django 'hey, get ready, I'm about to reference a static file and I want to be able to reference the static files so gather up your own list of static files, so you know what I'm talking about when I reference it'. So you only need to call it once, especially if you're calling it in your header file. That means it's being called on all your other files anyways.

So now we're going to say is link, the relation here is it's a stylesheet and the reference for the stylesheet is... and here's where normally you would give the path. Let's say like on Flask for example our stylesheet were stored in '/static' of our website (href="/static/css/bootstrap.min.css"), that might be a way to reference it and if you are familiar with Flask actually you could make it a variable and make it very dynamic. Same thing is true in Django, just slightly different. Instead of hard-coding it, we can do this, so the reference will be the 'static' directory and then from that 'static' directory it was located in 'css'.

```
<link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}" type="text/css" />
```

For the same reason that we did this before, you probably don't want it to be in just straight-up 'static'. It could go either way here, because generally at least the css that we're going to use on the whole website very rarely does like a website have a custom css where that custom css completely replaces the other css. So sometimes on some pages you're going to load a "local" stylesheet, so something that kind of appends the existing stylesheet, so you still actually will reference the main stylesheet. So in this case, in the case of css, fonts and js I don't think that you have to have like a '/personal/static/personal'. You can do whatever you want, if you want to add it, go for it. **One thing that I would recommend like later on when we have like an 'images' directory, I really would suggest you do '/personal/images' and then go in or even 'images/personal/' whatever the thing is.** But for the css, fonts and js this is something that's going to apply for all pages and probably you'll be just fine this way.

So now we've referenced the stylesheet. Let's refresh our site (we're including our stylesheet now):



A couple of things you'll notice, the text is different, it's also not pure black, it's like a very dark grey, also the edges are kind of gone there, that's ok. We can view the source and we can indeed see that we're not seeing the Jinja logic here, it's the actual HTML, so in theory the user can take this static link, click it and sure enough that's our stylesheet.

```
4 <head>
5   <title>Harrison Kinsley</title>
6   <meta charset="utf-8" />
7
8
9   <link rel="stylesheet" href="/static/css/bootstrap.min.css" type="text/css" />
10 </head>
```

**So keep in mind that anything you have in static, that's all publicly available ok?, so you**

**wouldn't put anything that needs to be secret in your static directories.**

So now we have some simple stuff, let's kind of improve our homepage a bit. Kinda like before I don't see much in hand coding our HTML here, so I'm going to copy-paste HTML for the tutorial.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Harrison Kinsley</title>
  <meta charset="utf-8" />
  {% load staticfiles %}
  <link rel="stylesheet" href="{% static 'personal/css/bootstrap.min.css' %}" type =
"text/css"/>
  <meta name="viewport" content = "width=device-width, initial-scale=1.0">

  <style type="text/css">
    html,
    body {
      height:100%
    }
  </style>
</head>

<body class="body" style="background-color:#f6f6f6">
  <div class="container-fluid" style="min-height:95%; ">
    <div class="row">
      <div class="col-sm-2">
        <br>
        <center>
          
        </center>
      </div>
      <div class="col-sm-10">
        <br>
        <center>
          <h3>Programming, Teaching, Entrepreneurship</h3>
        </center>
      </div>
    </div><hr>

    <div class="row">
      <div class="col-sm-2">
        <br>

        <br>
        <!-- Great, til you resize. -->
        <!--<div class="well bs-sidebar affix" id="sidebar" style="background-
color:#fff">-->
```

```

        <div class="well bs-sidebar" id="sidebar" style="background-color:#fff">
            <ul class="nav nav-pills nav-stacked">
                <li><a href="/">Home</a></li>
                <li><a href="/blog/">Blog</a></li>
                <li><a href="/contact/">Contact</a></li>
            </ul>
        </div> <!--well bs-sidebar affix-->
    </div> <!--col-sm-2-->
    <div class="col-sm-10">

        <div class='container-fluid'>
            <br><br>
            {% block content %}
            {% endblock %}
        </div>
    </div>
</div>
<footer>
    <div class="container-fluid" style='margin-left:15px'>
        <p><a href="#" target="blank">Contact</a> | <a href="#"
target="blank">LinkedIn</a> | <a href="#" target="blank">Twitter</a> | <a href="#"
target="blank">Google+</a></p>
    </div>
</footer>

</body>

</html>

```

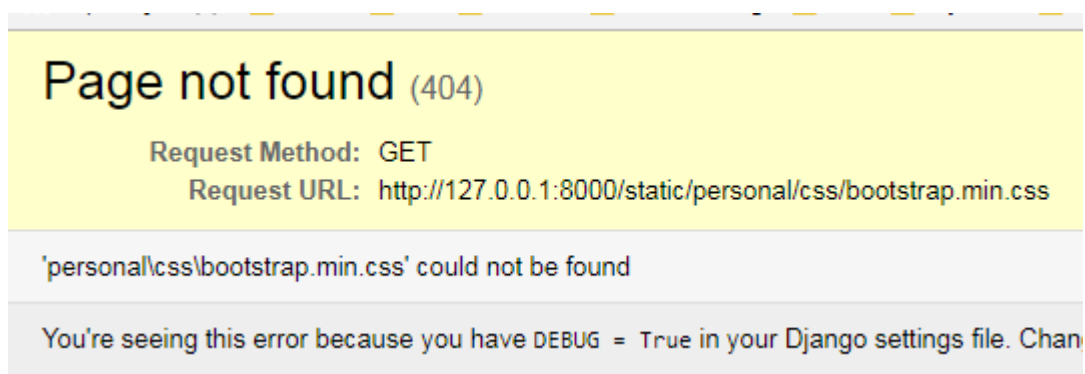
Our BS files have not been loaded. We can check the source code and see that the path is not correct.

```

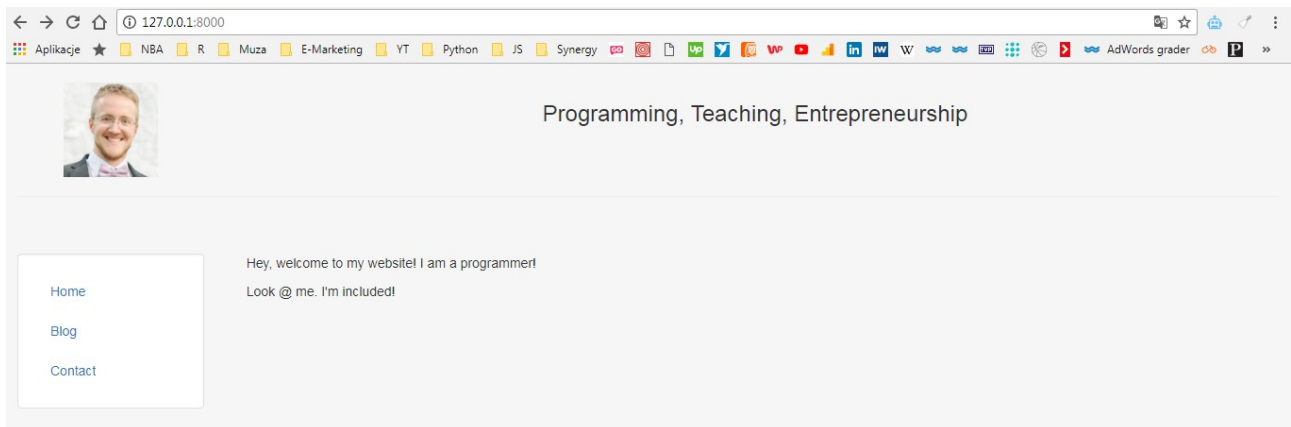
<link rel="stylesheet" href="/static/personal/css/bootstrap.min.css" type = "text/css"/>
<meta name="viewport" content = "width=device-width, initial-scale=1.0">

```

When we click in the present path URL we can see:



It turns out our static files path includes 'personal' folder. We can then change the path in the code or create a 'personal' folder and put all the BS files in there. We are going to do the latter. We also need to upload the missing photo. Correction and BUM, we get something like:



So obviously we're not winning any awards for this website, but it at least looks ok. So let's explain the code real quick.

```
<head>
  <title>Harrison Kinsley</title>
  <meta charset="utf-8" />

  <link rel="stylesheet" href="/static/personal/css/bootstrap.min.css" type = "text/css"/>
  <meta name="viewport" content = "width=device-width, initial-scale=1.0">

  <style type="text/css">
    html,
    body {
      height:100%
    }
  </style>
</head>
```

<head>:

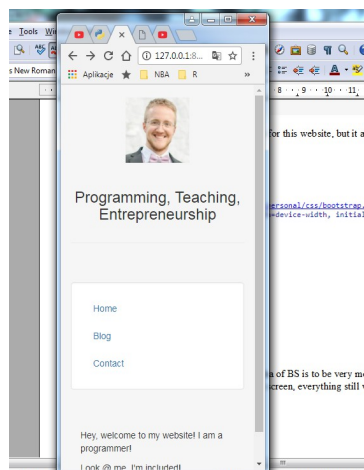
title

charset – character set(tings?)

we load the static files

we load up the css page

This is the metatag basically for BS. The idea of BS is to be very mobile-friendly. We've put some pretty basic HTML, but if we squish up the screen, everything still works:





Other code and BS explanation

**Tip: Mimicing screens in Chrome. F12 -> click on mobile icon**

In the next tutorial we are going to start actually building out our website. As you can see we have a home page, a blog and a contact page. So all that stuff is something that we actually want to build here.

## Passing variables from python to html - Django Web Development with Python 5 – 24.01.2016

In this tutorial we're going to talk about we're going to be creating a contact page and the way that we're going to do it is by passing information from our Python file to our HTML file using Jinja variables and some Jinja logic. The idea here is on our homepage we showed how you can make a really basic home page and this information here is all included via HTML, but we haven't actually talked about how you could take information from a Python file and pass that information to your HTML and then finally on to your user who's actually viewing your website. So the homepage, really basic, contact will be a little more complex and blog will be obviously much more dynamic.

In this example we're actually going to pretty much be hard coding the variables that we're going to pass through, but later on you can envision how you might connect with maybe mysql, pull some information from the database and take that information and pass it on to the template.

Let's get started. So first of all we're going to be building a contact. So should the contact page be its own app or should it just be part of maybe the personal page? Well, if it was going to be like a contact page where maybe you had like a contact form for the user to fill out and all this kind of stuff, it probably should be its own app. In our case it's just going to have some information like 'hey, if you want to contact me, here's my email'. So in that case this just is part of the personal app.

So the first thing that we need to do is not start an app and install it, we first need to access the views.py and the urls.py in '/personal'.

In the views.py right now we have a view for the index, index request basically, and now we're going to do is for a contact request, so in this case we're going to define 'contact'. We still pass 'request' as usual and then in here we're going to return 'render', and then we're going to return obviously the 'request', because we have to and then we're going to return 'personal/' and instead of any of these other templates that exist 'basic.html'.

So my thinking here with having a basic page is a lot of times with a website you're going to have a page that return mainly just some text, maybe an image, but it's just a very, very simple page, so something like a contact page, something like a technology page, about me page. In this case the whole website is about us, but these are just examples, these pages are very simple, so why would you have a specific HTML page for each of these?, that's silly. So we have a basic HTML page and this page is just like all the others, only it takes parameters or a variable that is the content.

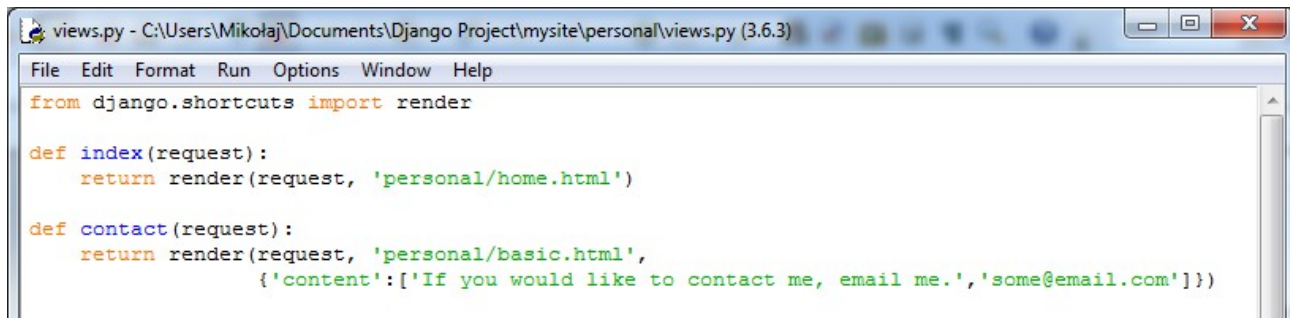
So now we are going pass this dictionary. As I've said 'render' takes basically 3 parameters:

- first is 'request', you got to have it,
- what you're going to render, which is the HTML file, you really got to have it,
- and then finally an optional dictionary.

So in the dictionary we can store things. For example we can store something under the name of 'content', so 'content' is the key of the dictionary, the value will be a 2-element list filled with text:

