

Django Tutorial for Beginners - 1 - Installing Django

nice commands from cmd

2. Creating a Project

You can create a New Project directly from PyCharm 2.28

manage.py – never edit this file, program that comes with Django, that let's you do a bunch of cool stuff to your project and it allows you to do things like access the database and create users for your website

__init__.py hey, Python treat this folder/directory as a package

setting.py – settings and configuration options for your website, so like the overall settings for your entire website

url.py table of contents for your websites, website name and then is some url, look at whatever url the user requested and then perform some functionality and there's a lot more to it than that, but this is basics

wsgi.py – webserver gateway interface it's just a special type of webserver, we're going to learn about that a lot later, don't worry about it right now

Django comes with mini webserver that you can use for development

3. Creating Our First App

PyCharm top right green arrow button runs the server

Ctrl + C stops your development server

> clear – command to clear cmd

Cool thing about Django is whenever you're working with your source code and you're adding new stuff, you don't need to reboot your server all the time.

Basically your website in Django is going to be composed of all of these little parts called apps, like popular section, forum, video player section, any like main part of your website, individual different parts.

The rule: each app should do one thing. If you can't explain what it does in one sentence, then you need to break it up more.

> ls – list of files within a directory

.idea file is for PyCharm

4. Overview of a Basic App

migrations.py – basically a way you can hook up or connect your website all of your source code with your database

admin.py

apps.py – configuration file or settings for this app, right now we don't have a lot of settings, it says the name of it is 'music'

models.py – basically a blueprint for your database, so it's a template how you're going to store data for this app

tests.py – you can create a bunch of little tests to make sure that you don't have any bugs in your code

views.py – easy, they are just Python functions and what they do is they take a user request and they give them back something/a response. 95% of the time users are going to request a webpage. Other times it isn't that direct like maybe they just want to log out or download something, so you would perform some action.

5. Views

urls.py so in here what you have is something called **url mappings**.

URL: <http://127.0.0.1:8000/admin/>

So basically what it does is this **part** of the url, it doesn't really matter, this is just the address of your server, just the IP address or domain name. This whenever I talk about urls, **this** is the actual part that matters. Django is going to ignore the first part and look at the second part. This is the thing that the user is requesting.

Url patterns match the user request

r in the url patterns is not a regular expression, it's raw string.

cluttered

We need to think ahead and think about the design. This is how actually you should set it up.

from . import views

from current directory ('.') import views. So from the same directory look for a Python file or module ('views').

In urls.py we need to import one more package which is 'include'. This is just a package that lets us include other files, which is exactly what we're trying to do.

So again, like I said, all of view does is **it takes the request and it sends back an HTTP response.**

There is a bunch of different responses you can send, but this is the most simple:

'from django.http import HttpResponse'

6. Database Setup

Django comes with default database SQLite. By default the DATABASES (in 'settings.py') is set equal to a 'sqlite3' engine.

Django can use basically any database, it can use Postgre or MySQL and those kinds of databases are something that you want to use in production, but just for testing it's really awesome that it came ready to use this (sqlite), because we don't got to worry about all the hard stuff of setting it up.

So this database is good to go, however the way it is set up right now we basically have all of our code, all of our apps and then separately we have this database, so they are not connected at all, we

still need to configure it to work with our code.

Basically behind the scenes Django is ready to work with a bunch of (already installed) apps. So these apps right here, some of them actually need to access the database.

'You have unapplied migrations' message in cmd - your source code, your app is not in sync with your database.

'python manage.py migrate' – it's going to sync up your code with your database. I say sync it up, but it's more technical than that. Exactly what's happening is whenever you type migrate, it's going to go into your 'website/settings.py', it's going to scroll down and look for installed apps. Now for each one it's going to go in that apps directory and look for what tables are needed to pretty much work with this app. So each app has its own tables that it requires and for example, whenever we're working on our 'music' app later, we're going to need a table for albums, another one for songs, so on and so forth.

7. Creating Models

Model is just a blueprint how you want to store your data.

What kinds of data are we going to be saving? Albums and songs, so we need to build a blueprint, for example an album needs a title, genre, artist, etc.

Every model or blueprint you create it has to inherit from 'models.model'.

We're just writing a regular Python class, so we can just work with it in Python, but later on whenever we migrate it over its database, this is actually going to be the same name as a column, so we don't have to write it twice, Django takes care of all of that for us pretty sweet.

We need to tell Django what kind of data this variable (field) is going to hold, text, float, integer.

'artist = models.CharField(max_length=250)'

One thing we need to specify whenever we are storing characters or text is 'max_length'.

2 classes: Album and Song

Whenever we create a song, it needs to be associated with an album, in other words a song needs to be a part of an album. How do we link these together?

Behind the scenes Django is going to make another column for us and this is a unique ID number (primary key).

'album = models.ForeignKey(Album, on_delete=models.CASCADE)'

Whenever we delete an album, any songs that were linked to that (or part of that) album, then go ahead and delete those as well.

8. Activating Models

Whenever we make an app, right after we make the template for it, we need to go to 'website/settings.py' and put it in INSTALLED_APPS. So that way whenever we boot up our server, it's going to go and say 'ok, let me go ahead and check out your installed apps and make sure they are all working with the database correctly'.

In `INSTALLED_APPS` we write the package name or the app name ('music').

Whenever you start your server it goes and looks right here (at `INSTALLED_APPS`) and for each app that is installed it's going to go look at those models and check to see that these models match the database (to jest clue). And right now they don't, because even though we defined our blueprint, we didn't reflect that in our database structure.

`'python manage.py makemigrations music'` – we're just telling Django 'hey, we made some changes to the music model, in this case we added 2 classes. After executing this command Django says 'ok, I made these migrations. In other words I made these changes right here (shown in cmd).

This step is for preview:

`'python manage.py sqlmigrate music 0001'` – all the migration is basically taking whatever changes you have and it converts it to an SQL file, so you see what it's doing, in this case is creating a table for albums, for songs and a bunch of columns. So again, migration is just a change to your database, that's it.

So now we have our SQL file or change file created, now all we have to do is just run it, like you're running your an SQL file.

`'python manage.py migrate'` – like we did in the last tutorial.

So again, the first command makes the change file and this command actually runs it. So now our database is synced with this code (in 'models.py').

9. Database API

`'python manage.py shell'` – starting Django database API shell

We basically have 2 tables, actually we have a bunch of them that Django created himself, but we created 2.

Our tables actually have more columns than the ones we created, but we'll talk later about it.

Database commands:

`'from music.models import Album, Song'` - anytime we want to use our models we need to import them just like you do in regular Python code.

Let's start performing some functions on our models:

`'Album.objects.all()'` - I want to see all of the items that are in the Album table.

Variable called 'a' and set this equal to a new Album object, this is just a regular Python object. The Album class has 4 attributes: 'artist', 'album_title', etc.

`'a = Album(artist="Taylor Swift", album_title="Red", genre="Country", album_logo="https://upload.wikimedia.org/wikipedia/en/thumb/e/e8/Taylor_Swift_-_Red.png/220px-Taylor_Swift_-_Red.png")'`

So now we have this object in memory, now we need to **save this object to the database**, because right now it's just in shell's memory:

`'a.save()'`

```
'a.artist'  
'a.album_title'
```

Some people call it an 'id' or a 'unique id' or 'primary key':

```
'a.id' is the same as 'a.pk'
```

Now another way to add an Album to our table:

```
'b = Album()' - create an empty album  
'b.artist = "Myth"  
'b.album_title = "High School"  
'b.genre = "Punk"  
'b.album_logo = "https://upload.wikimedia.org/wikipedia/en/4/46/The\_Offspring -  
The\_Offspring.jpg"  
'b.save()'
```

Whenever you create an object you can do so in two ways: pass everything in through the constructor or create a blank object without any attributes and then add them all manually.

We can change the album title from "High School" to "Middle School" by doing:

```
'b.album_title = "Middle School"'
```

and it will be updated (renamed).

10. Filtering Database Results

```
'Album.objects.all()' - now we have 2 albums or 2 objects in our database
```

Go to 'music/models.py' and inside the Album class. We make a special method named 'dunder string':

```
'def __str__(self):  
    return self.album_title + ' - ' + self.artist'
```

This method is a built-in syntax that means a string representation of this object.

dunder = double underscore

We need to restart the shell, because we've made changes to our models and shell remembers the old structure.

'cls' – clear cmd

You can look up specific albums and you can do that in a couple of different ways:

```
'Album.objects.filter(id=1)' - get the albums with the ID of one
```

```
'Album.objects.filter(artist__startswith="Taylor")'
```

The reason I'm teaching you guys this is because if you ever want to add objects or delete them from your database this is a really convenient way, **also the code that we wrote (in shell with filters), this isn't a specific to your Python data API shell, this is the exact code that we're going to be writing in here (in 'models.py').** So it's the same thing, it's just a lot of faster and easier to play around with it when you get your results instantly and you don't have to refresh any page over and over again.

11. Admin Interface

But I need to vent :)

'python manage.py createsuperuser' – creating admin

Of course whenever you making it in a production environment you would actually fill out something that's real (username, pass, email). In production environment this should be something a lot more safer and secure other then admin, pass1234.

Login to admin

So this is pretty much an administration of the backend, pretty much access to the database.

Showing Albums or Songs in Django administration or whatever tables you want to be able to manage from your admin area.

In 'admin.py':

'from .models import Album'

'admin.site.register(Album)' – we tell Django that Album class should have an admin interface, we register this Album class as an admin site.

Refresh admin page, voila!

Breadcrumbs at the top so you can navigate around easily.

There is a lot of stuff you can do.

12.

View is just a simple function that returns some HTML. Each url pattern is connected to a view. To simplify that even more, each url is connected to an HTML response. Obviously each webpage is connected to some HTML that needs to be displayed whenever you go to that.

We design 2 types of pages in Music section: first one is overview of all Albums and the second one is detail view (of each Album).

We make detail view structure first.

urls.py:

/music/712/ <---- pattern that we want to match in comment

url(r'^(?P<dupa>[0-9]+)/\$', views.detail, name='detail'),

After ?P<dupa> we can finally start matching this pattern. In parenthesis () we are grouping.

extracted integer (712) will be saved in dupa variable and then we can pass it to our function (detail) below in views.py:

def detail(request, dupa):

return HttpResponse("<h2>Details for Album ID:" + str(dupa) + "</h2>")

Every single view function is going to take the request. The request is just the HTML request.

Whenever they connect to your site and request something like a webpage or an image or whatever.

And there is a lot of background information about this like their IP address and yada yada, we really don't look at too much, but that's what it is.

'?P<dupa>' - what this bit of code does is it allows us to treat 'dupa' ('album_id' in original) just like a variable, so we can just pass it in.

Make sure I follow PEP8 and have a blank line at the end.

This function is called 'detail', so whenever we want to hook up a url pattern to it, we just need to say 'views', because that's the name of the module and '.detail'. And you don't add the parenthesis '()' at the end of '.detail', because it knows all the variables from these groups ('?P<dupa>') already.

'name='detail' - add a name and just name it whatever the function is. Whenever you have a name it allows you to do some really cool things in your HTML later on.

Right now we can go to any number in the browser url, such as 74359743, because we are not actually connecting to the database yet and validating that it is a valid album ID and we actually have something stored for it.

Summary: it's actually really simple, just look for whatever url the user typed in and connect it to one of these functions right in there.

13. Connecting to the Database

Aim: For the music homepage we're going to connect to a database, pull out all the albums that we have and display them. Once we have all the albums listed, the user is going to be able to click one and it's going to take them to that view that we made in the last video, a more detailed view of that single album.

views.py:

```
#from django.shortcuts import render
from django.http import HttpResponse
from .models import Album

def index(request):
    all_albums = Album.objects.all() <-- connecting to database and getting all of the albums,
    same commands as in Database API
    html = " <--- as variable
    for album in all_albums:
        url = '/music/' + str(album.id) + '/'
        html += '<a href="' + url + ">' + album.album_title + '</a><br>'
    return HttpResponse(html) <-- not indented!
```

```
def detail(request, dupa):
    return HttpResponse("<h2>Details for Album ID:" + str(dupa) + "</h2>")
```

If you work for a company in the proper structure to set up a Django project, you need to take your HTML and separate it away from your Python logic. So ideally you should have very, very little if none HTML code in your Python views.

14. Templates

Templates (eg. shopping templates, blog templates) are generic HTML documents with a bunch of dummy data in it.

We use templates in Django, because they allow us to pretty much separate our HTML from our backend stuff.

Anytime you want to work with templates in Django just write:

```
'from django.template import loader'
```

we're going to make this template a separate file and we basically need to load it in.

You can change the name of 'templates' in your settings, but this is kind of standard convention.

Inside the 'templates' you create a subdirectory with the same name as your app so 'music' (another standard convention).

```
'/music/views.py':
```

```
#from django.shortcuts import render
from django.http import HttpResponseRedirect
from django.template import loader
from .models import Album
```

```
def index(request):
```

```
    all_albums = Album.objects.all()
    template = loader.get_template('music/index.html') <---1)
    context = {'all_albums': all_albums} <---2)
    return HttpResponseRedirect(template.render(context, request)) <--- !!
```

```
def detail(request, dupa):
```

```
    return HttpResponseRedirect("<h2>Details for Album ID:" + str(dupa) + "</h2>")
```

1) By default Django is already set up to look in directory called 'templates' in your apps directory.

