Django Full stack Web Development

-Jose Portilla-

# Part B: Back-End

Django Full stack Web Development

[Part B: Back-End i1](#_Toc12816926)

[\*Note of symbol meaning 3](#_Toc12816927)

[I/ Python: 4](#_Toc12816928)

[1/Introduction 4](#_Toc12816929)

[2/ Regular Expression 4](#_Toc12816930)

[II/ Django: 6](#_Toc12816931)

[1/ Django lv1: 6](#_Toc12816932)

[a/ Django Setup 6](#_Toc12816933)

[b/ Django Project 7](#_Toc12816934)

[c/ Django Application 9](#_Toc12816935)

[d/ URLs mapping: 11](#_Toc12816936)

[e/ Django Template 11](#_Toc12816937)

[f/ Template Challenge. 12](#_Toc12816938)

[g/ Django static files 13](#_Toc12816939)

[2/ Django level two 15](#_Toc12816940)

[a/ Introduction 15](#_Toc12816941)

[b/ Django Models Overview 15](#_Toc12816942)

[c/ Creating Models 17](#_Toc12816943)

[d/ Population Scrips 18](#_Toc12816944)

[e/ Models-Templates-Views Paradigm 19](#_Toc12816945)

[f/ Django level two project 20](#_Toc12816946)

[3/ Django level Three: 21](#_Toc12816947)

[a/ Django Form 21](#_Toc12816948)

[b/ Forms code along 21](#_Toc12816949)

[c/ Form Validation 24](#_Toc12816950)

[d/ Model Forms (forms.ModelForm): 26](#_Toc12816951)

[e/ Django exercise – Save forms data to models 27](#_Toc12816952)

[4/ Django level four 28](#_Toc12816953)

[a/ Relative URLS with Templates 28](#_Toc12816954)

[b/ Code: 28](#_Toc12816955)

[c/ URL template inheritance 28](#_Toc12816956)

[d/ Filter: 28](#_Toc12816957)

[5/ Django Level 5: 32](#_Toc12816958)

[a/ Django passwords 32](#_Toc12816959)

[b/ User Models 34](#_Toc12816960)

[c/ Creating User Model and User Form: 35](#_Toc12816961)

[d/ Registration 36](#_Toc12816962)

[e/ Logins 37](#_Toc12816963)

# \*Note of symbol meaning

(?) Confused

(!) Notice importance info for future usages

(!!)  Struggle haven’t been solved

## I/ Python:

### 1/Introduction

* So in the course, Jose use Atom editor.
* He recommend install three packages to help learning easier:
  + Flatformoi-ide-terminal  to use terminal (command prompt directly in Atom)
  + Language-Python (This has already been installed in Atom)
  + Autocomplete- Python
  + Atom - django

#I can see that number of user of those package increase significantly .

Flatformio-ide-terminal : 200k  2.7 Mil

Autocomplete: 700K  2.2 Mil

*Người người học code nhà nhà học code…*

* For flatformio, change link in shell override to *“C:\Windows\System32\cmd.exe”*

### 2/ Regular Expression

* To use recular expression, we have to **import** the library: ***import*** *re*
* Some method:
  + *re.search(patterns[0],text)* : This create a match object / none object. Match object contains more than just Boolean value
  + *re.search().*start show start position of the match
  + Split(): *(re.split('@',email*)  same as spit() method for string
  + re.findall(“sd\*”,text): to find all match value to statement:
    - d\* as zero or more d
    - d+ as one or more d
    - d? as zero or one d
    - d{number} mean {number} of d
    - d(number, another\_number) mean {number} of d or {another\_number} of d
    - [] mean any\_of\_the\_inside\_character s

*# Ex: [sd] mean s or d. Ex [sd] mean zero or more s,d: sdddsssddss not dsssddss*

* + - # ## exclusion
    - pattern=['[^!?. ]+']

*#^ mean to exclude. This can be consider as a miltipel character*

* + - lowercase = ['[a-z]+']
    - [a-z] mean any cha **from** a to z (lowercase)
    - [A-Z] is for uppercase
    - for special:
      * special=[r'\d’]
      * d mean digits
      * D mean non-digits
      * s mean whitespace
      * S mean none-whitespace
      * w mean all the alphanumeric
      * W mean non-alphanumeric

## II/ Django:

### 1/ Django lv1:

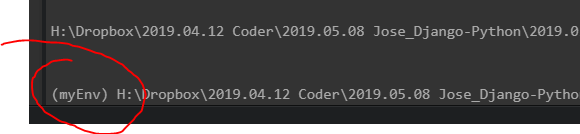
#### X/ Note for Ubuntu:

* *To run the conda command from terminal, we have to add the following line to .bashrc file:* 
  + *export PATH="/home/user\_name/anaconda/bin:$PATH"*
  + Run conda init to restart the terminal