```
def contact(request):  
    return render(request, 'personal/basic.html',  
                  {'content': ['If you would like to contact me, email me.', 'some@email.com']})
```

So right now we're just going to basically pass one thing, so in our basic HTML, because we're passing this ('content'), we can reference 'content' and the value of 'content' is the value in the dictionary of 'content'. So it will be a list of two strings. The whole file looks like:



```
views.py - C:\Users\Mikołaj\Documents\Django Project\mysite\personal\views.py (3.6.3)
File Edit Format Run Options Window Help
from django.shortcuts import render

def index(request):
    return render(request, 'personal/home.html')

def contact(request):
    return render(request, 'personal/basic.html',
        {'content': ['If you would like to contact me, email me.', 'some@email.com']})
```

The next thing we want to do is create the 'basic.html' in the 'templates' folder. We copy paste 'home.html' and edit the file. This file will indeed extend 'header.html', it does not need to include anything.

So this file, as I said, is going to just display basic information from that 'content' key of the dictionary, so that 'content' key is a list of strings that we want to show. We need to iterate through that list with Jinja and then obviously we can display the strings with Jinja as a variable which we haven't shown actual variables, I've shown you how to do it though, so here we'll actually do it.

We're passing that dictionary through, so that dictionary just comes to this HTML file and then we can begin referencing it like this.

\*Our dictionary key is called 'content', but it will not conflict with '{% block content %}'.

\*There is also [tutorial about dictionaries on sentdex's pythonprogramming.net](#)

In Jinja you start a for loop like this: '{% for c in content %}', but unlike Python, like HTML is blind to spacing and white space, so you have to close out for loops: '{% endfor %}'. The same is true for if-statements, example:

```
{% if c=='hi' %}
{{c}}
{% endif %}
```

\*there is also 'else' and all this kind of stuff

In this case we just want to print it out, so we'll have it in paragraph tags and then we just reference the variable 'c' and that's all we need here.

```
{% extends "personal/header.html" %}

{% block content %}
    {% for c in content %}
        <p>{{c}}</p>
    {% endfor %}
{% endblock %}
```

So at this point, are we ready to push live? Nope! We don't have anything that leads us to the contact page. So we still have to edit the urls.py in '/personal'. So then we're going to add a new url:

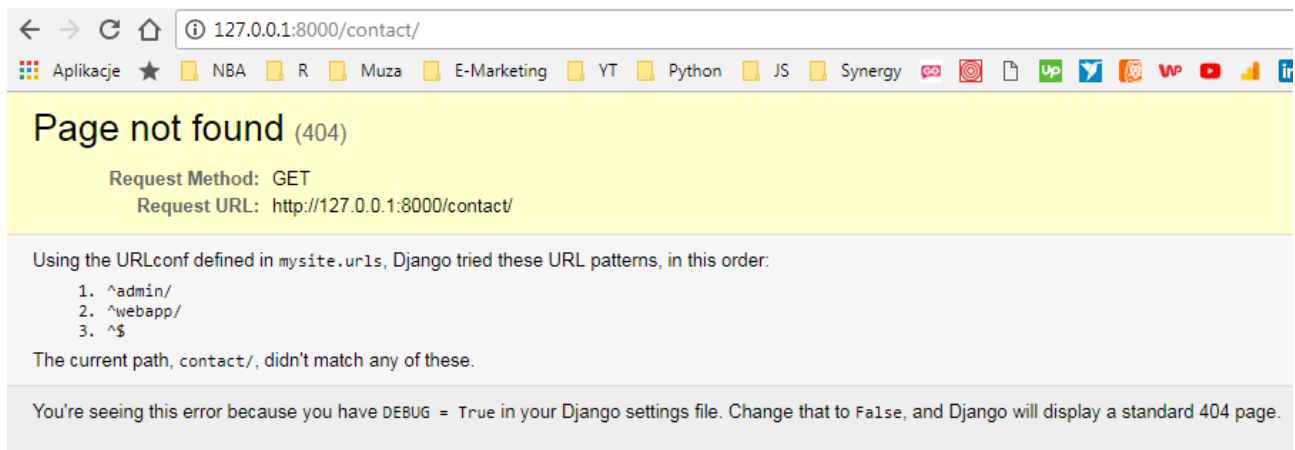
```
url(r'^contact/', views.contact, name='contact')
```

```
urls.py - C:\Users\Mikołaj\Documents\Django Project\mysite\personal\urls.py (3.6.3)
File Edit Format Run Options Window Help
from django.conf.urls import url, include
from . import views

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

We've already done the main urls.py, so we lead to this one and if it starts and ends it goes here, and if it starts with 'contact/', it is we want to return contact and you could even close off 'contact' here with '\$'.

Now in theory, we're ready, but as you will see, we're actually going to wind up with a little bit of a problem. So we will come over to our website, let's refresh, then we click on the 'contact' link and turns out we get a 404:



Now, a couple of things will happen. First of all we can look at the 404 error on our website and we can see it can only be the case is 'DEBUG' is set to 'True' (last grey line). When you push this website live, you will want 'DEBUG' to be equal to 'False'. You'll only want to debug when you're really like developing, you should never have debug equals 'True' on like a live production website, because it can cause problems.

So in our error we can see what Django attempted. Django basically tried to read the url and asked the question: 'does this url start or begin with 'admin' or does this url start and then end?' (additionally I have also 'webapp' url which I didn't delete before). So where is this list from? The list is coming from the main urls.py ('mysite/urls.py').

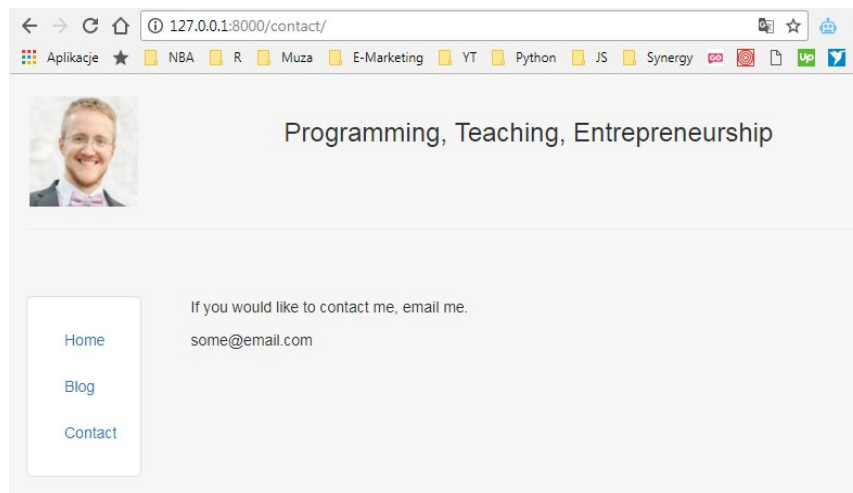
'contact/' is neither of those, so it never gets asked or sent to 'personal.urls' to further see about any other urls like our 'contact/'. Also, we can come over to this error here and we can even see that warnings:

```
System check identified some issues:
WARNINGS:
?: (urls.W001) Your URL pattern '^$' uses include with a regex ending with a '$'
. Remove the dollar from the regex to avoid problems including URLs.
System check identified 1 issue (0 silenced).
```

So as Django suggests in the warning, we remove the dollar sign from the regex:

```
url(r'^', include('personal.urls'))
```

Let's come back over now to our website, hit refresh and voila, sure enough everything worked:

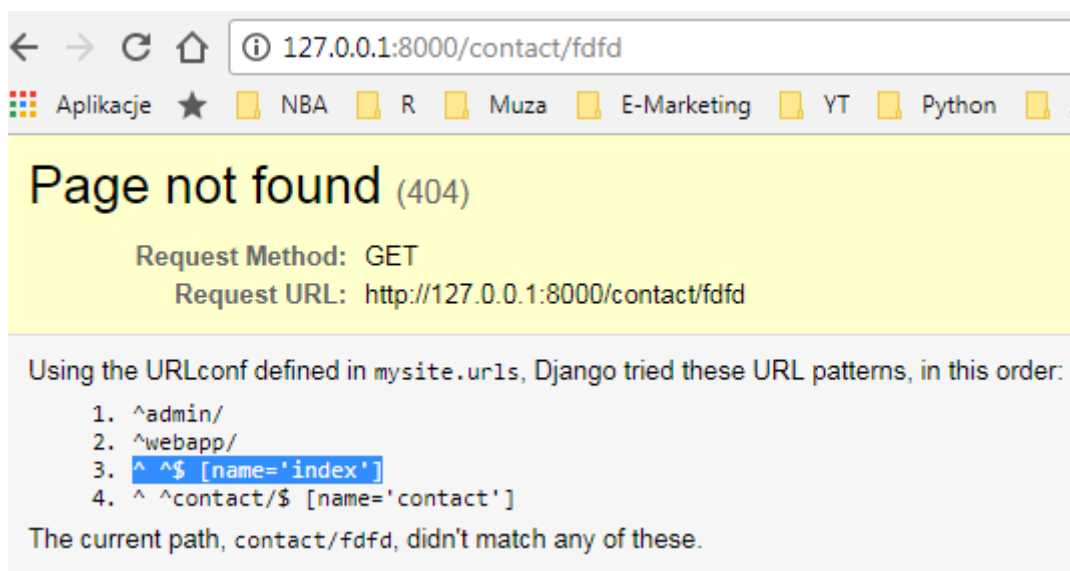


And if we did '/contact/fdfd', you'll see that it still works and it will continue returning to contact. Why is it doing that? Because we never actually finished that url with a dollar sign (in personal/urls.py):

```
url(r'^contact/', views.contact, name='contact')
```

vs

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



You can see in this example how Django is going through checking. First url patterns from the main hub and then url patterns from the next app. It can be seen in numbers 3. and 4 (main urls.py and then after space '/personal/urls.py'). Number 1. and 2. are only in the main hub urls file. You can see what Django attempted in which order and then you can start to debug from there.