2) Whenever we pass this album information to our template, we pass it through a dictionary and most people name this 'context', you can actually name it anything you want, but it's kind of the standard. It basically means information that your template needs.

Of course, rules don't change, whenever you have a view, you need to return something (in this case our finalized 'index.html' template).

```
'/templates/music/index.html':
```

```
{% if all_albums %} <--- 1)
<h3>Here are all my albums</h3>
<ul>
    {% for album in all_albums %}
        <li><a href="/music/{{ album.id }}">{{ album.album_title }}</a></li> <--- 2) 3)
    {% endfor %}
</ul>
{% else %}
```



```
<h3>There is no albums so far</h3>
{% endif %}
```

- 1) This is going to be true as long as it (all_albums) has at least one item (album).
- 2) This '/' is important here
- 3) Variables are in '{{ }}' brackets.

15. Render Template Shortcut

In previous tut we loaded a template, ran it through Django and converted it all into regular plain old HTML and then once we had all that rendered into plain HTML we can return it to the user.

Way (shortcut) to bypass or combine the load and render functions in 'views.py':

```
from django.shortcuts import render <--- new import
from django.http import HttpResponse
#from django.template import loader <--- deleted import
from .models import Album
```

```
def index(request):
    all_albums = Album.objects.all()
    context = {'all_albums': all_albums}
    return render(request, 'music/index.html', context)
```

```
def detail(request, dupa):
    return HttpResponse("<h2>Details for Album ID:" + str(dupa) + "</h2>")
```

'Wait a sec, I thought that you needed to return an actual HTTP response Bucky?'. Well, that's actually built-in to this 'render' function, so behind the scenes it takes this (request), converts it to an actual valid HTTP response and it's good to go.

16. Raising a 404 HTTP Error

We don't have an album with ID of 43, yet our webpage is displaying this '/music/43/'. 404 – we don't have the resources to display or it got deleted or something like that.

views.py:

```
from django.shortcuts import render
#from django.http import HttpResponse
#from django.template import loader
from django.http import Http404
from .models import Album
```

```
def index(request):
    all_albums = Album.objects.all()
    return render(request, 'music/index.html', {'all_albums': all_albums}) <--- used once so can be
```

presented like that, instead of 'context'

```
def detail(request, dupa):  
    try:  
        album = Album.objects.get(pk=dupa) #dupa to album_id  
    except Album.DoesNotExist:  
        raise Http404('Album does not exist')  
    return render(request, 'music/detail.html', {'album': album}) <-- used once so can be presented  
like that, instead of 'context'
```

'try except' statement

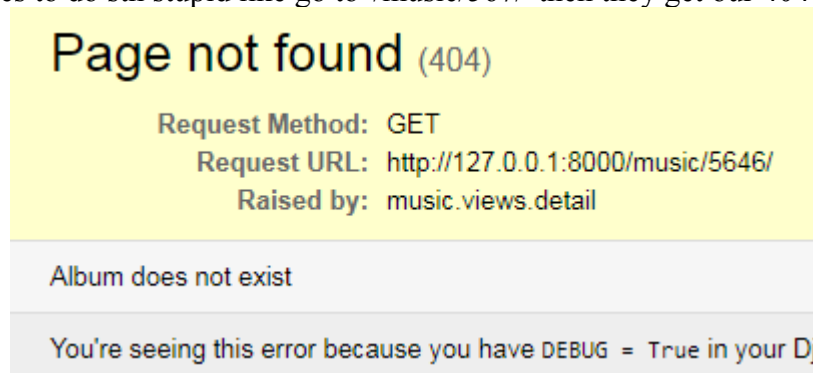
get

raise

DoesNotExist

As long as the ID is valid (so in the database) this is going to work perfectly, however if they type sth like 700 (which is not in the db) and we can't get that item, then it's going to generate 'Album does not exist' error.

If the user ever tries to do sth stupid like go to '/music/567/' then they get our 404 response:



If you want to give them a little clearer indicator of why they are getting this, again this is optional, but this is always a good idea to do.

'music/detail.html':

```
{{ album }}
```

Corey Schafer yt channel

17. Adding Songs to our Database

Right now we have a couple of albums in there, but we don't have any songs. More importantly, whenever we add a song, it has to be associated with an album.

String representation of the Song class (('def __str__(self):...'), sth like what we did in the Album class).

You only need to sync with the db (migrate), if you're adding or deleting attributes, because these represent columns in your tables. Whenever we make this little change ('def __str__(self):...'), we don't add or delete any columns, so it doesn't change the structure of our table.

Registering Song in the 'admin.py'.

Each song has an attribute that points to an album ('album').

```
>>> from music.models import Album, Song
>>> album1 = Album.objects.get(pk=1) = getting a reference to a (Taylor Swift) album
>>> album1.artist – just checking
'Taylor Swift'
>>> song = Song() - blank Song object
>>> song.album = album1 <-- interesting stuff
>>> song.file_type = 'mp3'
>>> song.song_title = 'I hate my boyfriend'
>>> song.save()
```

18. Related Objects Set

Another way to add songs to an album is a much quicker and cooler way in my opinion. If you take a look at our model, what Django already knows is that this 'Song' class is associated with this 'Album' class using a foreign key. In other words, because we related these two object in this kind of way it already knows that this album is going to have a whole set of songs associated with it. So what Django did it allows you to access the songs through a set.

If you just reference any album, that object it has a set, named the class name underscore 'set', but the class name is actually lowercase.

```
>>> album1.song_set – that's how you access this album's songs
```

```
>>> album1.song_set.all() - if you want to display them all just call the function 'all()'
<QuerySet [<Song: I hate my boyfriend>]>
```

Putting object in an album set at once (already with saving).

We need to pass in every attribute except this one ('album' attribute). Why? It already knows, because we referenced 'album1' that whenever we create a song it obviously got to belong to 'album1'.

Reference your album object, reference your set, call a function called 'create'.

```
>>> album1.song_set.create(song_title='I love bacon', file_type='mp3')
<Song: I love bacon>
```

You can refresh Django administration and bum, this bad boy is right there. Roasted!

One of the last things I want to point out is whenever you use sth like this create method:

```
>>>> album1.song_set.create(song_title='Ice cream', file_type='mp3')
it does all those things that I told you, but it also returns a reference to that song. So if you ever
need to take that object and use it in your program, what you can do is set it equal to a variable:
song = album1.song_set.create(song_title='Ice cream', file_type='mp3')
>>> song
<Song: Ice cream>
>>> song.album
<Album: Red - Taylor Swift>
>>> song.song_title
'Ice cream'
```

```
>>> album1.song_set.all() - look at all of this album songs
<QuerySet [<Song: I hate my boyfriend>, <Song: I love bacon>, <Song: Bucky is lucky>, <Song:
Ice cream>, <Song: Ice cream>, <Song: Golonka>]>
```

```
>>> album1.song_set.count() - how many objects or songs there are
6
```

Bill Gates yt movie about coding

19. Designing the Details Template

On the details page we want to have the album logo and under it album title in bold, and then list of the songs.

Let's design a blueprint what we want first:

```


<h1>Album title</h1>
<h2>Artist</h2>

<ul>
  <li>Song title - song file type</li>
```

And now how it should look like with all the variables:

```
 <--- "" around url, works without "" as well

<h1>{{ album.album_title }}</h1>
<h2>{{ album.artist }}</h2>

<ul>
  {% for song in album.song_set.all %} <--- without ()
    <li>{{ song.song_title }} - {{ song.file_type }}</li>
  {% endfor %}
</ul>
```

And the result:



Comments:

- 1) I don't get one thing, how album.song_set.all does know that it need to find all songs?

That confused me too at first. Read this:

https://docs.djangoproject.com/en/1.11/topics/db/examples/many_to_one/

It'll explain it.

- 2) Why is that even though the class name Album starts with capital letter, when he wrote in the html file, he wrote album.album_title instead of Album.album_title??

album with the lower case a is a specific instance. Album with uppercase A is the type.

You use album with a lowercase (as opposed to some other word) because that is what we defined it to be in music.views.detail

20. Removing Hardcoded URLs

Part of 'music/index.html':

```
<ul>
    {% for album in all_albums %}
        <li><a href="/music/{{ album.id }}">{{ album.album_title }}</a></li>
    {% endfor %}
</ul>
```

This part is **dynamic**. 'album.id' always changes.

This is not dynamic, it's **hardcoded** url. **It's not good to have it in your template (especially when you have a huge website with a lot of templates, avoid it).**

We want to make a hardcoded url dynamic.
(btw 'href' in 'a href' is an attribute, '{' is)

Blueprint:

```
<a href="{% url 'name' variable %}">
```

'name' is from 'name=' in url patterns in 'urls.py'

variable – what number do we want to add in there ('album.id')

New part of 'music/index.html':

```
<ul>
    {% for album in all_albums %}
        <li><a href="{% url 'detail' album.id %}">{{ album.album_title }}</a></li>
    {% endfor %}
</ul>
```

So whenever we use this non-hardcoded url it says 'we are using this pattern ('detail') and for an 'album.id' we're going to plug in whatever number this is, which is the actual album id'.

21. Namespace and HTTP 404 [*comment second part]

Now let's take a step back and think about things for just a second. Right now our project is small, it

only has one app 'music'. But what happens when we start creating more apps? Maybe we're going to make another one for videos, for profile forums, whatever. So basically what I'm trying to say is usually sites have like 10 or 20 or even more apps. So that's fine, however what happens if another app aside from 'music' has the same 'detail' view? Maybe it has 'video' 'detail' view, 'profile' 'detail' view. Well then whenever Django tries to look at this template ('index.html'), it says 'all right, you're telling me to get the url pattern for 'detail', but I have one in 'music', one in 'video', what the heck url pattern am I supposed to know what you're talking about?'

Well, we solve that problem by namespacing our urls. That means **we need to specify to which app ('music') this namespace ('detail') actually relates to.**

In other words, there might be situation that we have many apps and many url patterns with name='detail'. So to identify proper url pattern we add app_name = 'music' to '/music/urls.py' and 'music:' in 'music/index.html'.

'/music/urls.py':

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

```
app_name = 'music' <-- new part
```

```
urlpatterns = [
```

```
    # /music/
```

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

```
    # /music/712/
```

```
    url(r'^(?P<dupa>[0-9]+)/$', views.detail, name='detail'), #dupa to album_id
```

```
]
```

Part of the 'music/index.html' template:

```
<li><a href="{% url 'music:detail' album.id %}">{{ album.album_title }}</a></li>
```

Now we can have this keyword ('detail') in however many apps we want.

OTHER UNRELATED TOPIC, 404

'music/views.py':

```
from django.shortcuts import render, get_object_or_404 <-- new function
```

```
#from django.http import HttpResponse <-- old stuff
```

```
#from django.template import loader <-- old stuff
```

```
from django.http import Http404 <-- not needed anymore, can be deleted
```

```
from .models import Album
```

```
def index(request):
```

```
    all_albums = Album.objects.all()
```

```
    return render(request, 'music/index.html', {'all_albums': all_albums})
```

```
def detail(request, dupa):
```

```
    """
```

```

try:
    album = Album.objects.get(pk=dupa) #dupa to album_id
except Album.DoesNotExist:
    raise Http404('Album does not exist') <--- this whole commented part is replaced by...
    """
album = get_object_or_404(Album, pk=dupa) <--- ... this one in one line
return render(request, 'music/detail.html', {'album': album})

```

'music/views.py' after deleting redundant parts:

```

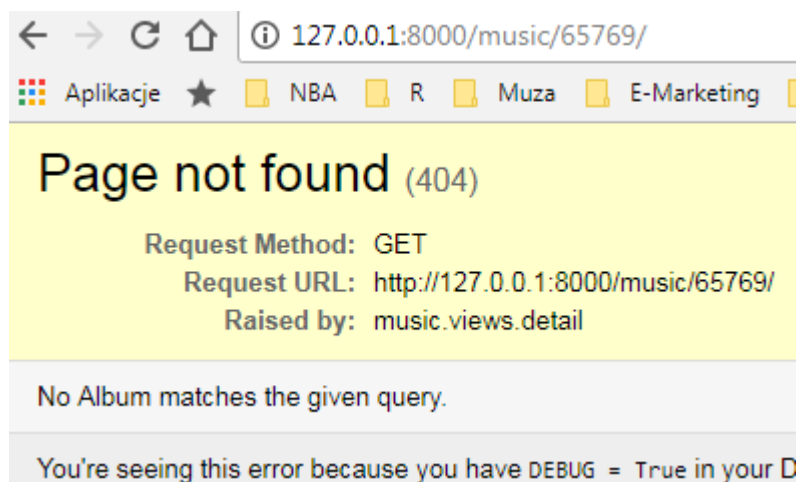
from django.shortcuts import render, get_object_or_404
from .models import Album

def index(request):
    all_albums = Album.objects.all()
    return render(request, 'music/index.html', {'all_albums': all_albums})

def detail(request, dupa):
    album = get_object_or_404(Album, pk=dupa)
    return render(request, 'music/detail.html', {'album': album})

```

After change it is still working:



22. Simple Form *

Actually there's a lot to cover concerning forms, but for this example I'm going to show a really basic way to include forms on your webpage.