As you might be able to guess, what's kind of neat about these regular expressions is you can begin to make some pretty crazy url patterns that have never been able to be done in the past, when like the url path was actually exactly the path on the server so you could only have paths that were like

legitimate operating system paths, whereas with Django you can have some pretty cool, hacky looking paths that are just neat to have and you can have various symbols and stuff like that, that you just wouldn't have been able to have in the past.

Anyway, so that's our example of building our contact page, pretty simple just like the homepage, but we actually learned how we can pass variables and even do a little bit of Jinja logic with our 'for' loop as an example there and then using the content as a variable to just put out to the screen. So there are some basics and now as we continue on in the next tutorial what we're going to be talking about is the blog. The blog is really where like if you think you've seen the cool stuff about Django up to this point, you haven't seen nothing yet. So blog is going to show us the true magic of Django, because so far we really haven't made use of models, I haven't shown you the admin (admin page) and all that which is actually really impressive for Django, maybe one of the best selling points of it if you ask me and even the models.py, as you'll see.

## Beginning Blog - Django Web Development with Python 6 – 26.01.2016

In this tutorial what we're going to be talking about is implementing the final step of our personal website which is adding a blog. So the first thing we need to ask ourselves when we go to add a blog is 'is it going to be part of the personal app or do we make a new app?'. Well, it makes most sense, to me at least, that the blog is a free-standing kind of app, so we should make a new app. That's what we are going to do first.

So let's go ahead and come to our terminal and start a new app called 'blog' (he has some macro that types?)

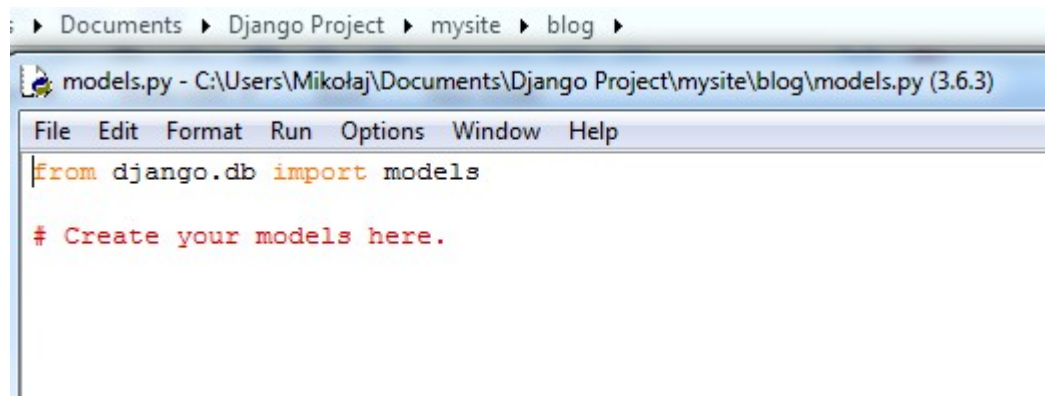
Then we install the new app.

Then you got to add some sort of pointer to it, so in the main urls.py we need to specify a url path that will lead someone to the blog:

```
url(r'^blog/', include('blog.urls'))
```

If we want to include 'blog.urls', of course we need to make 'blog.urls', but the first thing we're actually going to do is we are going to make a model, so this tutorial we will be talking a little bit about the idea of models, because up to this point we actually haven't talked about Django models. Django models are pretty much one of the most valuable things that Django is going to, at least the Django paradigm is going to run off of and give you as far as functionality.

So let's go into the 'blog' directory and edit this 'models.py'. You should already have this information here, it's just by default, Django just automatically populates this. I'm going to get rid of this comment though.



So in a model, you've got basically your whole 'models.py' would be this is your Django models, you can think of it like your entire database. And then within your 'models.py' you're going to have various classess. Each class you can think of like a table in your database and then each variable within that class you can think of like a column. From there you can define various attributes to your data, so this would be like your metadata and I think the best way for you to understand that part is we just do it and then I'll explain it once you see it.

This going to be a blog, so what does a blog need? A blog needs at least posts, right?, so we're going to say 'class Post', so this will create a table called 'Post' and then 'models.Model'. So you might be thinking 'well, how does that create a table?'. If you recall back in the beginning we looked at our 'settings.py' file.



```
# Database
# https://docs.djangoproject.com/en/1.11/ref/settings/#databases

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

In here you define what database you want, the default is always 'sqlite', because that's a standard library with Python, so you already have it. So that's already defined, so what Django does is it knows where your backend is, it knows where your 'models.py' are and it just generates all the information required. It creates your tables and all this kind of stuff, and you'll see how it does that here. Probably not in this tutorial, but in the very next tutorial.

Anyway going back to the 'models.py'. So this is a table basically, it inherits from the 'models.Model'. Now the next thing we're going to define the columns in our table. So what is the information we might need with our 'Post'? Well, you probably have a 'title', you're going to have a 'body' information or the post itself and you might have a 'date' assigned. For each column if you are familiar with making databases, tables and stuff like that, you know that every column and when you define a column, and you actually create a table, let's say if you're creating a table yourself you specify the column name and then the data type, and then any constraints, possibly within that data type. Django does the exact same thing, only Django has its own model fields and again it will convert from the Django models to the most applicable data type possible for the database backend that you chose.

I'm going to bring over this link here:

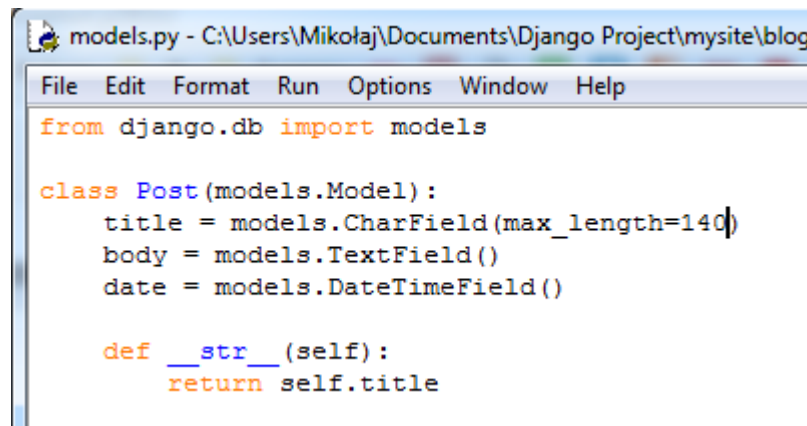
[<link>](#)

This is just from the Django documentation (again, all tutorials also have text based tutorials, I link to those at pythonprogramming.net). This just gives you all the possible data types and constraints, and stuff like that your columns.

So for 'title', this will just be some examples, but for 'title' this one's just going to be a 'models.CharField()', so it's a character field. And then for 'body' this will be a 'models.TextField()'. So the idea with the character field, this is like a shorter character field and then a text field is usually going to be like a, you know, a text field as it sounds. For 'date' this would be a 'models.DateTimeField()' and of course like for data you can actually have it be a CharField if you wanted okay?, but if you make it a DateTimeField you can do all kinds of date/time logic and just use date/time as it's a date stamp basically within Django. So it's important that you use the most applicable one if you can. So if you're familiar with like a database backends, you could in theory call everything a 'var char' (?), right? But that's not the best idea, but it's possible, but not the best.

So then, as I said, you know call a name datatype and then any constraints you want. So an example of a constraint might be a 'max\_length' and then this would be '140'. Again you can always go to this Django models/fields docs and kind of read through that for everything you need there. So that's everything we need to create a table, but one thing you need now for like metadata purposes is, for example if you want a reference post like the idea of having like this will create the table and everything in our database, it'll be great, but then like later on let's say you want a reference like maybe you want to list out all the posts ok?, or at least like all the post titles, when you go to do that, you're going to have a little bit of a problem, because if you try to reference this,

any element that's a, you know the basically this post object, what's going to happen is here I try to reference like 'Post.title' that's a 'Post' object, right? It's just gonna literally be like 'Post' object and maybe at the address where it's stored you know on the hardware and that's not a good thing. You want to be able to actually get the string text of the title, so we add a new method here and that's just going to be defined 'def \_\_str\_\_(self):' and then it just simply returns 'self.title'. So what this does is it will just return the title, so if we reference 'Post.title', that will be a string rather than the, you know, just calling it literally post object at, you know, wherever and if you're following along in Python 2 – shame on you, but if you are, you change this to Unicode rather than string, just a fundamental difference between Python 2 and 3.

A screenshot of a code editor window titled 'models.py - C:\Users\Mikołaj\Documents\Django Project\mysite\blog'. The editor has a menu bar with 'File', 'Edit', 'Format', 'Run', 'Options', 'Window', and 'Help'. The code inside is as follows:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=140)
    body = models.TextField()
    date = models.DateTimeField()

    def __str__(self):
        return self.title
```

Ok, so that is our model, now the next thing we need to do is basically define the 'urls.py', some sort of view for the user and then we are ready to go live. So I'm going to cover the urls and the view in the next tutorial.

## What is the fastest way to learn Django?

42 Answers

Chris Bartos

Chris Bartos, I'm a professional Django developer

Answered Jan 15

Maybe you want to learn how to make websites for fun or for business. Maybe you might be working on certain projects. Maybe you are trying to put something on your resume.

When I first learned how to program I was 12 years old. My mother got me a CD-ROM for my birthday called "Learn to Program BASIC". BASIC is a very basic programming language. "Learn to Program BASIC" was more of a game than a tutorial and for a 12 year old, that was what I needed. I wanted to play games!

That game changed the course of my life. It made me pursue Computer Science as a degree and when I finished my degree I found a job and I became a professional programmer. I started making websites and over time I mastered Django, PHP, Ruby on Rails, Java, etc.

From my 12 year old self to now, I kind of learned how to learn how to program. And I'm going to share with you the secret.

Now, the WORST thing you can do when you're learning how to program is to read the entire documentation. Reading the documentation should be used when you need to know how to do certain things.

When you're first starting out the BEST thing to do is to take a tutorial. The tutorial should be SIMPLE and STRAIGHTFORWARD. The reason for that is, you shouldn't take tutorials to learn everything you need to know. You take tutorials so you can get a bird's eye view of the technology you are learning.

The next thing you should do is brainstorm simple projects that you can create with that new technology. For example, start with a blog. What do you need in order to create a blog?

Well... you need a post and you need at least 1 user. You can use these two entities and create models out of them.

What do you need to create a post? Well... you need a `creation_date`, a user that created that post, title of the post, body of the post.

What do you need to create a user? Well... Django already has a User model that you can use. Why not start out with the built-in model instead of re-inventing the wheel?

Now, you have models and each model has certain attributes and you have a relationship between the two.

etc. etc. etc.

This is your thought process. While you are trying to build a new project, when you come upon something you don't understand or you're not sure what to do, THEN you go and ask or find the answers in the documentation.

After you finish your project, then start a new project and repeat the process!

Why is this the BEST way to learn?

Because you are building something useful or cool.

You figure out how to do something when you need to know it instead of reading boring documentation

You can build from that understanding by modifying your project and try new things (see if things work)

Building things allows you to put those things on your resumé!

I want you to brainstorm a project that you can make putting together everything you've learned.

What project are you going to build now? Start building it slowly and simply. Then, when you get something you're unsure about, go search for an answer.

Chris

---

Socratica yt channel about Python classes and object

z czym mam do czynienia (obiekt, klasa), jak to działa, co się tutaj dzieje, jak działa cały mechanizm, szperanie w kodzie źródłowym (azwa.nazwa.nazwa2)  
stay open minded, patient., wizualizacja sukcesu

G: django tutorial import -> [djangobook.com](http://djangobook.com)

G: what is python module -> [docs.python.org/2/tutorial/modules.html](https://docs.python.org/2/tutorial/modules.html)

## Blog View and Template - Django Web Development with Python 7 – 28.01.2016

In this tutorial we are gonna be building up the last tutorial creating our blog. So in last tutorial we built this 'model.py' file. As I said, basically the class is the table, each variable here is the column in the table and then these are the data types and some constrains ('max\_length ...') basically for those. And then this is just a meta information ('def \_\_str\_\_(self): ...'), so we can reference the string values of these attributes and stuff.

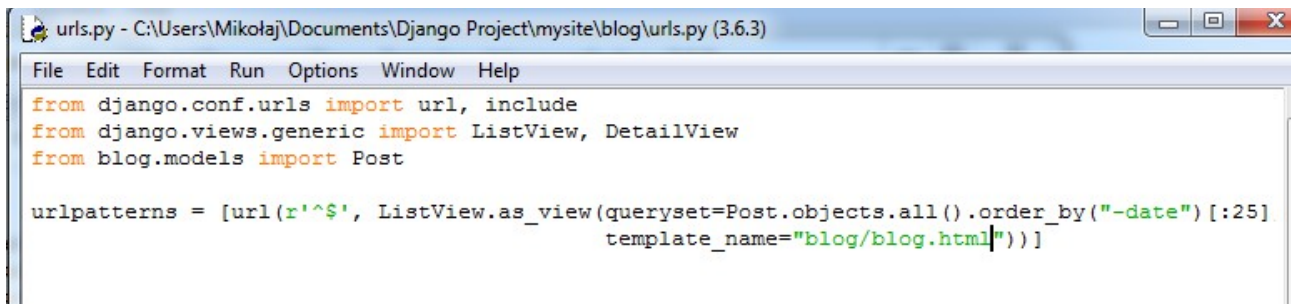
So that's our 'models.py'. Now our wisdom of Django up to this point tells us we still need a couple things. We need the 'urls.py', we need the 'views.py' and we probably need a template, right? But what I'm about to show you is a way to kind of circumvent the need for creating a 'views.py'. Django offers us what are called 'generic views' and we pull some of those over real quick. So this is an example here:

<https://docs.djangoproject.com/en/1.11/ref/class-based-views/generic-display/>

We are going to be using 'DetailView' and 'ListView'. The first one we're actually going to use is 'ListView' and then later on 'DetailView'. So as time goes on you'll find that probably most people wind up using generic views most of the time and then right after maybe 'DetailView', 'ListView' and then render from from 'views.py', so you can render just anything, but you'll see why these views are really useful here in this, in the next tutorial as well.

So moving that aside, what we're going to do is we're going to close out of this and create a 'urls.py' file. We won't actually need to create a 'views.py'. Well, 'views.py' already exist, but we won't need to edit it. So copy/paste 'views.py', call it 'urls.py', edit it and clear it out.

So we're going to do 'from django.conf.urls' we're importing 'url' and 'include' just the same we've always used, then 'from django.views.generic import ListView' and we're going to use 'DetailView' later so I'll just go ahead and import it, but here we're going to use 'ListView'. So then 'from blog.models' we're going to import 'Post'. So what's going to happen here is basically when we go to the Blog, what are we going to return when a user like first clicks on the Blog navbar button, what are they expecting to see? I'm going to see like the latest blog or a blog, or a random blog, they're going to see probably a list of blogs, hence our 'ListView'. So let's define our 'urlpatterns', these are a list, the first pattern would be 'url(r'^\$', because, keep in mind, this is already leading with 'blog/' (i.e blog homepage), so that's the pattern that we want. Next 'ListView' and if you recall before that we would return 'views' here, but instead we're going to return 'ListView.as\_view' and then we're going to say 'queryset=' this is going to be the data that we're going to pass through here from the database, so this will be 'Post.objects.all().order\_by()' and we can order by any of those columns basically, but we want to order by the 'date' column. We would do a negative date '-date', so this would be a descending date, so the latest posts are first. That's really all you need, but at some point you would want limit this, especially if you had like a 100 blogs or something, so you could do something like this '[:25]' up to the first 25. Then after that we would add this to the 'template\_name' of, and we don't currently have it created keep in mind, 'blog/blog.html' (Django starts with '/templates' folder by default). And that should be it, so that's our 'urls.py' file.



```
urls.py - C:\Users\Mikołaj\Documents\Django Project\mysite\blog\urls.py (3.6.3)
File Edit Format Run Options Window Help
from django.conf.urls import url, include
from django.views.generic import ListView, DetailView
from blog.models import Post

urlpatterns = [url(r'^$', ListView.as_view(queryset=Post.objects.all().order_by("-date")[:25],
                                           template_name="blog/blog.html"))]
```

Now what we need to do is actually create this template which will be 'blog/blog.html' in the 'templates' directory. Chances are we don't have another template that's going to conflict with 'blog.html', but again, it's just kind of best practice, so you don't have to think about it, it's always going to be led by that 'blog/' name, so you're just not going to see come any conflicting templates.

So once you've got that, you got this 'blog.html' (open), we're ready to populate this with some information, so this one, first of all as usual, it's going to extend and this extends (again 'extends' always starts with the idea that it's 'templates' first) the 'header.html' which is located in '/personal/header.html', so it extends 'header.html'. Anytime you extend something you're going to have 'block content' and 'endblock'. And now we're actually going to put the content, so as we said before this is a 'ListView' and we are listing out all of the available blog entries. So what we would do is if we're going to list out something chances are we are going to use a for loop to iterate through it. So we're going to say: `{% for post in object_list %}` and in fact (let's pull up the 'urls.py' file again).

So this was the query ('queryset') that we're running, `object_list` again this might feel a little magical, like where do we pull that from? It's coming from that 'objects' here, it's an 'object\_list', 'all' of the objects, the blogs ordered by the date, but it is everything in that blog, so it's a 'Post' object and what's going to have, it's going to have the title, the date and the body information when we pull it. So anyway that might seem somewhat magical, but that's how these detailed views work. So then (coming back to 'blog.html') every time we start a for loop, we might as well end it with `{% endfor %}`, because we don't want to forget to do that. And now we'll come in here and we will just put a header five tags '<h5>', so this will be like each url and then what we're going to do is first we want to have the date, so how might we reference the dates? If we pull up (open) the 'models.py' file, basically the date column is literally just date okay?, so when we reference date we say something like this (back in 'blog.html') 'post.date', we're saying for 'post' and 'object\_list'. 'object list' is a list of objects of the capital P 'Post' class, but as you hopefully know, when you do a for-loop, you're basically defining a new variable here, so each object we're calling 'post', each object is going to have a title, body and date attribute. So when we say 'post.date' that's referencing the date column of this specific 'post' object. So 'post.date' and then we're going to filter for the date 'date:' and we're going to say the display mode for the date will be capital Y, lowercase m, lowercase d "'Y-m-d'". So this is a full year, so 2015 or 2016, month would be just two digit month and day would be two digit day, okay? So this is an example of why you would want 'post.date' to be a date-time field. Two reasons already that we're using this. One is when we go to order it in the 'urls.py' here when we go to order by date. The only reason we're getting away with that we could probably do it with var char (?) and we'll go by maybe digits and it might still work depending on which order you did it, but this is how we're able to do that. And then two is we can format the dates, because it's a datetime object.

So once you've got the date what we're going to do now is allow the user to actually click on the post title, so to click on it we would need obviously some link tag here, so '<a>' and then the reference 'href=' will start `/blog/{{ post.id }}`. So if we look back at our models, we don't have an id field (we have a title, body, date). Turns out Django is just going to do that for you and the id, if you're familiar like pretty much everybody's going to have like a self earn autoincrement primary

key as the first column okay?, Django is going to do that for you automatically and you're going to see why that was very nice of them later on.

So anyway, 'post.id' that's referencing that very first column that just it starts at zero and it continues autoincrementing as time goes on. So in the url it will link to that '/blog/' whatever that 'post.id' happens to be and then we'll close off that link tag '</a>'. And then in the link, what do we want the link text to basically say? Well, we want it to be a '{{ post.title }}' and really that's it.