What we do?

For each listed song there will be a radio button next to it and at the bottom there will be a submit button. When you select an item and hit favourite it will put a little star next to the song aka favourite it.

Of course it's a pretty dumb way to favourite songs, if you're making an actual application, but it's gonna be a really easy example for teaching purposes.

First we change the 'Song' class, because it needs another attribute:

```
class Song(models.Model):
    album = models.ForeignKey(Album, on_delete=models.CASCADE)
    file_type = models.CharField(max_length=10)
    song_title = models.CharField(max_length=250)
    is_favourite = models.BooleanField(default=False) <--new attribute
    def __str__(self):
        return self.song_title
```

Then migrations: makemigrations and migrate.

First makes the changes (make the SQL file aka make the change file), second applies the changes.

SQL database icon in PyCharm changes to blue meaning changes were applied.

Whenever you change the structure of your database, you also have to restart your server.

Now we create a favourite urls map.

Remember how I said that these urls usually point to a website, pretty much the user requests a view and it gives them back a website. But this isn't always the case, it isn't always a 1:1 relationship. Sometimes you want a url that just performs some logic.

```
urlpatterns = [

    # /music/
    url(r'^$', views.index, name='index'),

    # /music/712/
    url(r'^(?P<dupa>[0-9]+)/$', views.detail, name='detail'), #dupa to album_id

    # /music/712/favourite/
    url(r'^(?P<dupa>[0-9]+)/favourite/$', views.favourite, name='favourite'),

]
```

23. Adding Forms to the Template *

Now we make the form in 'music/detal.html':

```


<h1>{{ album.album_title }}</h1>
<h2>{{ album.artist }}</h2>

<!--
<ul>
    {% for song in album.song_set.all %}
        <li>{{ song.song_title }} - {{ song.file_type }}</li>
    {% endfor %}
</ul>
```


-->

```
{% if error_message %}
    <p><strong>{{ error_message }}</strong></p>
{% endif %}

<form action="{% url 'music:favourite' album.id %}" method="post">
    {% csrf_token %}
    {% for song in album.song_set.all %}
        <input type="radio" name="song" id="song{{ forloop.counter }}" value="{{ song.id }}">
        <label for="song{{ forloop.counter }}">
            {{ song.song_title }}
            {% if song.is_favourite %}
                
            {% endif %}
        </label><br/>
    {% endfor %}
    <input type="submit" value="Favourite">
</form>
```

24. Favorite View Function *

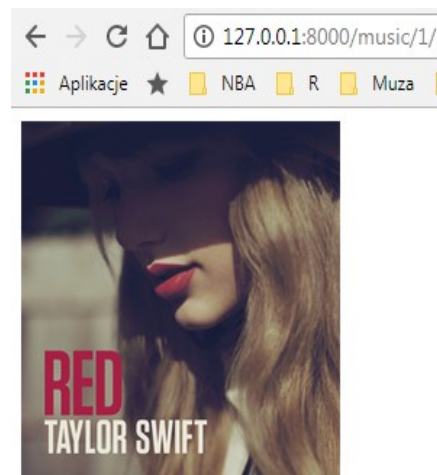
'music/views.py':

```
from django.shortcuts import render, get_object_or_404
from .models import Album, Song
```

```
def index(request):
    all_albums = Album.objects.all()
    return render(request, 'music/index.html', {'all_albums': all_albums})
```

```
def detail(request, dupa):
    album = get_object_or_404(Album, pk=dupa)
    return render(request, 'music/detail.html', {'album': album})
```

```
def favourite(request, dupa):
    album = get_object_or_404(Album, pk=dupa)
    try:
        selected_song = album.song_set.get(pk=request.POST['song'])
    except (KeyError, Song.DoesNotExist):
        return render(request, 'music/detail.html', {
            'album': album,
            'error_message': 'You did not select a valid song'})
    else:
        selected_song.is_favourite = True
        selected_song.save()
        return render(request, 'music/detail.html', {'album': album})
```



Red

Taylor Swift

- ☐ I hate my boyfriend ★
 - ☐ I love bacon ★
 - ☐ Bucky is lucky ★
 - ☐ Ice cream ★
 - ☐ Ice cream ★
 - ☐ Golonka
-

25. Bootstrap and Static Files

Viberr is an application that let's you upload, store, and play all of your music from the cloud. You can now manage and listen to your music from any device, anywhere in the world.

There's a lot to cover with forms and views, but before we get there, right now our number one priority is to work on a design.

We need to figure out how to include CSS into our template. Anytime you want to include images and I'm not talking about user uploaded images, but about logos, images for your background, anything like that or your own CSS to these templates. Those are called static files. They are kind of different from your Python files.

We put static files in '/music/static/music/', similar path like with templates. Additionally Bucky created 'images' in '/static/music', because he wanted it separated from other files.

Another thing we want to start adding is our CSS files. We create '/music/static/music/style.css':

```
body {  
    background: yellow url("images/background.png");  
}
```

In order to say 'ok Django, in this template we're going to start using the static files'. Beginning of the 'index.html':

```

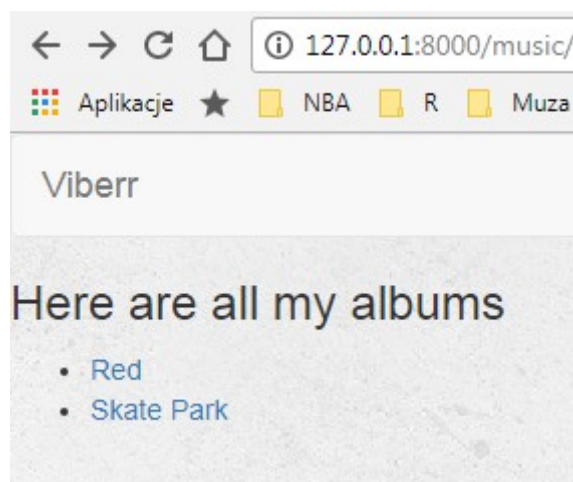
<!-- Loads the path to your static files -->
{% load staticfiles %} <-- this loads the path, it doesn't load the individual files, you have to do
it manually, what we do below
<link rel="stylesheet" type="text/css" href="{% static 'music/style.css' %}"> <-- so just like you
were including a stylesheet and regular HTML. Static references your 'static' directory
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"> <-- Bootstrap
from CDN

<nav class="navbar navbar-default"> <-- here we're starting to build our navigation bar
  <div class="container-fluid">

    <!-- Logo -->
    <div class="navbar-header">
      <a class="navbar-brand" href="{% url 'music:index' %}">Viberr</a>
    </div>

  </div>
</nav>

```



26. Navigation Menu

We continue making our navigation bar. First we design everything and add items with dead links, which we'll fill later.

Beginning of the 'index.html':

```

<!-- Loads the path to your static files -->
{% load staticfiles %}
<link rel="stylesheet" type="text/css" href="{% static 'music/style.css' %}">
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
<link rel="stylesheet" type="text/css" href="https://fonts.googleapis.com/css?
family=Satisfy"> 1)

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>

```

```

<script src="https://ajax.googleapis.com/ajax/lib/jquery/1.12.0/jquery.min.js"></script> 2)

<!-- <nav class="navbar navbar-default"> -->
<nav class="navbar navbar-inverse"> --> changed from default to inverse, changes the color to a dark theme
  <div class="container-fluid">

    <!-- Header -->
    <div class="navbar-header">
      <button type="button" class="navbar-toggle" data-toggle="collapse"
data-target="#topNavBar"> 3)
        <span class="icon-bar"></span> <-- 3 lines in a toggle 'hamburger' button
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{% url 'music:index' %}">Viberr</a>
    </div>

    <!-- Items -->
    <div class="collapse navbar-collapse"> 4)
      <ul class="nav navbar-nav"> 5)
        <li class="active">
          <a href="{% url 'music:index' %}">
            <span class="glyphicon glyphicon-cd" aria-hidden="true"></span>&nbsp; Album
          </a>
        </li>
        <li class="">
          <a href="#"> <-- dead link
            <span class="glyphicon glyphicon-music" aria-hidden="true"></span>&nbsp; Songs
          </a>
        </li>
      </ul>
    </div>

  </div>
</nav>

```

- 1) This one is for the font, it is Google font, check them out, they are awesome and free.
- 2) BS and jQuery javascript files.

BS is not only a designing framework, it actually comes with some cool functionality and the reason that I'm including this is because our menu is going to have a bunch of items: albums, songs, etc. and cool thing about BS you can actually set it up, it comes with the functionality where all the items hide under one icon when the display isn't wide enough (like on different devices, like mobiles or tablets).

- 3) Button that is going to appear if the device doesn't have room to display all of our items, it's going to turn into a menu instead.

Again, these classess are not classes that we have to write ourselves, these all came with bs.

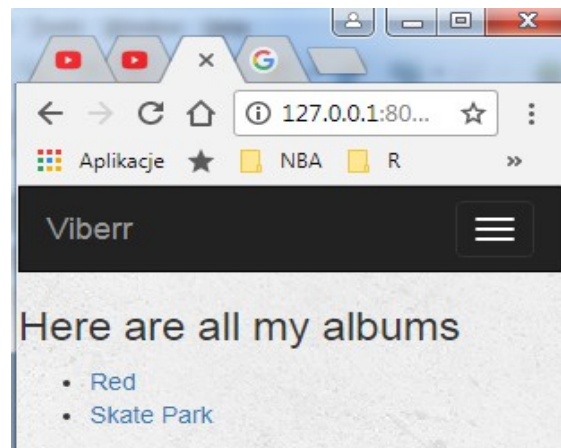
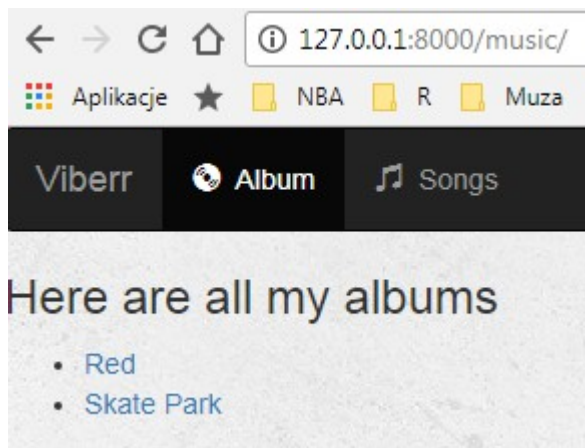
Bucky made also bs tutorial.

'#topNavBar' this name doesn't matter, all of your items in here: Albums, Songs etc. you're going to surround in a container called '#topNavBar' and that way it knows whenever you click this button it's supposed to toggle those or make them visible or invisible, all bs stuff.

4) This little section that is going to hold all of our navigation items is going to be collapsible, so that's going to be hidden unless we click that button on smaller screens.

5) First we add those 2 Album and Songs buttons

Webpage results:



27. Finishing the Navigation Menu

In this video we're going to finish our navigation bar. We already made a couple of buttons on it and now I just want to add that search form and remaining buttons to our items.

New part of 'index.html':

```
<form class="navbar-form navbar-left" role="search" method="get" action="#"> 1)
  <div class="form-group"> <-- input surrounded with div, this is for styling mostly
    <input type="text" class="form-control" name="q" value=""> 2)
  </div>
  <button type="submit" class="btn btn-default">Search</button>
</form>

<ul class="nav navbar-nav navbar-right"> 3)
  <li class="">
    <a href="#">
      <span class="glyphicon glyphicon-plus" aria-hidden="true"></span>&nbsp; Add
Album
    </a>
  </li>
  <li class="">
    <a href="#">
      <span class="glyphicon glyphicon-off" aria-hidden="true"></span>&nbsp; Logout
```

```
</a>
</li>
```

- 1) This form won't be working right now, because we didn't build any search functionality
- 2) So later on, whatever they type in we need to save it to a variable ('q'). You can name it 'query', I'll stand with 'q', because that's kind of the standard, whenever people build searches. 'value=""' shows some text in it.
- 3) Class making that items get displayed on the right-hand side of the navigation bar.

```
<!-- Items -->
<div class="collapse navbar-collapse" id="topNavBar">
```

We gotta add this part. This is all bs, so we don't have to write any of this by hand. Basically what happens is we say 'hey, whenever you need to display this button, because you don't have room for all of these items, whoa look at topNavBar ('data-target="#topNavBar"), that's your target ('id="topNavBar") and these are the items that you want to be collapsing (all in '<!-- Items -->' section), so basically anything that's inside here is going to be hidden.

We have this button which is active and you always want one button active at a time and this is just an indicator for the user to know 'hey, I'm on this section'.