'blog.html':

```
{% extends "personal/header.html" %}

{% block content %}
    {% for post in object_list %}
        <h5>{{ post.date|date:"Y-m-d" }}<a href="/blog/{{ post.id }}">
        {{ post.title }}</a></h5>
    {% endfor %}
{% endblock %}
```

That should be everything we need up to this point. So this should be relatively simple, the only truly magical thing is this 'object\_list'. You'll just have to take Django's word for it that it populates the object list variable. I think that'll be it for this tutorial, in the next tutorial we'll talk about migrations and what's going on there and then after that we'll probably have a live blog.

G: what is web application

<https://www.fullstackpython.com/object-relational-mappers-orms.html>

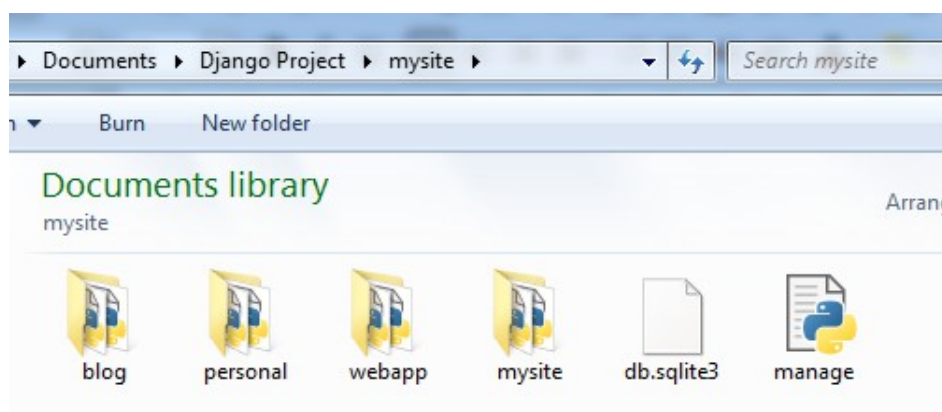
## Database & Migrations - Django Web Development with Python 8 – 30.01.2016

In this tutorial what we're going to be talking about is migrations. So basically if you recall, every time we start an app, one of the first things we need to start thinking about is install that app and create like a 'urls.py' to that app. When you create models the first thing that should pop in your head is we need to make migrations, okay? So do you remember how I was telling you that models correspond to your database information (open up 'models.py'). So this is your table, columns, data types. What Django does is it separates you almost entirely from actually writing any SQL or any other database information or language. So it's pretty high level, so when that happens sometimes things can seem a little too magical or sorcery, or something like this, but what it does is basically it takes your model and in the backend it reads whatever your database backend is and it automatically generates the SQL required to make modifications, insert to your database, make updates and even deletes. So we're going to be talking about here is how that really works, okay?, or at least how it works with you. So what we're going to do is we're going to close out our models and like I said we need to do a migration. So if you remember way back when we went to go run the server and it gave us like a little bit of an error, it wasn't like a stopping error or anything, or a breaking error. It was just like a notification like 'hey you've got migrations that you need to make and that was because of the admins or the admin app which you never created an admin app, but it does exist with all Django, well, at least by the default. So if we go to 'mysite/mysite/setting.py', you'll see that there's actually quite a few apps here.

```
# Application definition

INSTALLED_APPS = [
    'webapp',
    'personal',
    'blog',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

You've got the admin app, auth, sessions, messages, content type static files and so on. So you do have other apps that you probably didn't even create. So you have to run migrations anytime you've got models and if we go back here you'll see that we do have a 'db.sqlite3':

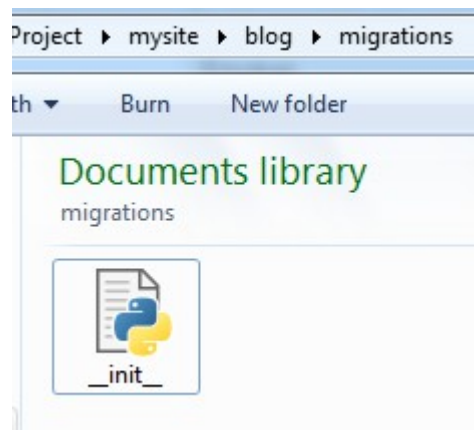




It's only 36 kilobytes (mine db has 128 KB), it's not really populated with anything, so there's really probably nothing special there.

So what we're going to do is we're going to make these migrations. So when we go to make migrations we'll probably find that we have two major migrations that we wind up making. One is for the 'models.py' and the other one's going to be for admin unless you already migrated admin.

So if we pop in to 'blog/migrations', you'll see there's really nothing here besides an '\_\_init\_\_' file, that just says like : 'hey treat me like a package'.



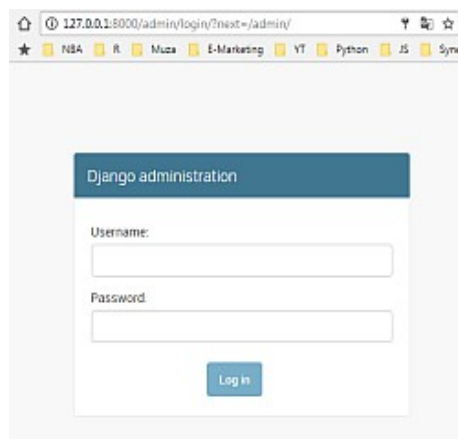
Otherwise there's really no other migration, let's see 'personal' has some migrations, but 'mysite' really doesn't have anything. Anyway, so each of your apps is going to have a migrations, that's specific to it, so we'll see here in a moment. So let's go ahead and we're going to run the server. First, as usual, it's a 'python manage.py runserver' and it actually didn't find any issues (same for me):

```
Administrator: C:\Windows\system32\cmd.exe - python manage.py runserver

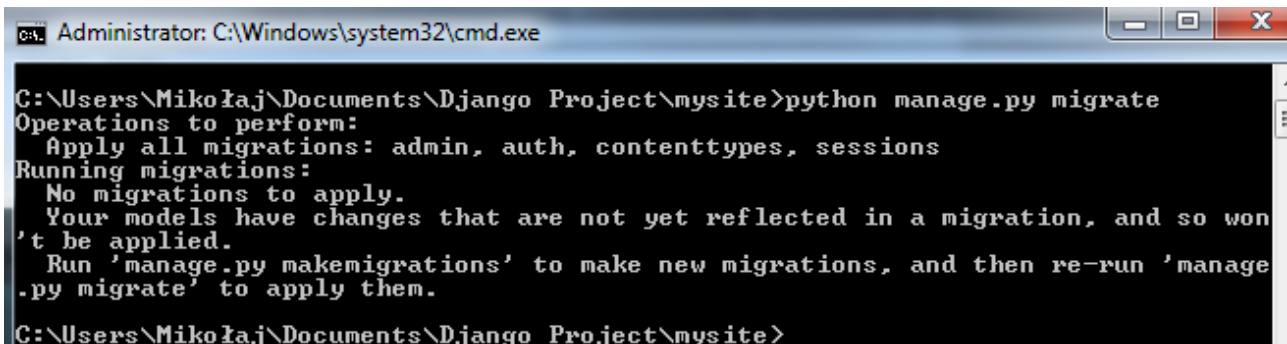
C:\Users\Mikołaj\Documents\Django Project\mysite>python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
November 20, 2017 - 09:54:18
Django version 1.11.6, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

I possibly run the migraton for the admin. Let's go ahead and visit our website real quick. So this is our website, everything is sort of running at least. Now let's go to '/admin' and so far it looks like admin actually does work:



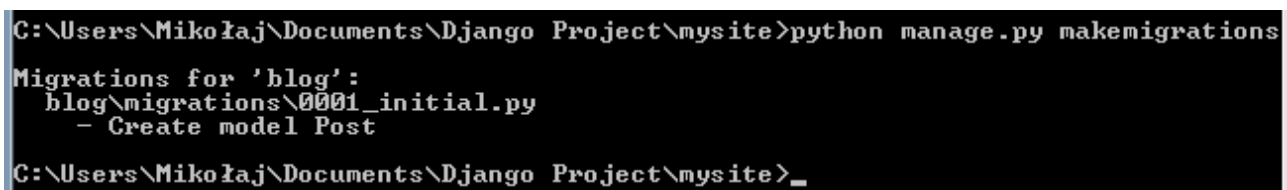
We're not going to be talking about admin here, but I didn't think I'd actually set up admin on this specific one. That's ok, let's go ahead and break this and we're going to go and do 'python manage.py migrate', run that and so maybe we've already done, yeah.



```
Administrator: C:\Windows\system32\cmd.exe
C:\Users\Mikołaj\Documents\Django Project\mysite>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  No migrations to apply.
  Your models have changes that are not yet reflected in a migration, and so won't be applied.
  Run 'manage.py makemigrations' to make new migrations, and then re-run 'manage.py migrate' to apply them.
C:\Users\Mikołaj\Documents\Django Project\mysite>
```

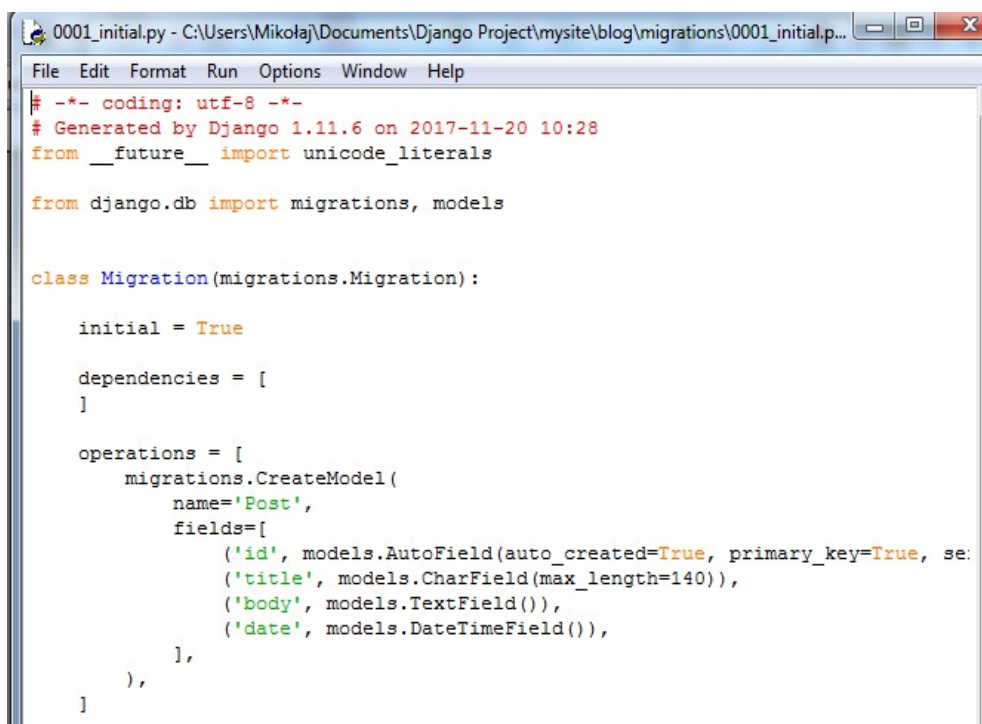
So I guess I've already run the migrate, but hopefully like go ahead and do that, just so you the admin one out of the way.

So the next thing that we're going to do is, so when you make migrations basically there's like two major steps and probably it's always best to do three major steps. So the first major step is going to be to go 'python manage.py makemigrations'. So you make your migrations and you'll see that okay, we made migrations at least for blog, so Django goes through and sees 'there was a change to the 'models.py' since last I knew of it' and it creates this migration and you can see it creates it in '0001\_initial.py Create model Post'.



```
C:\Users\Mikołaj\Documents\Django Project\mysite>python manage.py makemigrations
Migrations for 'blog':
  blog\migrations\0001_initial.py
    - Create model Post
C:\Users\Mikołaj\Documents\Django Project\mysite>_
```

So let's go check that out real quick before we move on. We go into 'blog/migrations' sure enough there's our migration. Let's go and look at it real quick.



```
0001_initial.py - C:\Users\Mikołaj\Documents\Django Project\mysite\blog\migrations\0001_initial.p...
File Edit Format Run Options Window Help
# -*- coding: utf-8 -*-
# Generated by Django 1.11.6 on 2017-11-20 10:28
from __future__ import unicode_literals

from django.db import migrations, models

class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Post',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, se:
                ('title', models.CharField(max_length=140)),
                ('body', models.TextField()),
                ('date', models.DateTimeField()),
            ],
        ),
    ]
```

And we can see that okay, here's what it's doing. It says 'here's our 'Migration' (cursor pointing at class), it's the initial migration, no dependencies, here's the 'operations' it's going to run, it's going to create this model, call it 'Post'. Fields, remember I was telling you before that Django is just automatically going to create that ID for you, sure enough there it is. 'auto\_created', 'primary\_key', maybe it just autoincrements on its own or something, I don't know. Anyway, maybe that's what 'auto\_created' is, I'm not sure somewhere you would expect to see 'auto increment' as well, but maybe Django just knows that or something, I don't know. If someone knows which of these parameters is the 'auto increment' or why, maybe it's 'auto\_created' just doesn't sound right, but maybe because it's the 'primary\_key' it's autoincrementing, because primary key means is going to be unique, but anyway whatever. So that's the '0001\_initial.py', cool that looks good to us. Normally I would not really check the 'initial.py', like you can look at them if you want, if you want to look at your migrations. I want to really look at them the only time you would do it is if you're trying to actually like go back in time, because you'll save all these migrations, so it's like a constant state of your application which is pretty cool. So I'm going to close out of here and so that's the first thing we would do.

The other thing you can do is like if you want to make migrations for like a specific application, you can do something like this `'python manage.py makemigrations blog'`, okay? So you can do something like that, but this is going to now that we've already made that migration before it just says no change was detected, don't worry about it, okay?

```
C:\Users\Mikołaj\Documents\Django Project\mysite>python manage.py makemigrations
blog
No changes detected in app 'blog'
C:\Users\Mikołaj\Documents\Django Project\mysite>
```

So now, in theory, you can run a 'migrate' right now and live changes will occur to your database. But first it might be wise to see maybe what's going to happen when you make the migration. Luckily you can do this, so with SQL there is no like undo button okay?, so you can undo with Django like the schema, but if you go deleting your database, as far as I know right now, there's no like, I mean you can always make backups of course, but you got to be really careful anytime you're changing SQL around.

So what you can do is you can say you can go `'python manage.py sqlmigrate blog'` and then the actual migration ID let's say, so this one is '0001', hit enter:

```
C:\Users\Mikołaj\Documents\Django Project\mysite>python manage.py sqlmigrate blog 0001
BEGIN;
--
-- Create model Post
--
CREATE TABLE "blog_post" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "title" varchar(140) NOT NULL, "body" text NOT NULL, "date" datetime NOT NULL);
COMMIT;
C:\Users\Mikołaj\Documents\Django Project\mysite>
```

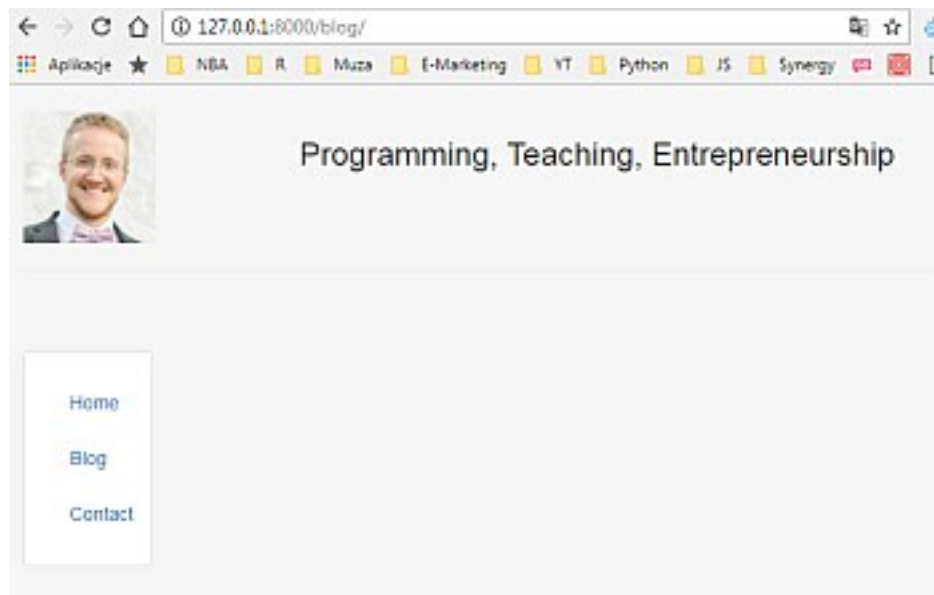
And this is going to give you the SQL that will be run, so this is a 'CREATE TABLE', the table name is 'blog\_post', which is pretty cool, because it just automatically takes 'Post', calls it 'blog\_post' with it anyway that's pretty cool. And ID, that's automatically, yeah see, where did you get that autoincrement?, that's killing me, how does it know?, that's some Django magic. Anyway so it's your 'PRIMARY KEY AUTOINCREMENT', it gets your 'title' which is calling a varchar 'NOT NULL', 'body' is a text data type 'NOT NULL', 'date' datetime. So these data types here, varchar that's a SQL data type, a 'body' text might even be, that's like this almost specific SQLite data type, you rarely would use text, I'm pretty sure in MySQL. And then you've got 'date' datetime, so as you can see this normally you would write this query, but Django just writes it for you based on your models and you might ask 'well, why am I using models then?' and if you're not totally convinced

up to now, when we get into the admin you're going to realize why you do models.

So if everything looks good then you would go ahead and run `python manage.py migrate` ok?

```
C:\Users\Mikołaj\Documents\Django Project\mysite>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying blog.0001_initial... OK
C:\Users\Mikołaj\Documents\Django Project\mysite>_
```

'Apply to all migrations:' good, render them, apply the blog initial migration ok, we don't see any errors. Let's go ahead, run our server, visit our homepage and click on 'Blog':



It works, but nothing is there. Well, clicking on 'Blog' tells us that, just the mere rendering of blog tells us it visited the table, it made the query, no errors, everything happened as it was supposed to. What's missing here is there are no blogs (i.e. blog posts), okay? So how do we begin creating blogs? Well, we could make ourselves like a new blog, a kind of model or a new blog view where we can like type up our blog and stuff like this. And that's what you would have to do in most frameworks let's say, but with Django, Django has a really powerful admin that can access all of your models and you can update your models and change them and make new ones and stuff like this. So that's what we're gonna be talking about in the next tutorial, so we'll be adding some blogs and that would be basically the completion of our blog application, so pretty cool... If you can't like load your blog page, there shouldn't be anything there. If there is stuff there, I'm not sure how that worked out for you, just more Django magic, I don't know. One of these days Django will write your blogs for you.

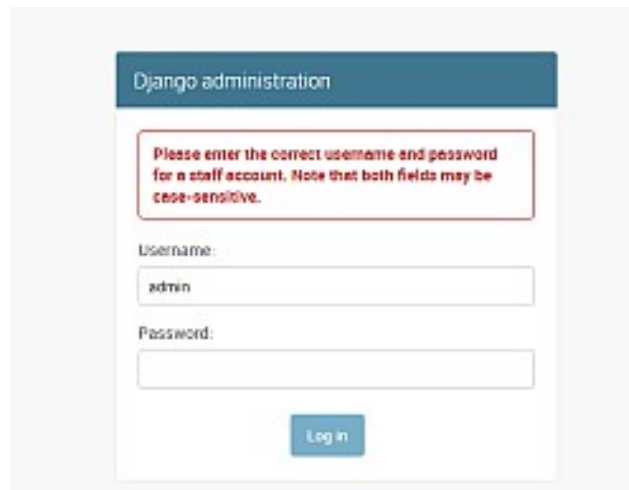
Nice support line at the end.

Comment below the tutorial:

It's auto incrementing because it's an AutoField :)

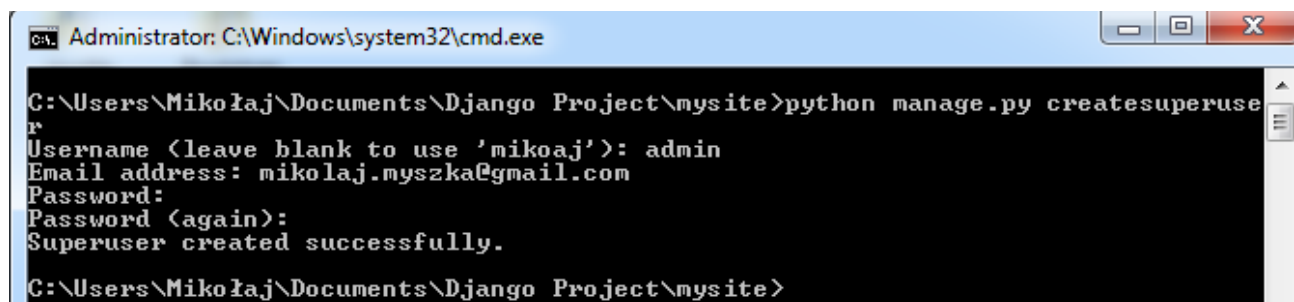
## Admin - Django Web Development with Python 9 – 01.02.2016

In this tutorial what we're gonna be talking about is the admin app that comes with your Django application. So the first thing we need to do when we want to utilize the admin is, as you'll see if we go to our website ok?, and you want to go to admin, you just do '<http://127.0.0.1:8000/admin>' and then you start thinking 'hmm, what do I do now?', type admin, admin?'. No.