It's important to note this is the default bs style:

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
```

We want to make some tweaks, so whenever we include our css file, we need to put after bs style to overwrite it.

'style.css':

```
body {
  background: yellow url("images/background.png");
}
```

```
.navbar { <-- it's going to make it square edges instead of curvy ones
  border-radius: 0;
}
```

```
.navbar-brand { <-- for the logo
  font-family: 'Satisfy', cursive; <-- this is a backup, so if for some reason the user cannot load
'Satisfy' then we'll give them cursive by default
}
```

Disclaimer: that's how the beginning of 'index.html' should look like:

Order matters, first bs css, then fonts, then jquery, then bs js and finally our css file with our customizations.

From comments: "hi bert dont know if you solved problem yet? the fix would be to place the jquery link first and then the bootstrap.min.js make sure that its 3.3.5 or lower since the latest is not compatible with jquery3.2.1 hope it helps."

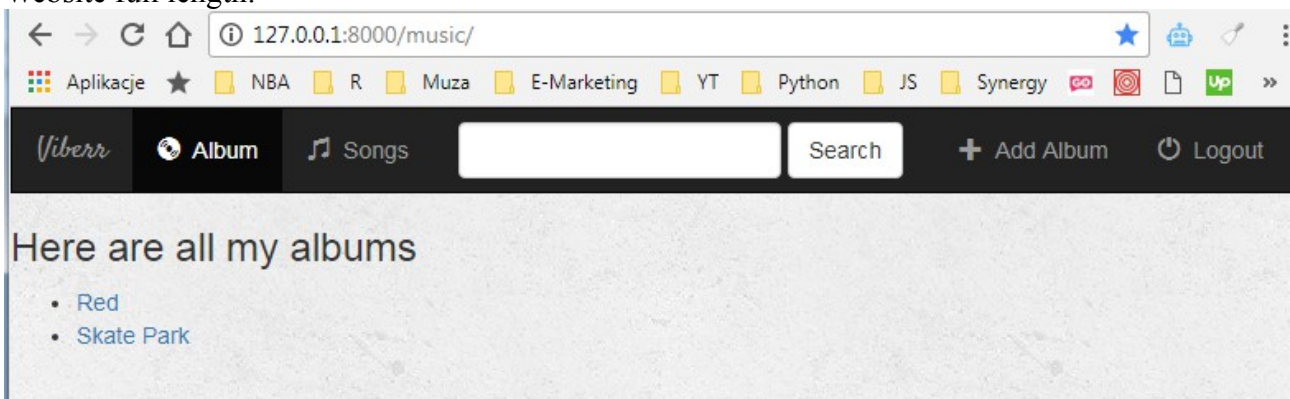
```

<!-- Loads the path to your static files -->
{% load staticfiles %}
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
<link rel="stylesheet" type="text/css" href="https://fonts.googleapis.com/css?family=Satisfy">
<!--<script src="https://ajax.googleapis.com/ajax/lib/jquery/3.2.1/jquery.min.js"></script>--> 1)
<script src="https://code.jquery.com/jquery-3.2.1.min.js" integrity="sha256-
hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
<link rel="stylesheet" type="text/css" href="{% static 'music/style.css' %}">

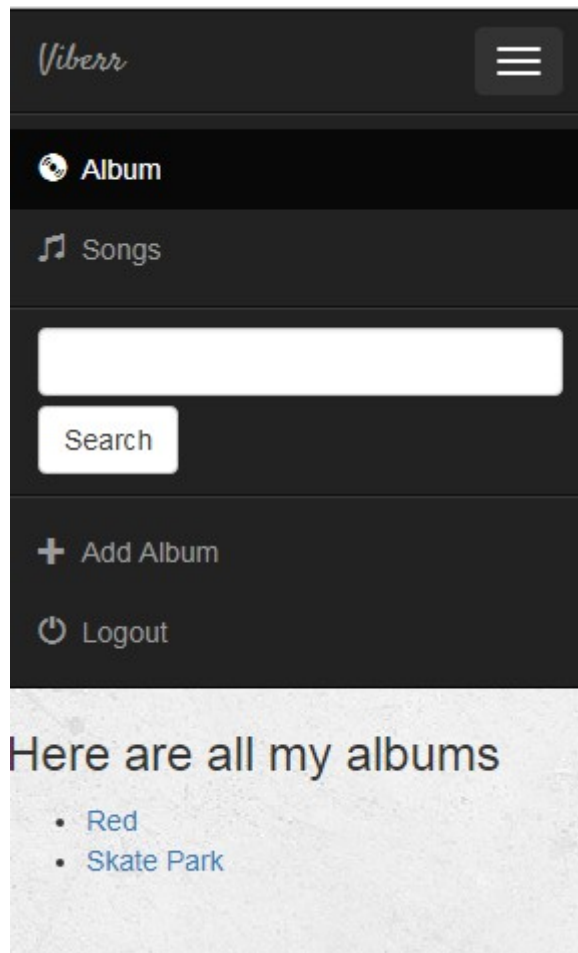
```

1) This was Bucky's, but it didn't work so I used new link from jquery website (next line).

Website full length:



and collapsed:



Tip: Ctrl+F5 or incognito mode if changes are not visible in the browser (cached).

28. Creating a Base Template

Ok guys, look at this beautiful navigation bar at the top of our website, looking good.

Number one rule of software design: Dont Repeat Yourself DRY.

If ever you're making a website or a piece of software and you find yourself having the same code or doing the same thing over and over again, then you're doing sth wrong.

So instead of having the same code in every single template, we're going to build a generic blueprint that is identical or never changes on all pages.

So this is the structure of every single page 'base.html':

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

<head>
  <meta charset="UTF-8">
  <title>{% block title %}Viberr{% endblock %}</title> 1)
```



```

<!-- Loads the path to your static files -->
{% load staticfiles %}
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
<link rel="stylesheet" type="text/css" href="https://fonts.googleapis.com/css?family=Satisfy">
<script src="https://code.jquery.com/jquery-3.2.1.min.js" integrity="sha256-
hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
<link rel="stylesheet" type="text/css" href="{% static 'music/style.css' %}">
</head>

<body>
<!-- <nav class="navbar navbar-default"> -->
<nav class="navbar navbar-inverse">

...
</nav>

{% block body %}
{% endblock %}
</body>

</html>

```

1) When you don't write anything, then it's just going to keep whatever is in the block by default. Including these blocks is actually optional if you want to overwrite the base template.

'index.html':

```

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

{% block body %}
{% if all_albums %}
<h3>Here are all my albums</h3>
<ul>
{% for album in all_albums %}
<li><a href="{% url 'music:detail' album.id %}">{{ album.album_title }}</a></li>
{% endfor %}
</ul>
{% else %}
<h3>There is no albums so far</h3>
{% endif %}
{% endblock %}

```

'detail.html':

```

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

{% block title %}Album Details{% endblock %}

{% block body %}


```

```

<h1>{{ album.album_title }}</h1>
<h2>{{ album.artist }}</h2>

<!--
<ul>
  {% for song in album.song_set.all %}
    <li>{{ song.song_title }} - {{ song.file_type }}</li>
  {% endfor %}
</ul>
-->

{% if error_message %}
  <p><strong>{{ error_message }}</strong></p>
{% endif %}

<form action="{% url 'music:favourite' album.id %}" method="post">
  {% csrf_token %}
  {% for song in album.song_set.all %}
    <input type="radio" name="song" id="song{{ forloop.counter }}" value="{{ song.id }}">
    <label for="song{{ forloop.counter }}">
      {{ song.song_title }}
      {% if song.is_favourite %}
        
      {% endif %}
    </label><br/>
  {% endfor %}
  <input type="submit" value="Favourite">
</form>
{% endblock %}

```

So it's a lot easier whenever you have to update your overall theme or template in one location using a base theme.

Now we have a much better, cleaner layout.

29. Generic views

Basically whenever you make a website or go to the website you're going to notice that all the websites that you go to, kind of have the same pattern. They either display a list of objects or they have details about one object. So if you think about a website like YT, whenever you search for a video, it just has a list of all the videos that you can scroll through and find your video and then when you click one, it has details about that video, of course the player and all the comments and related videos, whatever.

So what about website like Facebook? On your newsfeed it has a list of everyone's post and then you can click either someone's profile or picture or something and then it has details about that object.

So again, it all comes down to 2 things: a list of objects and details about one single object. So I'm sure no matter what website you think of, they always have these same patterns.

Now, on our website that we're making, we're kind of doing the same thing, the home page is just a list of all of our albums and it's going to be the cover photo, the artists and stuff later on, whenever we design it properly, but it's basically a list of all of our albums then whenever we click one, there's details about that individual item. So the album logo, all the songs, tomata tomato... So since Django realised that people were just creating these same patterns over and over again, why not help them out and speed up the process a little bit.

Note: before this tutorial Bucky pretty much cleaned out everything in the 'index.html' and 'details.html' and got rid of all the favouriting form, because that's not the proper way to actually favourite songs, that was just a little example. So Bucky pretty much stripped everything down, except displaying the bare basics like the album name, song titles and tomata tomato...

Note2: Old files were renamed to: '[filename] – actual till 28', which means these files were used up to 28. tutorial.

So instead of the way we've been making views before, which were basically functions, what we're going to make now is something called Generic Views (Gvs). So in this tutorial I'm just going to make two gvs. The first one is going to be a list gv, which is going to list all of our albums and the second one is going to be a detail gv, which is going to give us details about that individual album. **So it's going to be the exact same as before to the user, but we're gonna write a lot less code.**

So how do we make a gv? Well, instead of functions, we actually use classes.

'music/views.py':

```
from django.views import generic
from .models import Album

class IndexView(generic.ListView):
    template_name = 'music/index.html'
    context_object_name = 'all_albums'

    def get_queryset(self):
        return Album.objects.all()

class DetailView(generic.DetailView):
    model = Album
    template_name = 'music/detail.html'
```

So for the homepage, since it's the index page, I'm just going to say 'class IndexView()'. Now the type of view we use we actually inherit it, so there are a couple different types of course, in this tutorial I'm going to be teaching you guys about the list gv and also the detail gv. **But remember, on the index page is just a list of all of our albums, so we're actually going to inherit from 'generic.ListView'.**

First we need to specify what template we're using, so 'template_name = 'music/index.html'', so that just says whenever we get a list of all of the albums, plug them into this template right here. Pretty easy :)

Now, there is only one other thing that we need to make in here and that is a **QuerySet** function. So basically we're going to make a function called 'get_queryset' and all this is going to do is we're going to query the database for whatever albums we want and in this example we will get all of them, so 'return Album.objects.all()'.

And guys, you know what? This is actually all you need to do for this gv. That's all we need to do for the homepage.

Let's make a `DetailView` now. So 'class `DetailView()`' and again whenever you're looking at the details of an album, it's not a list of things, it's just details about one object. So the gv for this is 'generic.`DetailView`'. And this is even easier, so since you're not actually getting a list of objects, all you need to say is... ok, first of all, what model or what type of object are you trying to get the detail of and we'll just say 'model = `Album`' trying to look at the details for an album. And the last thing is the template name just like 'template_name = 'music/details.html'', so whenever you give me that album, what template do you want me to plug it into and that's just the 'detail' template right there.

Now the only other file we have to change is this 'urls.py':

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

app_name = 'music'

urlpatterns = [

    # /music/
    url(r'^$', views.IndexView.as_view(), name='index'),
    # /music/712/
    url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as_view(), name='detail'),
]
```

We just need to remove this 'favourite' url part, so not a whole lot is going to change in this file, but **a couple things I want to point out that are very important is this. You know how I said that each url pattern has to be hooked up to the view function.** Well, right now we're using these classes, I mean this class doesn't even have any functions or methods inside it ('class `DetailView`' in the 'views.py'), so what we need to do instead of these functions (-> 'views.index'), we actually need to reference on the class and then convert it to a view. So just write 'view' and then you write the class name which is 'IndexView' and then since you actually need a function you just call 'as_view()'. So we're pretty much saying 'hey, we're using a class, but since you need a view just go ahead and treat it as a view'.

We're going to do the same thing right here with the 'DetailView' url pattern. The last thing I want to point out is whenever you're using a detail view it actually expects the primary key, so we're going to give it the primary key of one (Taylor Swift album) or two (Myth album). So instead of 'album_id' (I had 'dupa') here, we're going to write 'pk' and that's it.

Last piece of the puzzle. Whenever you query all of the albums right here ('return `Album.objects.all()`'), it's pretty much going to return a list of all of your album objects, but it didn't know what variable to store it in so we could use it in our template. How did it know that it was supposed to be called 'all_albums'? Well, by default whenever we use a `ListView` it's going to go ahead and query this (-> 'def get_queryset' function) and it's going to return it in an object called 'object_list'. **Again, the default name whenever you return a list of objects from the `ListView`**

(so not `DetailView`?) generic function is going to be called `'object_list'`, but if you ever want to override it, then you just have to make a variable called `'context_object_name'` and make it equals `'all_albums'` (by default `'context_object_name = object_list'`).

Hopefully when we refresh our website, it should look exactly the same as before introducing the gvs.

To sum up, there were bunch of patterns that people were using over and over again whenever they're making websites, mainly displaying a list of objects and displaying the details for a single object. So instead of having to write all those functions like you always do, instead what you can do is use these gvs. So it's a lot cleaner and check it out, we now made two entire webpages and look how cleaner this is than just using view functions.

From the comments:

- 1) how come the detail view know that the object name is 'album' in detail.html ?? is it the lower case of the "model" name mentioned in `DetailView` class ???
- Django automatically chooses an appropriate context variable name based on the model name ('Album' in this case, which is converted to the lowercase 'album' when being passed to the template). See here for more details, about 3/4 of the way down:
<https://docs.djangoproject.com/en/1.10/intro/tutorial04/>
- 2) `model = Album` is the same thing as `query_set = Album.objects.all` and it's the shortcut way of `def get_queryset()`
- 3) Great tutorials man. Keep up the good work but here's a side note. For the `IndexView` the code is simple enough to just use the default implementation. So instead of overriding `get_queryset` all you have to do is override the model attribute to use the model you want.

INSTEAD OF:

```
class IndexView(generic.ListView):
    template_name = 'template.html'

    def get_queryset(self):
        return Album.objects.all()
```

YOU CAN JUST DO:

```
class IndexView(generic.ListView):
    model = Album
    template_name = 'template.html'
```

And that's it!!!

ALSO(from the Django docs):

" If `context_object_name` is not set, the context name will be constructed from the `model_name` of the model that the queryset is composed from. For example, the model `Article` would have context object named 'article'."

Try Django 1.8 - Create an MVP Landing Page – coding entrepreneurs
Git & GitHub Crash Course For Beginners – Traversy Media
Socratica yt channel

30. Model Forms

Note: Bucky changed layout again to save time.

The design of your website doesn't really matter to Django, so design it however you want as long as you follow along with these tutorials you guys are going to be learning the basics of Django.

What are model forms (mf) and why do we need them? **Well, mfs help us speed up development whenever we include forms on our website and they do this by saving you a bunch of time by a) generating the actual HTML code for the forms, b) taking care of basic form validation.** So for example, whenever we have a form and we need the user to input the album name, if they just don't type anything at all, it's going to say 'hey, this can't be blank, you need to write something in here...' so on and so forth. And the last thing is they actually take care of saving that data to the database. So we looked how we can do this manually using that database shell or command line and we also looked at how we can do it in the admin panel.

Now we're going to make a form where the user can type in all the information for a new album and then it can go ahead and save it all behind the scenes. We don't have to worry about any of the hard work.

So first things first, go ahead and hop over to 'models.py'.

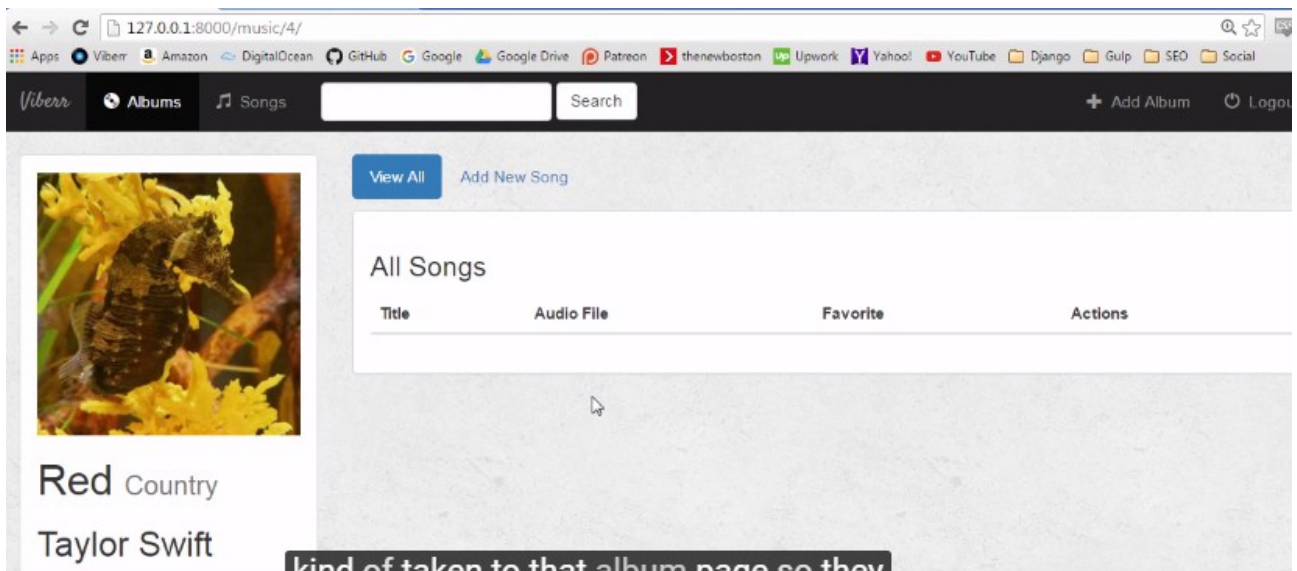
```
from django.db import models
from django.core.urlresolvers import reverse <-- we need to import one other module

class Album(models.Model):
    artist = models.CharField(max_length=250)
    album_title = models.CharField(max_length=500)
    genre = models.CharField(max_length=100)
    album_logo = models.CharField(max_length=1000)

    def get_absolute_url(self): <-- pass in self, 1)
        return reverse('music:detail', kwargs={'pk': self.pk})

    def __str__(self):
        return self.album_title + ' - ' + self.artist
...
```

You know how I said that in this example what I'm going to do is I'm going to make a real basic form that lets the user type in the album name and the artist, and the genre and also a logo, and the hit submit. Well, whenever they hit submit and it adds it to the database, where are they going to be redirected? We're going to redirect them to this page right here like '127.0.0.1:8000/**music/4**'.



I think it will be cool if after they submit the form then they are kind of taken to that album page, so they see 'all right, I need to add a song now tomato tomato'.

So in order to do that we add another function, we will call it 'get_absolute_url'.

And what we're going to return is essentially the details page of the album that we just created. We're going to 'return reverse' and the first parameter is the view name, so that was 'music:detail'. Now remember, whenever we use this 'detail' view, what it takes is the primary key of whatever album we are trying to view the details of. So, how do we pass that in? Well, the primary key is kind of a hidden field right here and in order to pass it in just write 'kwargs' (in other words 'keyword args') and we just want to use the primary key of whatever this object is, whatever one we just created so 'self.pk'.

So whenever we create a new album it's going to add it to the database, is going to give it some primary key (like 5 or 6) and it's going to take it to this detail view with whatever number the primary key is. Simple enough.

So the next step in this process is a hop over to your views.

```
from django.views import generic
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from .models import Album

...

class AlbumCreate(CreateView):
    model = Album
    fields = ['artist', 'album_title', 'genre', 'album_logo']
```

All right, so we're going to add a new class that essentially creates this form view, but in order to do that we actually need to import it.

So basically whenever you want to make a form to create a new object, what you need to import is this 'CreateView'. Now, there is also 'UpdateView', this is the form for editing an object. And there is also 'DeleteView', so this isn't an actual form that you can see, because whenever you delete an

object you don't need to fill out a form, you just hit delete and then it redirects you somewhere. But we'll take a look at that in the next tutorial.

So whenever we create our class to add a new album what we're going to do is this 'class AlbumCreate()', we're going to inherit from 'CreateView'. So the first thing it says is 'all right, you're trying to create a new object, what type of object are you trying to create?'. Well, just like before, we need to specify it right there, so 'model = Album'. 'All right, you're trying to create a new album, that's fantastic. Now the only other thing that you need to write are what fields do you need, so what attributes do you want me to allow the user to fill out'. Now, you need to specify these, because sometimes you just don't want to put all the fields in there. Maybe you want to like make a crawler that gets the logo automatically or just fill in the genre behind the scene or something. But in this example, we're just going to allow the user to type in every single thing. So in order to specify that just write fields and you put those as a list, so 'fields = ['artist', 'album_title', 'genre', 'album_logo']'. Ok, that looks pretty good.

Now, the last thing we need to do in terms of setting it up kind of in the code is over in urls we need to assign a url pattern to this view ('class AlbumCreate' in 'views.py') right here (in 'urls.py').

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

app_name = 'music'

urlpatterns = [

    # /music/
    url(r'^$', views.IndexView.as_view(), name='index'),
    # /music/712/
    url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as_view(), name='detail'),

    # /music/album/add <- url structure
    url(r'^album/add/$', views.AlbumCreate.as_view(), name='album-add'),

]
```

'as_view()' just like before, because it is a class we do need to convert it to view.

And the reason why we don't need to specify a primary key is just because we're creating a new album, so it doesn't have a primary key yet. After we add it and after we create it that's when it's going to be assigned a unique id aka primary key.

Now we've got form generated, but what we have to do after this is we actually need to make a template that we can plug it into. And that's what we're going to be doing in the next video.

31. ModelForm and CreateView

Now that we made a form using model forms (mfs), now we actually need to make a template using HTML so we have somewhere to plug it into. **So this is actually pretty important, the way that you name it. Whenever you're using mfs, the default (again, make sure it is in templates and then your app directory, like 'templates/music') name it needs to be the model name underscore form and the model name is in lowercase.** So in this case I am making a form for

creating new albums, so I'm just going to write 'album_form' (.html). Whenever we make a form for making new songs, it's going to be 'song_form'.

So that's the default file that it's going to look at and that is why, if you're looking through the views like 'all right, where did I specify the template name?'. We didn't, because we don't need to. Pretty cool :)

This is an empty panel, in other words, a white square and then the form is going to go right in here. Again, it doesn't really matter how you design your page, this is just Bucky's personal preference to make it easy.

'album_form.html':

```
{% extends 'music/base.html' %}
{% block title %}Add a New Album{% endblock %}
{% block albums_active %}active{% endblock %}

{% block body %}
<div class="container-fluid">

    <div class="row">

        <div class="col-sm-12 col-md-7">
            <div class="panel panel-default">
                <div class="panel-body">

                    <form class="form-horizontal" action="" method="post"
enctype="multipart/form-data"> 1) 2) 3)
                        {% csrf_token %} 4)
                        {% include 'music/form-template.html' %} 5)
                        <div class="form-group"> 1)
                            <div class="col-sm-offset-2 col-sm-10">
                                <button type="submit" class="btn btn-success">Submit</button> 6)
                            </div>
                        </div>
                    </form>

                </div>
            </div>
        </div>

    </div>

</div>
{% endblock %}
```

1) class, just some styling, BS stuff

2) action, you set this equal to nothing, because it's going to be redirected to the same view and that's where it gets handled behind the scenes

3) enctype, right now whenever the user adds a new album we just say 'hey, go ahead and paste in some url of the album cover, but later on we're actually gonna let them upload a file. So even

though we don't need this now, I'm going to set this equal to 'multipart/form-data' and that is just making sure that we can handle whatever files they upload in the form, whenever we need it later on.

4) cross site request forgery token, this is for common security practices

So whenever Django generates these forms for us, what it does is it gives us some pieces of information. So for each model that we generate it from it's going to give us a label, which basically says 'artist', 'album_title', 'genre' or 'album_logo'. And another thing it does for each one, is it gives us that form field. So all of these right here are just going to be inputs. **So pretty much it's going to give us a huge dictionary of labels and inputs.**

So what we could do is we could go through and in this form handle each one, so ok I want to make form group and put the label to the left and input to the right. Then I'm going to do it for the other one, make sure it's lined up and then do it to the right. And then later on whenever we make another form we have to do it again and again... **But that again breaks the number one rule of software development: DRY.**

So instead the proper way to handle these is even though you can do it the long way if you want, **it's better to make a generic form template and then just include it (5).** This is going to be a simple template for how we want our labels and the form fields to be laid out. We're going to lay them out left and right, right next to each other like a nice grid or table, but you can do it whatever you want.

Ok, so we have all the fields, all the places that the user can type in the information, so the last thing we need to put in here is a button (6).

All right, so now we need to make the 'form-template.html' file.

```
{% for field in form %} 1)
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-10">
    <span class="text-danger small">{{ field.errors }}</span> 2)
  </div>
  <label class="control-label col-sm-2">{{ field.label_tag }}</label> 3)
  <div class="col-sm-10">{{ field }}</div> 4)
</div>
{% endfor %}
```

Again, whenever Django generates these forms for you, they're going to give you each field one by one. So in this model (class 'Album') they are basically a whole bunch of input areas, but they can also be maybe a dropdown or whenever you let the user upload files, it can be a button that says 'choose a file', whatever. So it comes in a list called '**forms**' and we are going to name each one 'field' (1).

So the name of this is whatever you named your **object**, we named our 'field'. So whenever they have an error like let's say they didn't fill out the artist name, then I'm going to want that error message to appear right above the input so they know what field they are getting an error message on. This part only displays when there are errors (2).