The image shows the Django administration login page. It has a blue header with the text 'Django administration'. Below the header is a red-bordered box with the message: 'Please enter the correct username and password for a staff account. Note that both fields may be case-sensitive.' Underneath this box are two input fields: 'Username:' with the value 'admin' and 'Password:' which is empty. At the bottom of the form is a blue 'Log in' button.

So you don't have an admin, you don't know what the login is. So the first thing you have to do is actually create a super user or an admin. So we're going to go and break the running server and say '`python manage.py createsuperuser`'. Now you can define the user. Username, email address, password whatever you want it to be.

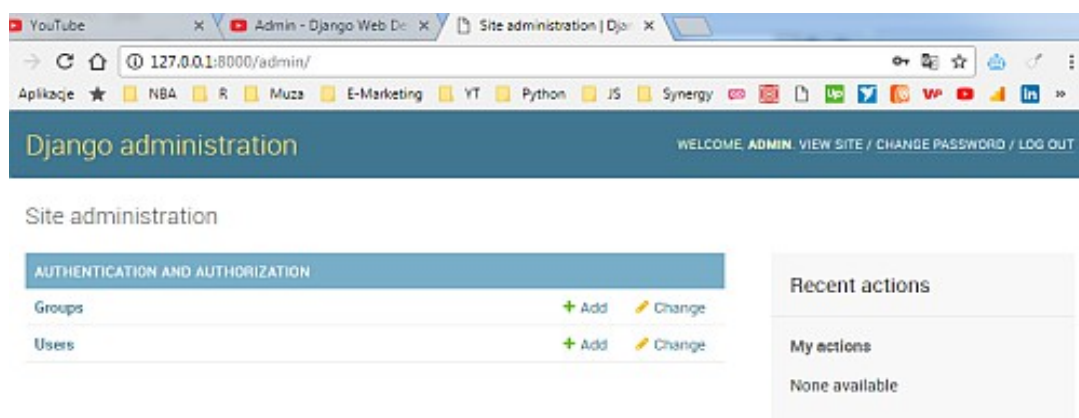


```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\Mikołaj\Documents\Django Project\mysite>python manage.py createsuperuser
Username <leave blank to use 'mikoaj'>: admin
Email address: mikolaj.myszka@gmail.com
Password:
Password <again>:
Superuser created successfully.

C:\Users\Mikołaj\Documents\Django Project\mysite>
```

Okay, I got our user and I need to log in before I forget the password. We need to run the server and login again at '<http://127.0.0.1:8000/admin>'. Ok, so once you've done that, you log in, this is the Django administration.





So you'll see here that you've got 'Groups' and you've got 'Users' in here. And then what you could do is you can click on 'Groups' for example. There we don't have any groups, but you could like click on 'Add Group +' and do fancy stuff here. And you can go into 'Users' and look at that, there we are and we could set you know, are we an active user, if you uncheck this it's basically the equivalent of deleting an account, it doesn't delete the account, but it doesn't let them log in. 'Staff status', 'Superuser status', you can create custom groups and give very specific custom user permissions what can this user actually do, you've got all this stuff bla bla bla (looking at 'Important dates'). Okay, cool, but we don't really care that much about that. What we really want is, well, we want a pony right?

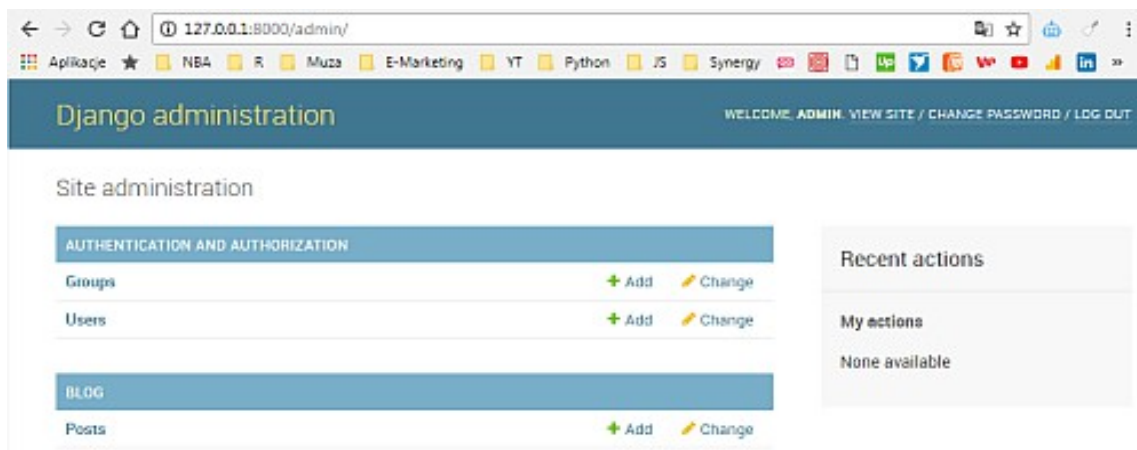


We want to be able to access our blog table basically here and edit the blog, post blogs, do everything by this administration panel, okay?, because we want a pony. So what we're going to do is first we have to admit that we're not going to get a pony, but Django is going to give us something close to a pony. So what we're going to do is we're going to go into 'blog/admin.py' file. Go ahead, open that up and you'll see that we're importing some stuff naturally ('from django.contrib import admin') and then you can '# Register your models here.' So what we're going to do is 'from blog.models import Post', so that's our 'Post' class, which defines that table and all that fun stuff. Now we're going to do is simple 'admin.site.register(Post)' and we register 'Post'.

```
admin.py - C:\Users\Mikołaj\Documents\Django Project\mysite\blog\admin.py (3...
File Edit Format Run Options Window Help
from django.contrib import admin
from blog.models import Post

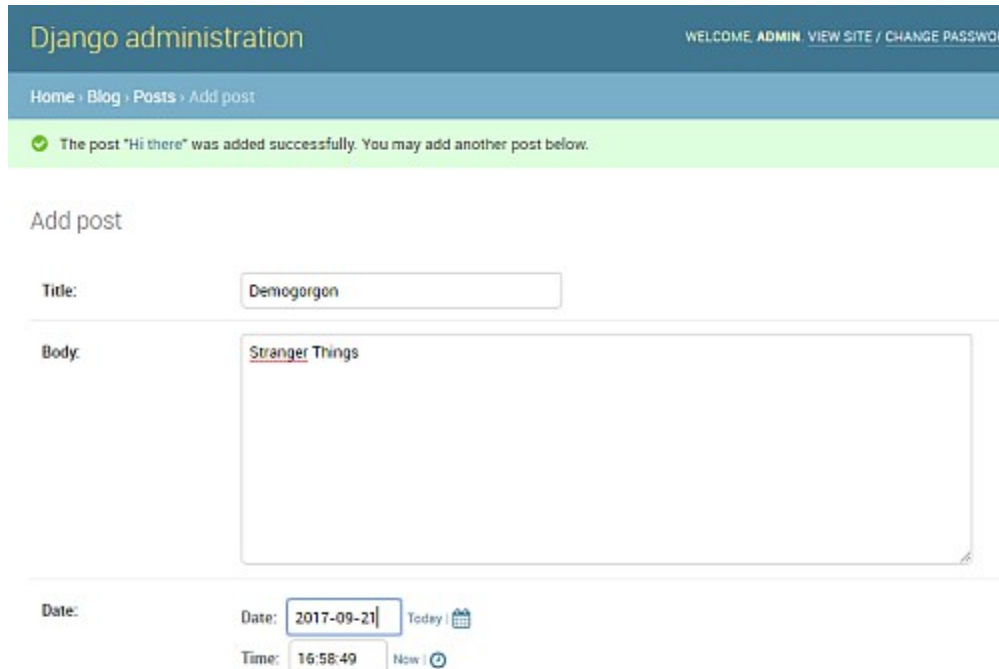
# Register your models here.
admin.site.register(Post)
|
```

Save that, come back to your admin panel, go back to the home, BOOOM!, there is Post:



So now you can see that 'hey we've got Add and Change'. We can also click on 'Posts', there's zero posts now, but then we can also add a post here. So let's go ahead and add a post.

We fill in a title, body and then for date what's cool is you can very quickly just say 'Today', 'Now' and 'Save and add another' let's say. We repeat this operation twice. You can also do like this with date, like you could say this post was made back in September, okay? 'Save'.



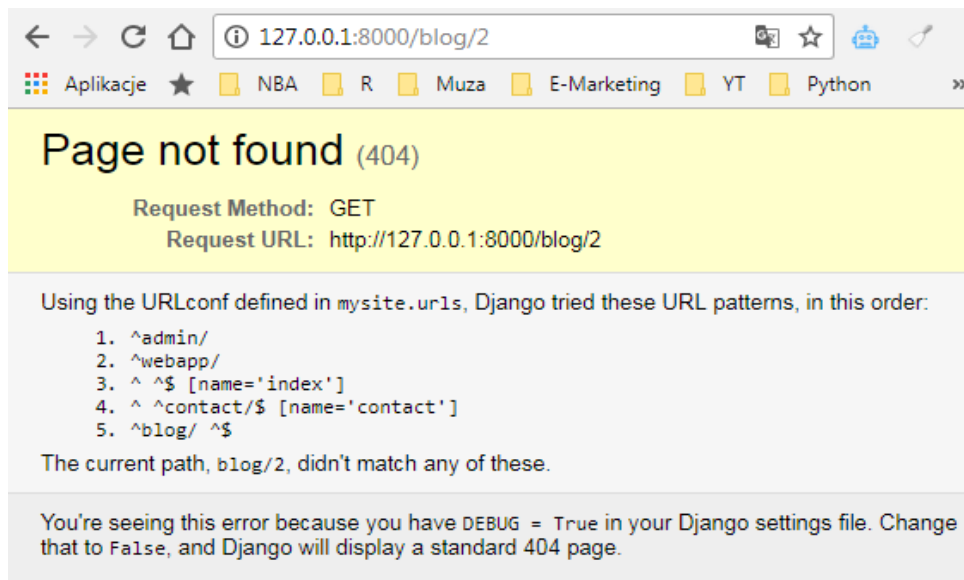
The screenshot shows the Django administration interface. At the top, there's a blue header with 'Django administration' on the left and 'WELCOME, ADMIN | [VIEW SITE](#) / [CHANGE PASSWORD](#)' on the right. Below the header is a breadcrumb trail: 'Home > [Blog](#) > [Posts](#) > Add post'. A green success message banner reads: '✓ The post "Hi there" was added successfully. You may add another post below.' Below this is the 'Add post' form. It has a 'Title:' field with the value 'Demogorgon'. The 'Body:' field is a large text area containing the text 'Stranger Things'. At the bottom, there are 'Date:' and 'Time:' fields. The 'Date:' field is set to '2017-09-21' with a calendar icon and a 'Today' button. The 'Time:' field is set to '16:58:49' with a clock icon and a 'Now' button.