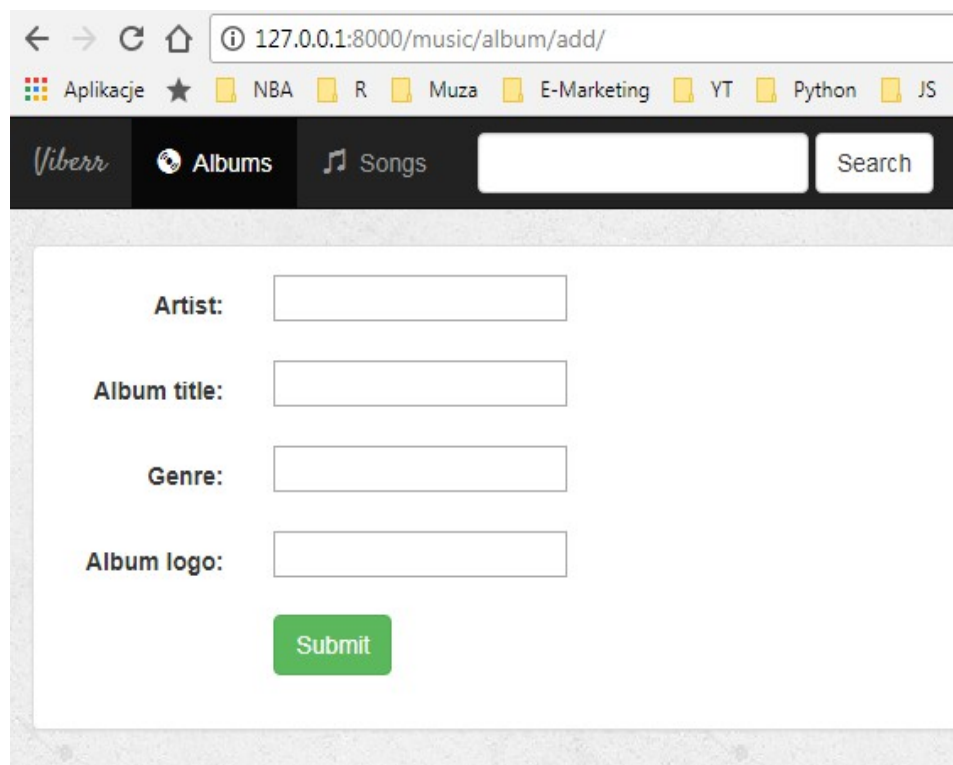
This is the part that is always going to display and the first thing is label. In other words, to the left is just going to say 'artist' and then it's going to have an input box, then it's going to say 'album title' for the label and then it's going to have an input area.

That's the actual text what essentially the label is (3).
Actual input area or drop-down or whatever (4).

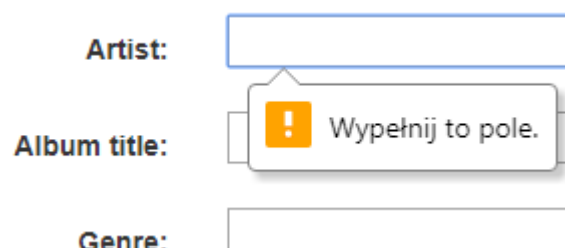
Last thing we need to do is add a link to make sure the user has some way of getting to this form, so in 'base.html':

```
...
<ul class="nav navbar-nav navbar-right">
    <li>
        <a href="{% url 'music:album-add' %}"> <-- just dead link # before the change
        <span class="glyphicon glyphicon-plus" aria-hidden="true"></span>&nbsp; Add
Album
    </a>
</li>
</li>
</li>
...
```

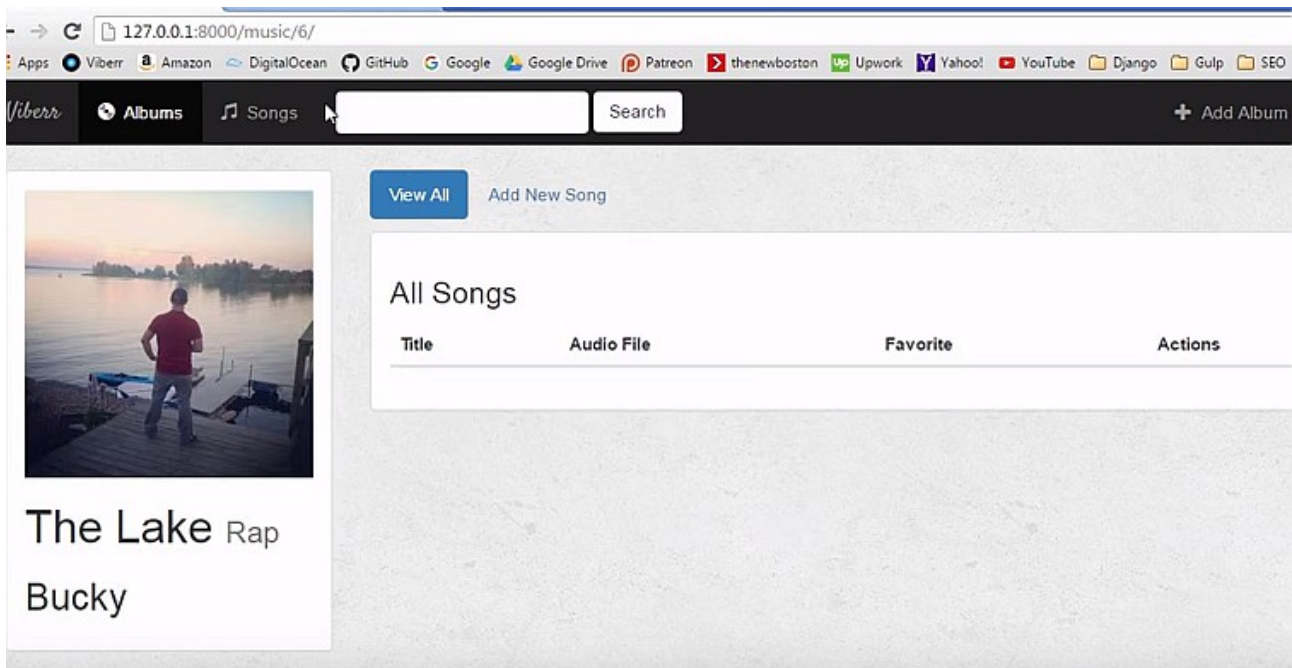
Everything should be good to go now. Just refresh this to make sure our link is updated. Click on '+ Add Album' in the navbar and check this out:



When error appears:



After filling the form and hitting the 'Submit' button it redirects you to the details page (e.g. '/music/6/')



And remember, we already plugged in by getting the primary key of this object and now whenever I look at albums again, now I have a new album, boom roasted look at that.

How awesome is that? So that is the basics of using model forms (mofos). **It makes it a whole lot easier, because we didn't need to generate any of this form HTML, it does simple data validation and also took care of adding it to the database** all for us, pretty sweet.

32. UpdateView and DeleteView

In this video we're going to be talking about 'UpdateView' which is essentially a view that you can edit an album or object. And the cool thing about this is it's actually the same exact form as adding it. We just pre-populate it with whatever information is in there, so it makes it really easy. And also the 'DetailView' which gives the users a really quick way to delete objects.

'views.py':

```
from django.views import generic
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.core.urlresolvers import reverse_lazy <- we are importing one more module
from .models import Album

class IndexView(generic.ListView):
    ...

class AlbumCreate(CreateView):
    model = Album
    fields = ['artist', 'album_title', 'genre', 'album_logo']
```

```
class AlbumUpdate(UpdateView):
    model = Album
    fields = ['artist', 'album_title', 'genre', 'album_logo']
```

```
class AlbumDelete>DeleteView):
    model = Album
    success_url = reverse_lazy('music:index')
```

So since creating an album and editing an album are so similar, we can actually just copy-paste it. Again, 'AlbumCreate' is for creating a new object, however whenever you edit it, you need to inherit from this class 'UpdateView'. So of course the first thing, just like all of these, you need to say what model it is, what kind of object you're trying to update. And that's it. Bucky wishes they name it 'EditView' instead of 'UpdateView', since it is for editing.

Now the 'DeleteView' part. You see, whenever you're making a new album or editing it, you're pretty much going to use this standard form right here:

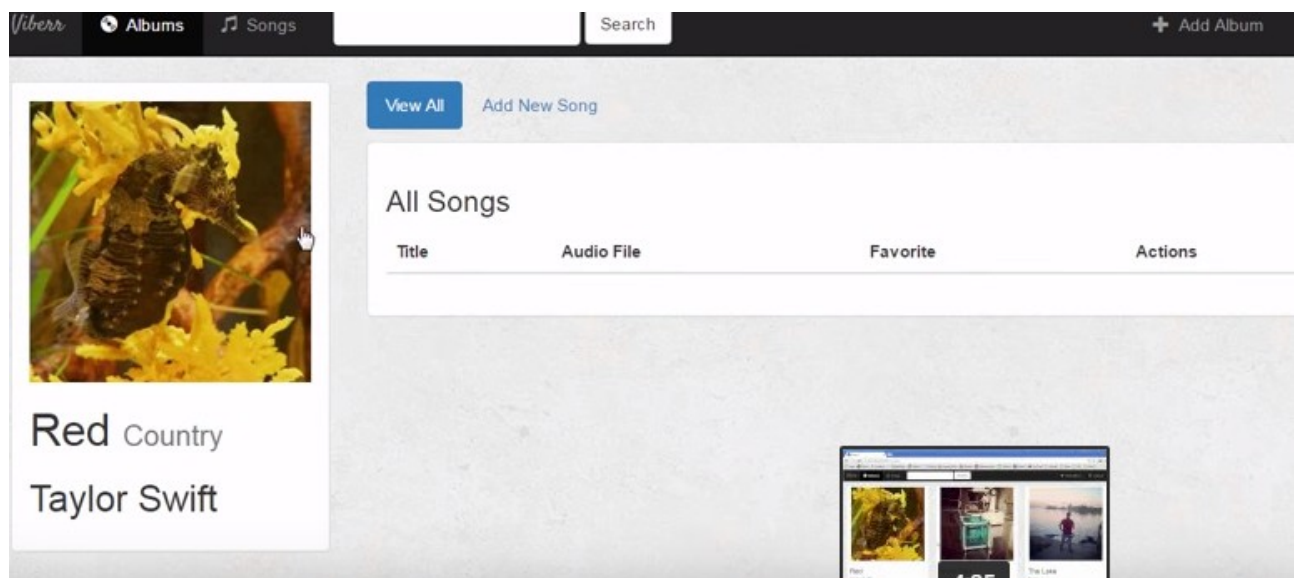
Artist:

Album title:

Genre:

Album logo:

However whenever you just click 'delete', it doesn't take you to a new form and you don't have to like fill anything out. It just pretty much deletes this and then it's going to say 'ok, after you delete an object, where do you want me to go? Because maybe you're on this details page:



and you delete it'. Well then this page doesn't exist anymore.

So we need to use this module right here ('reverse_lazy') and it pretty much says 'where do you want me to redirect to?'. So it's actually called 'success_url', so whenever you successfully delete an object just call 'reverse_lazy' and then pass in the name of the view, and we're just going to redirect them to the homepage. And that's all we need to do, easy peasy :)

So after this we have to hop over in 'urls.py' and make our url patterns to link an 'UpdateView' to this (class 'AlbumUpdate' in 'views.py') and a 'DeleteView' to this (class 'AlbumDelete').

'urls.py':

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

app_name = 'music'

urlpatterns = [

    # /music/
    url(r'^$', views.IndexView.as_view(), name='index'),
    # /music/712/
    url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as_view(), name='detail'),

    <-- those urls structures with hashtags are very useful
    # /music/album/add/ 1) <-- this is the page with the form while adding a new album
    url(r'^album/add/$', views.AlbumCreate.as_view(), name='album-add'),
    # /music/album/2/ 1) <-- we are updating/editing the album with the pk=2
    url(r'^album/(?P<pk>[0-9]+)/$', views.AlbumUpdate.as_view(), name='album-update'),
    # /music/album/2/delete/ 1)
    url(r'^album/(?P<pk>[0-9]+)/delete/$', views.AlbumDelete.as_view(), name='album-delete'),

]
```

All right, so this is the url that you're going to redirect the user to whenever they are adding a new album, editing it or deleting it (1).

Now that we have everything set up, we can actually link 'delete' button (with the trash icon in the homepage) to the 'delete' url (in 'urls.py'). How do we do that?

'index.html':

```
...
<!-- Delete Album -->
        <form action="{% url 'music:album-delete' album.id %}" method="post"
style="display: inline;"> 1)
        {% csrf_token %}
        <input type="hidden" name="album_id" value="{{ album.id }}" />
        <button type="submit" class="btn btn-default btn-sm">
            <span class="glyphicon glyphicon-trash"></span>
        </button>
    </form>
...
```

1) we need to pass in the primary key of that album

Now let's check our edit form. We don't have any button for album edition, but we can just copy-paste the url in the browser, in this case '/music/album/6/':

It gives you the exact same form, but now all the information is pre-populated and ready for editing. To delete it, just click on 'trash' button and remember, it always redirects you to the homepage. We stated that in the 'reverse_lazy' part, so whenever you want to redirect them to another that maybe says 'congratulations, you successfully deleted an album', this is where you would change it.

But again, look at how clean our code is right now and a lot easier whenever we let Django take care of everything behind the scenes.

Python Tutorial for Absolute Beginners #1 - What Are Variables? CS Dojo

33. Upload files

Target of the tutorial: We replace 'album_logo' type which is a text url to a file (a picture) that can be uploaded from user's computer to our server.

First we go to admin panel and delete all our albums and songs. Reason: we change the structure of our database ('alum_logo' field type).

After that instead of these field being a character field, which essentially just means text, what you need to make it is a file field.

models.py:

```
...
class Album(models.Model):
    artist = models.CharField(max_length=250)
    album_title = models.CharField(max_length=500)
    genre = models.CharField(max_length=100)
    #album_logo = models.CharField(max_length=1000) <-- we replace this...
    album_logo = models.FileField() <-- ... with this
...
```

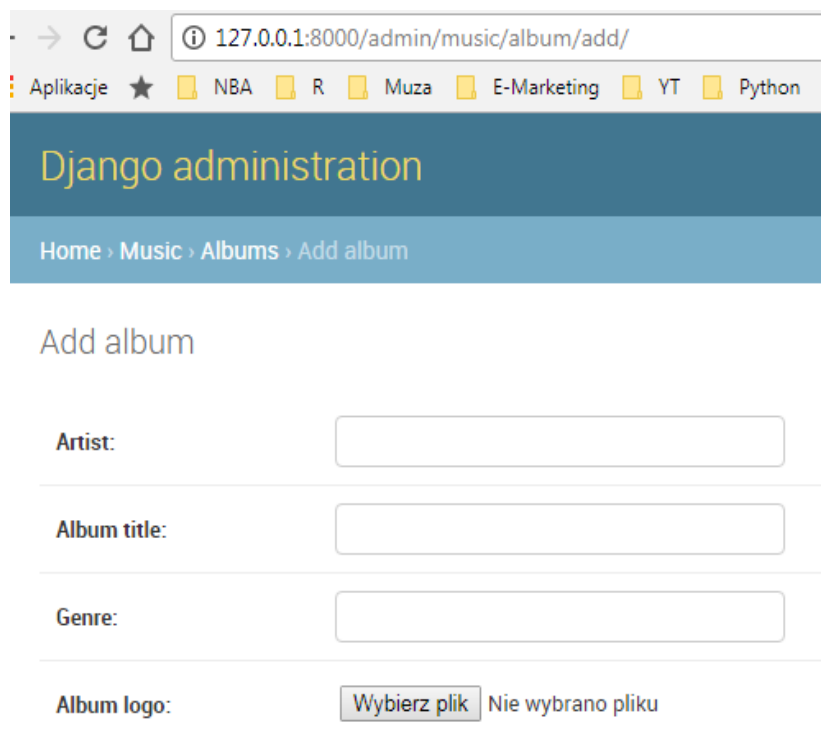
So now that we actually changed the structure of our database we need to:

```
>>> python manage.py makemigrations music
and
```



```
>>> python manage.py migrate
```

In admin panel we can add a new album and we can see that we have a 'choose file' button so that is proper:



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/admin/music/album/add/'. The browser's bookmark bar includes 'Aplikacje', 'NBA', 'R', 'Muza', 'E-Marketing', 'YT', and 'Python'. The page title is 'Django administration'. The breadcrumb trail is 'Home > Music > Albums > Add album'. The form is titled 'Add album' and contains four fields: 'Artist:', 'Album title:', 'Genre:', and 'Album logo:'. The 'Album logo:' field has a 'Wybierz plik' button and the text 'Nie wybrano pliku'.

So now we need to specify a couple things. Most importantly, whenever the user uploads an image, where are we going to store it? So where do we specify that?

'website/settings.py' (at the bottom):

```
...  
# Static files (CSS, JavaScript, Images)  
# https://docs.djangoproject.com/en/1.11/howto/static-files/  
  
STATIC_URL = '/static/'  
  
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')  
MEDIA_URL = '/media/'
```

We actually need to add two global variables:

'MEDIA_ROOT' this is just the file path of the directory that we're going to pull all of the files in.

And we actually want to join this 'BASE_DIR' (defined at the beginning of the settings file) and this just resembles whatever directory this is (mine is: 'C:\Users\Mikołaj\Documents\Django Project\website'). So we're going to take that 'BASE_DIR' and we're going to join it with 'media'. The last one that we need to write is this 'MEDIA_URL = '/media/'

So essentially whenever the user uploads an image, it's going to create a new directory named 'media' and you're essentially going to have 3 directories: website, music and media.

Why do you need both of these 'MEDIA_ROOT' and 'MEDIA_URL'? Well, the first one is the

actual directory on your computer that's going to store those files and the second one actually references the relative url and this is what your browser's use for accessing the files over HTTP. So the first one is for kind of your server and the second one is for users and the browsers.

Now hop over to 'website/urls/py':

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

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

if settings.DEBUG:
    urlpatterns += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

We need to do here a couple things. The first thing we need to do is specifically import these settings (we just introduced in the settings.py), because we tweaked some things on it (1). So what we are saying is 'hey, whenever we're just testing this out, go ahead and use this url (MEDIA_URL = '/media/') to store all the files in.

Debug True it pretty much means development mode. So if you look at your settings.py, there is 'DEBUG = True' part and it says 'right now you are making this program in debug mode', this is good, that helps us find all the bugs, but whenever we put this on the live server, we actually want to change it to false, which means production mode and not developer mode. So whenever your website is in actual production mode, you probably don't want to store all the images in the same directory right here. Whenever your website's really big, you actually store all of your media on another server or somewhere else on your own server, but anyways what saying 'alright, whenever we're just in developer mode just go ahead and use the url we just told you to'.

So in order to do that, we're just going to append the items right here, because if we stuck them in there (outside the if loop), they're going to be there all the time.

Ok, so the if loop (2) pretty much says 'whenever we're in developer mode, just use these' and we can change it later in production.

So from here we've got tiny, tiny change that we have to make. Let's hop over to our homepage, i.e. 'index.html'. So basically wherever you had your picture (logo) before, this isn't just a piece of text anymore, it's actually an image object. Right now we're referencing the image object, so what we want to do is just take this image object and refer to its url. This is going to give us the file path of wherever this image is stored. So now we can use that file path, because of course your image source (img src=) needs to be a file path and not just a bunch of byte data.

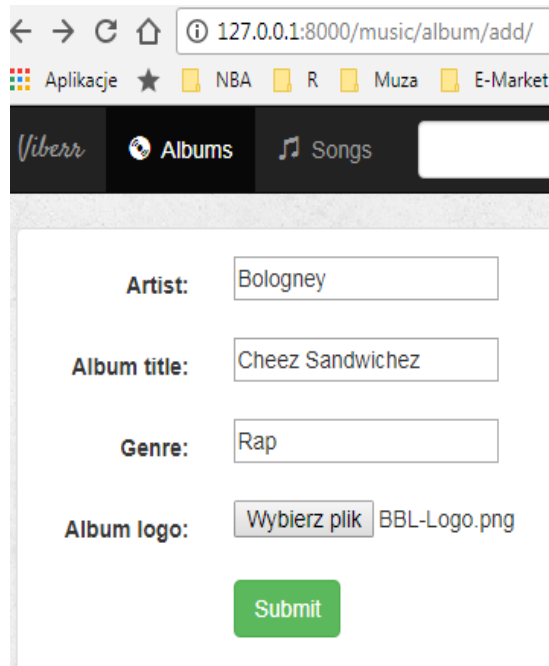
So in 'index.html' and 'detail.html' (wherever needed) we change:

```
album.album_logo
to:
```

album.album_logo.url

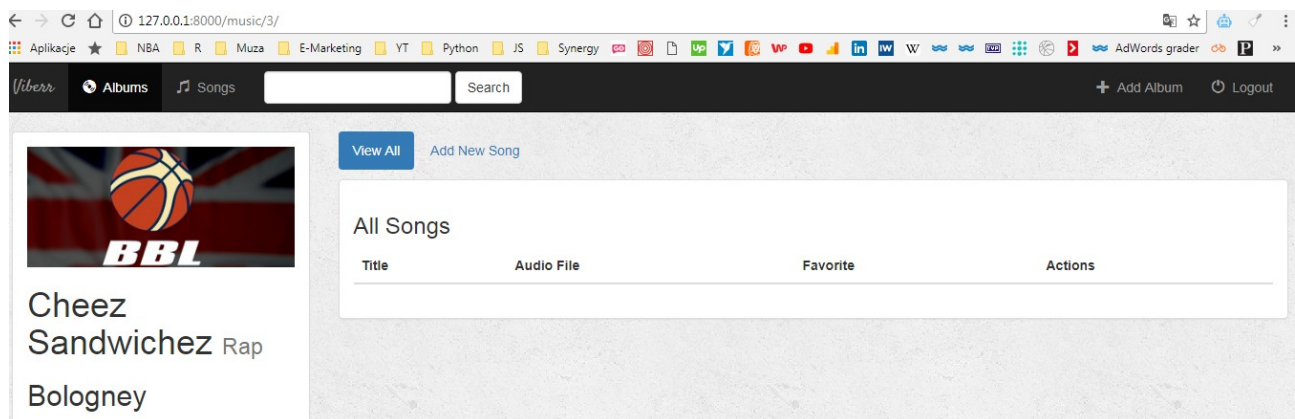
Again, this is just the url of the file wherever it is stored on your server.

Let's now add a new album:



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/music/album/add/'. The page has a dark header with the 'Viberr' logo and navigation links for 'Albums' and 'Songs'. Below the header, there is a form to add a new album. The form contains four input fields: 'Artist' with the value 'Bologney', 'Album title' with the value 'Cheez Sandwichez', 'Genre' with the value 'Rap', and 'Album logo' with a file selection button labeled 'Wybierz plik' and the filename 'BBL-Logo.png'. A green 'Submit' button is located at the bottom of the form.

and submit:



In the meantime Bucky referenced the wrong thing :)

Look how it generated this 'media' directory right here, so all of your user uploaded files are gonna be stored right in there.

So again, pretty simple, just a few files to change and boom roasted, users can now upload files.

34. User Registration

Purpose: In this video we're going to let users register for our site or create a new account and also log in.

Before we get to the code, we need to talk about a couple core concepts. First of all, as you know, Django already comes with the ability to create users and we know that, because we already created an admin. Now, if you play around in your admin panel and click 'Users', you're going to see that this is your information right here:

Home > Authentication and Authorization > Users

Select user to change

Q Search

Action: Go 0 of 1 selected

<input type="checkbox"/>	USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
<input type="checkbox"/>	admin	mikolaj.myszka@gmail.com			✓

1 user

So you have a 'Username', 'Email Address', 'First Name', 'Last Name' and we don't have to fill these out ('First Name', 'Last Name'), we just left them blank and also the 'Staff Status'.

And again, if you click it (on 'admin'), you can see that here is your password that is hashed:

Home > Authentication and Authorization > Users > admin

Change user

Username:
Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password: **algorithm:** pbkdf2_sha256 **iterations:** 36000 **salt:** S2opwQ***** **hash:** wTyruR*****
Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using [this form](#).

and **it is important to note that whenever Django stores passwords, they're not raw text.** So whenever hackers hack your database, they don't get anyone's password and a bunch of these fields right there below.

So we can just go ahead and make the forms like we did before, but what you always want to do is you want to make these in kind of a different way. Now I say that, because whenever we have a website and I want users to sign up, I really don't care about their first or last name or maybe there's some other thing that you want them to fill in, maybe like their relationship status if you're making a dating site. So whenever you talk about user authentication or registration, you always want to handle it manually. So I'm going to show you guys best way to do that.

In our 'music' app we create a new file called 'forms.py'. So you know like before, whenever we wanted to create, let's say a new album, we just said 'all right, we're going to use this model Album ('model = Album' in 'class AlbumCreate') and it generated a new form based on this model. Well, that's essentially what we're going to do with the users, however since we want to tweak it a little bit, for example we don't need an input for first and last name, cause we don't care about those, we're pretty much going to inherit from 'Users' and overwrite whatever we want to or add whatever fields we want to.

'forms.py':

```

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

class UserForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ['username', 'email', 'password']

```

So this is going to give us the base 'User' class, just a generic user class that we can use.

And now we're going to make a new user form class and that's going to tweak it to whatever we want to display on the form. So 'class UserForm' and this is going to inherit from 'forms.ModelForm'. So again, whenever I talk to you guys about model classes, model forms all you're doing right here is you're going to make a blueprint that's going to be used whenever you're making the forms.

So the first thing we're gonna do is we're just going to make a new class 'Meta'. It's basically information about your class. Sounds kind of weird, but there you go.

So the base model is going to be User ('model = User'), in other words whenever a user creates or signs up for your site, it's just going to go in the same table right here (i.e. same fields):

Action:	<input type="text" value="-----"/>	<input type="button" value="Go"/>	0 of 1 selected		
<input type="checkbox"/>	USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
<input type="checkbox"/>	admin	mikolaj.myszka@gmail.com			✓

so we're not creating anything brand new, we're just saying what information we need out of them. So for the 'fields' we just want to say what fields we want to appear on the form. So what I'm going to request is they pick a username, email address and also their password. We can leave all the rest of the stuff out (e.g. First name, last name, staff status).

We actually need to specify the password, it's going to be equal to 'forms.CharField'. Now, if we just leave this like this, then it's just going to be a plain text, in other words whenever user tries to type in their password, it's not going to give you those **actressess**, it's going to give you just normal text (like 'yo mofos'), so then anyone peeping over their shoulder can see what their password is. So how do we specify that this is actually a password field and you need to hide those characters? Well, just go ahead and write 'widget=forms.PasswordInput'. So now whenever we display this form, what's going to appear on the page for the user is area to pick a username, email and passwords.

So now over in 'views.py', now that we have our blueprint of how we want the form to be laid out, taken care of, I'm actually going to show you guys how to do a bunch of cool stuff in this video.

What I'm going to do is not only whenever they register it creates an account and pops our information into this database, but I'm also going to show you guys how to have them log in automatically, in that way on every page you can say 'hey Bucky' or whatever custom information you want specific to them. And aside from logging in, I'm also going to show you guys after you submit the form it redirects you to the homepage, but you may want to redirect them to like their profile page, their newsfeed, whatever.

'views.py':