Alright, so now these are our 'Posts'. We can edit them, delete them etc. Cool. So now, what if we just go to our homepage and we click on 'Blog':



There's our information right? It's just there and sure enough the titles are, links and in all our excitement we click on one post link and ahh:





It doesn't work. Why doesn't it work?, you ask. Well, of course we don't have any url path that explains or works with '/blog/2'. We can even see, well this is your '/blog/' (url pattern 5.), but this is the homepage, it required it to be '/blog/' not '/blog/2'.

So now what we have to do is create the handling, we need the urls, we need the view for the individual posts. So that's what we're gonna be covering in the next tutorial. This one should be quick and easy, quick and painless, easy enough for you guys to get up to this point, but if you had any problems, feel free to leave them below. One thing to keep in mind if this is the first time you visisted the admin, which if you're following along it probably was, if it didn't work for whatever reason, you can always go to 'mysite/settings.py' and make sure admin is installed. Make sure you have this line ('django.contrib.admin'):

```
# Application definition

INSTALLED_APPS = [
    'webapp',
    'personal',
    'blog',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

It should be there by default, but if it's not there for whatever reason, you might need to add that part, okay?

## Individual blog pages - Django Web Development with Python 10 – 08.02.2016

In this tutorial we're going to be continuing to work on our blog. If we go to our homepage we find that everything works. We click on 'Blog', that works. We click on a blog post, that does not work and that's not really aligned with the fact that we want a pony. So how do we make our blog return the specific blog post that we've clicked on? Well, what we're going to first do is we know that, okay the first problem is using the first 'mysite.urls', we did go to 'blog', but this does not correspond with this '^\$' (in '5. ^blog/ ^\$'). This is start and end, so what we need to do is we need to handle for a '/' and then a number. So what we're going to do first is go into 'blog/urls.py' and add another url.

Now we define a new url, this url is going to look a little scary at first, but I promise it's going to all come together. So first we're going to write it, then we'll explain it. Actually we might explain it as we go, we'll see. Anyway, what we're going to do first is url, the url is going to be this expression `'url(r'^(?P<pk>\d+)$')`.

Parentheses say 'hey we're capturing this everything here inside these parentheses, '?P' is a named capturing group, let's pull up the Django here:

<https://docs.djangoproject.com/en/1.11/topics/http/urls/#named-groups>

Basically what we're doing there, the reason why we want to be able to do that is because we're going to grab specifically by each post ID and then also we're grabbing right in between these parentheses very specific..., you'll understand here in a moment. Just understand '?P' corresponds to named groups. The next thing is in these brackets '<>' we put 'pk'. 'pk' stands for primary key and as you've seen already the primary key is that ID column, so that's the first column in our table basically and every post gets a unique primary key, this is like starts at zero, it's autoincrementing, so it's like 0, 1, 2,... all the way up to how many posts you have. So that primary key is quite specifically is this right here ('/blog/1' in the website address) right? We're trying to get to the blog information that is of the primary key of one. So the primary key there and then that primary key is a backslash d '\d' which corresponds to being a digit in regular expressions and then we use a plus '+'. Now, if you're not familiar with regular expressions, like I said before, you can go to:

<https://pythonprogramming.net/regular-expressions-regex-tutorial-python-3/>

'+' matches one or more, so when we have a '\d+' that means it's a digit and it's either one or more digits, okay? So one to a billion will be acceptable here and beyond a billion of course. So then that's what we're capturing here basically and then end '\$'.

So basically what this is saying here is the url begins and again of course that's beginning after '/blog' right?, because what's happening here is '/blog' leads us to this 'urls.py' and then from here is what we define. So '/blog/' and then you've got this information here ('url(r'^(?P<pk>\d+)\$')') and then it ends. Well, this information here as long as it is, is just saying 'hey the primary key which happens to be a digit, you don't actually have to have digit there, but you want to do that, because you know the primary key is supposed to be a digit and if users are trying to get some LOLs, they might try to do hacky things on your website and one of those things might be instead of having a primary key maybe put a string, maybe put an SQL query in there just to see, because what's happening is it's going to query the database for that primary key, so then you know a wannabe hacker might try to play games. So we're going to say this has to be a digit, otherwise don't return anything. And note that that would have worked, just saying Django is a little smarter than that, but just keep that in mind.

So now, instead of 'ListView', this is a 'DetailView', so 'DetailView.as\_view' and then the model we're going to have for this 'DetailView' is 'Post', just that 'Post' class from the models and then 'template\_name' is going to be equal to 'blog/post.html'. So the model already exists, we created that long ago in a previous decade and then the 'template\_name' obviously this doesn't exist, we need to create that. So that's what we're going to do now, so let's go ahead and save this.

```
urls.py - C:\Users\Mikolaj\Documents\Django Project\mysite\blog\urls.py (3.6.3)
File Edit Format Run Options Window Help
from django.conf.urls import url, include
from django.views.generic import ListView, DetailView
from blog.models import Post

urlpatterns = [url(r'^$', ListView.as_view(queryset=Post.objects.all().order_by("-date")[:25],
                                           template_name="blog/blog.html")),

              url(r'^(?P<pk>\d+)$', DetailView.as_view(model=Post,
                                                       template_name="blog/post.html"))

              ]
```

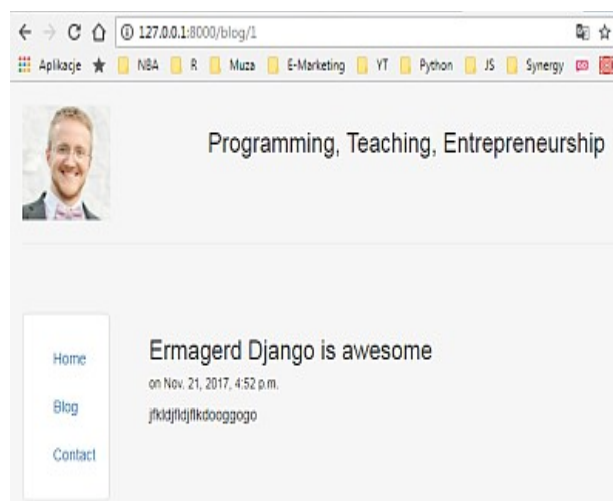
Come into 'templates/blog' and now just copy/paste 'blog.html', rename it to 'post.html' and edit. Okay this will indeed extend 'personal/header.html'. It's going to have 'block content' and 'endblock'. So this is individual blog page, so how do we want that individual blog page to look? You can obviously do a whole lot of things. We're going to say the title will be in header three tags. Then when it was published (date in header six) and then the body.

All these posts, the posts that are coming from us right?, this is coming from our admin page, so for admin page is compromised we got probs. So we're gonna say a filter 'safe'. Basically what this does is it allows us to put HTML into the Jinja, so if you didn't have this and in your post you had paragraph tags and strong to make things bold, and stuff like that, if you didn't put '|safe', it would come through as plain text tags. If that doesn't make any sense to you, if I remember I will show you what I mean there. And then again we will do '|linebreaks', so everytime there's a line break, there will be a line break, okay? That's our post body, no problem and that should be everything we need really, let's just tab this over (all inside the 'block content' and 'endblock'), save and we should be ready to rumble.

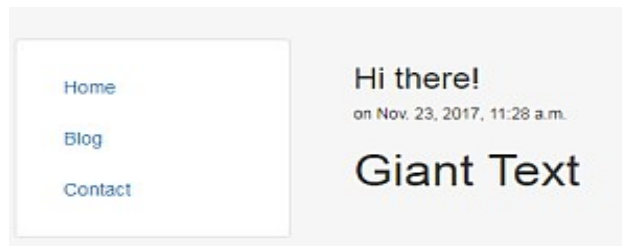
```
{% extends "personal/header.html" %}

{% block content %}
    <h3>{{ post.title }}</h3>
    <h6>on {{ post.date }}</h6>
    {{ post.body|safe|linebreaks }}
{% endblock %}
```

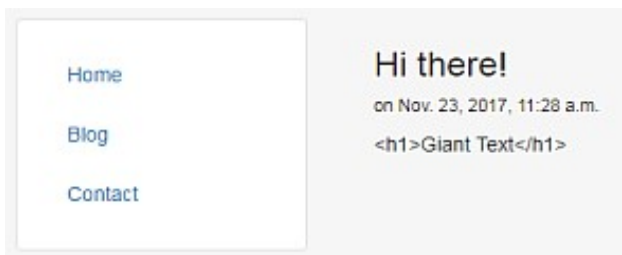
Let's come over here (server console), everything looks good there, let's bring up our website here, refresh and here's our blog:



Ok, so now let us head to our admin page and let's add a new blog (post). In body let's put something in header one tags, like '<h1>Giant Text</h1>'. Save this and go back to the 'home/blog' page and then to our just created post with 'Giant Text'. There it is:



What happens if we did not say this text was 'safe?', so we change '{{ post.body|safe|linebreaks }}' to '{{ post.body|linebreaks }}' and save.



We get the plain text values and if you view the source, you'll see that they were changed here:

```
57 <h3>Hi there!</h3>
58 <h6>on Nov. 23, 2017, 11:28 a.m.</h6>
59 <p>&lt;h1&gt;Giant Text&lt;/h1&gt;</p>
```

You'll see that you get these little the converter basically, so it's not the actual tag. So that's what 'safe' does. There are a lot of other filters and stuff like that with Jinja and we can cover those at a later tutorial, but just know that's what you can do. So now when you go to post your blog (post), not only can you use like header tags, you do like image tags, you can do scripts, you can do all kinds of stuff and because you can do things like scripts, that's why if say you've got a forum and you're allowing users to post, you probably don't want to say that text is 'safe', because they are going to be putting all kinds of stuff in there. So that's kind of why by default Django is not going to treat that as HTML, they are going to do converts and escape things, and stuff like that in your favor, but since it's coming from the admin page right?, your posts are all coming from admin, we're going to go ahead and say 'hey, that's some safe information, okay?'. So at this point you now have your homepage or you know, really your about me page, you can put some information here ('Home'), you got a contact information and you've got even a blog that you can add things to. Of course your blog is not pretty, it's not the best looking blog in the world, so I highly suggest you would spice this up a little bit and you're really your whole personal website. It's not very attractive yet. Also the links standing here (LinkedIn, Twitter, etc. at the bottom) obviously don't really work, they just lead to nothing, but you could populate those. So I suggest you spend some time on Bootstrap, especially if you do end up, you know, pushing this live and all that.