```

from django.views import generic
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.core.urlresolvers import reverse_lazy
from django.shortcuts import render, redirect <---- 1)
from django.contrib.auth import authenticate, login <----- 2)
from django.views.generic import View
from .models import Album
from .views import UserForm <---- 3)
...

```

So we need to import a bunch of stuff (1). So this 'redirect' module is going to redirect them to whatever page you want after they log in.

What this 'authenticate' module is going to do (2) is going to take a username and a password and it's going to verify that they are indeed a user in the existing database. And this 'login' it just attaches a session id, so then no matter what page you're on, you don't have to like authenticate them every time and they don't have to log in every page. It just gives them an id that they can use while they're browsing the site and, you know, a **railer** session id.

And the last thing (3) is we actually need to import this 'forms.py' file that we've just created. Looking beautiful mate!

Python OOP Tutorial 1: Classes and Instances Corey Schafer zrob to!!

This is kind of easy tutorial so don't look at things to complex.

To jest jak budowa z klocków

funkcje spelniaja swoje funkconalnosci

<https://simpleisbetterthancomplex.com/tutorial/2017/02/18/how-to-create-user-sign-up-view.html>
 Creating the Django User Registration Form (Django Tutorial) | Part 15 Max Goodridge
 Dominik Kozaczko yt channel

35. User Model and Creating Accounts

So you know before whenever we wanted to create a new object we use 'CreateView', whenever we wanted to show a list of objects we use 'ListView', 'DetailView' was to get the details of one object.

'views.py':

```

...
class UserFormView(View): <--- 1)
    form_class = UserForm <--- 2)
    template_name = 'music/registration_form.html'

    #display blank form
    def get(self, request): <--- 3)
        form = self.form_class(None) <-- 5)
        return render(request, self.template_name, {'form': form}) <-- 6)

    #process form data
    def post(self, request): <--- 4)

```

```
form = self.form_class(request.POST) <--- 7)

if form.is_valid():

    user = form.save(commit=False) <--- 8)

    #cleaned (normalized) data
    username = form.cleaned_data['username']
    password = form.cleaned_data['password']
    user.set_password(password)
    user.save()
```

Well, since we're customizing a bunch of stuff, we are just going to import or inherit from generic view. So I'm going to make this class called 'UserFormView' and again, we are gonna inherit from 'View' (1).

So the first thing you need to specify in here is what is your form class, in other words, what is the blueprint that you're going to use for your form and of course that's just the one we created over here (in 'forms.html') 'UserForm' (2).

Next thing we need to specify is template name, we don't have it created yet, but basically it's an HTML file that the form is going to be included in.

Now I want to show you guys one of the coolest things about class-based views. You know how basically whenever the user goes to the form the first time like '+Add Album' form, then they're just making a GET request, so they're just getting a blank form. Now whenever they hit 'Submit' then where does that information go? Well, it actually goes to the same url, so how does it get handled differently, because whenever you submit a form it's not a GET request, it's actually a POST request. **So in other words, we're using the same url to handle 2 different types of request: GET which pretty much means 'just give this form for the first time' or POST which means the user filled out information and they want to submit the form.**

So from here what you can do is sth like this: you can say if method equals POST and if method equals GET, and that logic is going to work fine, because whenever it's a GET method then that means they just want the registration page, the blank form and whenever it's a POST method then that means they actually submitted the form. So you can split your logic up like that, but then you have a bunch of code and I mean we're going to be writing a bunch of stuff in here, so it's going to get cluttered (zaśmiecony). However there is a better way to do it and that is this. **Whenever you're using class-based views you can actually take your GET and POST logic, and separate it into built-in functions.** So the built-in function for GET requests is just 'get' (3).

So again, whenever the user wants this form and it's a GET request, it's going to call this function right here (3).

Now you know what's coming next, whenever they submit a form and it's a POST request, then you can use this function right here and then all of your logic is separated from one another. And look how beautiful this is, I just wanna make out with my computer right now :)

All right, let's go ahead and take care of the GET request first (a new user coming to your site and they didn't sign up yet, they don't have any account). We want to make a form variable and set it equal to 'self.form_class', in other words we just want to use this form – the 'UserForm' and what context do we need to pass in there? 'None' (5). So by default it doesn't have any data, that's what the user is trying to fill in some data in it. Now we need to just render it like we have been doing everything else. So we need to pass in the 'request', 'self.template_name' and this basically means 'all right, for the form, where do you want me to plop it in, what HTML file, that one right there. And the last one is just the form itself (6). So in other words, all this is doing is displaying a blank form to the user, nothing new, I mean we did it before like in the first tutorial.

So now we get to the fun part, what happens when the user actually types in their information and hits submit. Well, now we need to register them, add them to the database. Of course the first thing we need is a reference to the actual form. However instead of 'None', we actually want to pass in whatever information they typed into that form, so that is actually the 'request.POST'. So whenever they hit 'Submit' all of that gets stored in this POST data right here (7) that we can pass to the form and the form can validate that data. **So this is already built-in Django functionality, is going to validate it, make sure it's correct.** In other words, this is always going to be true as long as they didn't use like some weird symbols from China or like they forgot to fill in their username or sth like that. As long as it is valid data.

So we're going to write 'if form.is_valid()', what do we want to do? Well, eventually we're going to take their information and we're going to store it inside the database, however before we just take their data and plop it right in our database, we usually want to make some checks and balances and do some further validation ourselves. So what I'm going to do is I'm going to make a 'user' object and this is just going to be a user object using whatever they typed into the form so 'form.save()'. And what you also want to put here is 'commit=False' (8). So what this line does is it just creates an object from the form. What it doesn't do is it doesn't save it to the database yet, so at this point we didn't enter their information into the database, we're just pretty much storing it locally, so we can do whatever we want with it. First thing we want to do is actually we want to get the clean or normalized data. Clean data is basically data that is formatted properly. In other words, let's say that you want the user to input a date. You know how everyone in the world uses a different date format. So this is going to normalize it or unifies it so everyone's using the same format and it just makes sure that it's ready to enter your database properly. So for the username, in order to get the clean version of the data, it just formed, it's just 'form.cleaned_data' and then you just write whatever field so '['username']'. We do the same for password. So this is actually the data that you want to plop in your database, so that is the username and password. There is one more thing that needs to be explained and that is how you set a user password. Passwords aren't just a normal text, if you look they are this weird hash value:

Change user

Username:	<input type="text" value="admin"/>
Required. 30 characters or fewer. Letters, digits and @/./+/-/_ only.	
Password:	algorithm: pbkdf2_sha256 iterations: 24000 salt: ogRCmr***** hash: zlcylh***I*****
Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.	

So if you just try to change it to plain text, you're going to get a whole bunch of errors. So whenever you do need to change the user's password, this is how you do it. You take the user object and you call a 'set_password' function and this is where you pass in their password.

So after this everything is set properly, you can now save the user. 'user.save()' actually saves them to the database. So after this line runs, they are going to appear right in there (in the Django administration).

So that's what registration is, boom roasted, the user is now registered for the website and they are in the database.

Commentarze z dolu tutoriala

36. User Authentication and Login

Target: I'm going to show you how to authenticate and log in the user.

'views.py':

```
class UserFormView(View):
    form_class = UserForm
    template_name = 'music/registration_form.html'

    # display blank form
    def get(self, request):
        form = self.form_class(None)
        return render(request, self.template_name, {'form': form})

    # process form data
    def post(self, request):
        form = self.form_class(request.POST)

        if form.is_valid():

            user = form.save(commit=False)

            # cleaned (normalized) data
            username = form.cleaned_data['username']
            password = form.cleaned_data['password']
            user.set_password(password)
            user.save()

            # returns user object if credentials correct
            user = authenticate(username=username, password=password) <--- 1)

            if user is not None: <--- 2)

                if user.is_active: <--- 3)
                    login(request, user) <--- 4)
                    return redirect('music:index')

            return render(request, self.template_name, {form: 'form'}) <--- 5)
```

Now we are going to make a new variable called 'user' and you actually want to call '**authenticate**'. So this is going to take two keyword arguments and we're going to pass in the 'username' and 'password' variable. **So essentially what this function does is it takes a username and password and it checks in the database to see if they are an actual user, to see if they exist (1).**

So now we can check to make sure that we actually got a user back. You can say 'user is not None'. So again, whenever you run this function (authenticate), if it's authenticated then it's going to take that user and return it as the 'user' variable. So now it should be equal to Bucky or admin, or whoever (2).

So we need to check one more thing and this is kind of weird, but basically what Django let's you

do is they let you have users in here (in Django administration) that you can either like ban or make inactive, maybe someone just acting up (wybryk) and you didn't want to delete their entire account so you just disable their account, so you need to check for that as well before they log in, so 'if user.is_active' (3) then this means alright their account doesn't get banned and of course they logged in proper password, the user exists, so how actually do you log them in and attach a session? Well, it's really easy, you just pass in 'request' and 'user', that's it (4). They are now logged in to your website, so you can actually now refer to them is 'request.user' and again, don't write this, actually, but later on whenever you want to print out their username or sth then you just write 'request.user.username' or 'request.user.profile_photo', whatever information you have about them.

And check this out, so now they are logged in and you can now refer them that way, but after they log in I'm actually going to want to redirect them to the homepage, because if they just log in and hit submit and then it just shows them a blank form, then it's like 'what aaaaaa... I am log in? Did I press the right thing?' :) So in order to do that you just write 'return redirect('music:index')', boom roasted!

So if this is true and they submitted the form properly then hopefully they log and get redirected righr there (to homepage), but if they didn't then what I'm going to do is this, I'm just going to say 'ok, try again'. So I'm just going to 'return render(request, self.template_name, {form: 'form'})' (5). So again, if they didn't log in or if their account is banned or whatever, then we're just going to say 'all right, you know what, here's a blank form for you'.

So now the last 2 things we need to do:

'urls.py':

```
...
# /music/register/
url(r'^register/$', views.UserFormView.as_view(), name='register'),
]
```

You can also name it 'Create an account' or whatever you want.

And we also need to create this registration form (template copy pasted from 'album_form.html' with minor tweaks).

'registration_form.html':

```
{% extends 'music/base.html' %}
{% block title %}Register{% endblock %}

{% block body %}
<div class="container-fluid">

    <div class="row">

        <div class="col-sm-12 col-md-7">
            <div class="panel panel-default">
                <div class="panel-body">
                    <h3>Create a New Account</h3>
                    {% if error_message %}
                    <p><strong>{{ error_message }}</strong></p>
                    {% endif %}
                    <form class="form-horizontal" action="" method="post" enctype="multipart/form-
data">

                        {% csrf_token %}
                        {% include 'music/form-template.html' %}
                        <div class="form-group">
```

```

        <div class="col-sm-offset-2 col-sm-10">
            <button type="submit" class="btn btn-success">Submit</button>
        </div>
    </div>
</form>

</div>
</div>
</div>

</div>

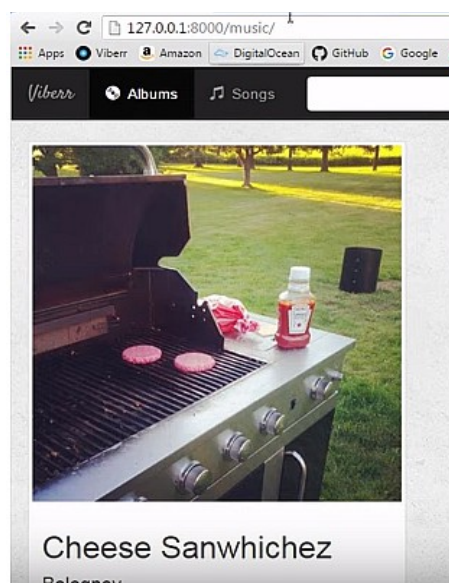
</div>
{% endblock %}

```

Let's look at our registration page:

The reason this is pre-populated is this is actually Google Chrome (default save password setting), it has nothing to do with Django.

We fill in the form and whenever we submit, check it out, it logs us in and redirects to the homepage.



Now if we log in as 'admin' to Django Administration, we see new user:

Action: 0 of 2 selected

<input type="checkbox"/>	USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
<input type="checkbox"/>	admin	admin@email.com			✓
<input type="checkbox"/>	baconbuddy	bacon22@gmail.com			✗

2 items

So that is after a quite a long tutorial how you create new users, how you let them register and also log in to your Django website.

I'm going to take a break now, I'm going to watch a hockey game, the Pittsburgh Penguins they better win :)