#### a/ Django Setup

* Packages to install:
  + pip install bcrypt
  + pip install django[argon2]
  + Install sqlparse
  + Install pillow
* Django is a free open source web framework, started **from** 2003, in Lawrence Journal - World. (*Many sites use Django such as Pinterest, PBS, Instagram, BitBucket, Washington Times, Mozilla, and more)*
* Django documents and tutorial are excellent as the creator was surrounded by tons of writer in the Journal.
* “Virtual environment” – “venv”:
  + This allow us to install Python and Package in our local computer to:
    - Test the new update without affecting our website as update usually contain new feature which may change the web’s compatibility
* To use virtual environment with conda:
  + conda create –name myEnv.  myEnv is name of the new virtual environment we’ve created.
  + To install something in the environment, there some way:
    - IF we didn’t create the environment:
      * *conda create –name myEnv django*: To create environment together with the lastest Django other thing such as Python 3.6/ Python 3.7 ,etc
      * *conda create –name myEnv django = version*: To create the environment with the specific version of Django/other thing such as Python 3.6/ Python 3.7 ,etc.
    - If we have created the environment:
      * *conda install django*:
  + activate myEnv  then we can avtive the environment by this. And **from** now, anything installed with pip or conda will only be installed for this environment. This allow us to install separated versions of Packages and Python in a computer. deactivate myEnv is to deactivate the environment.

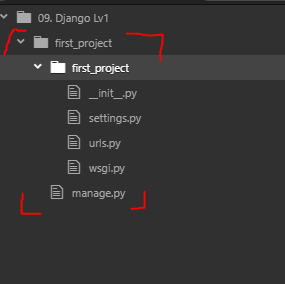
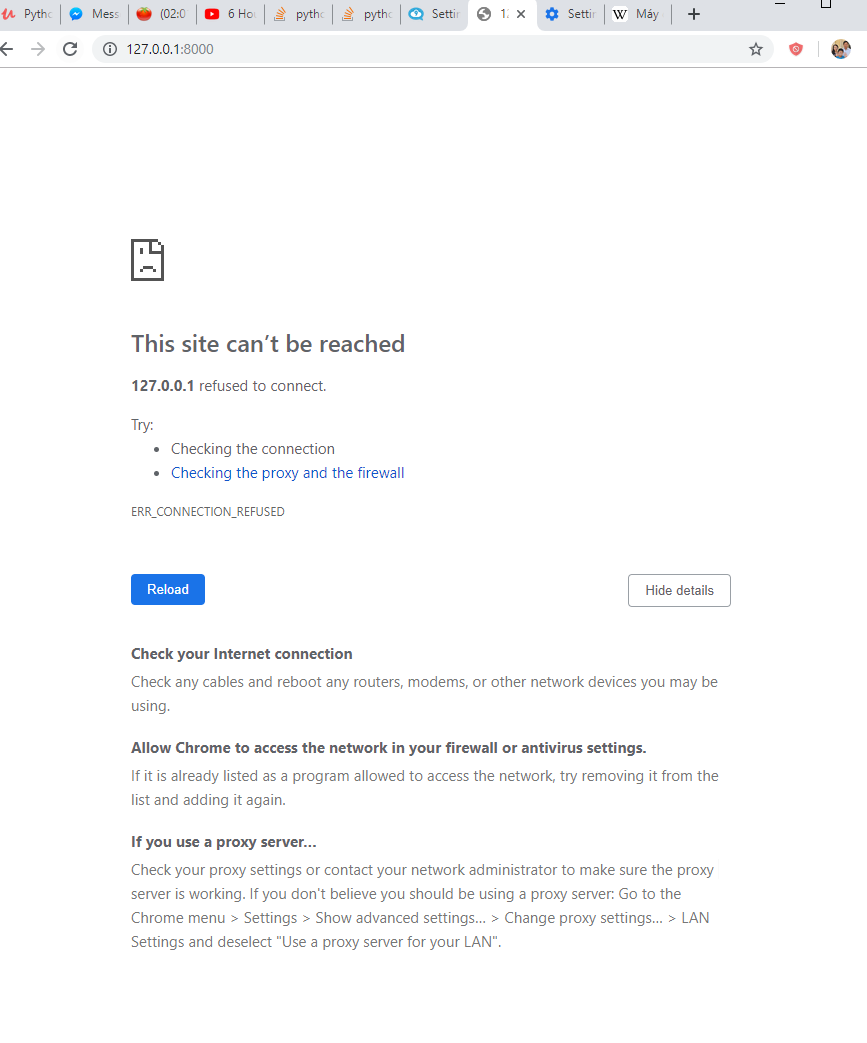
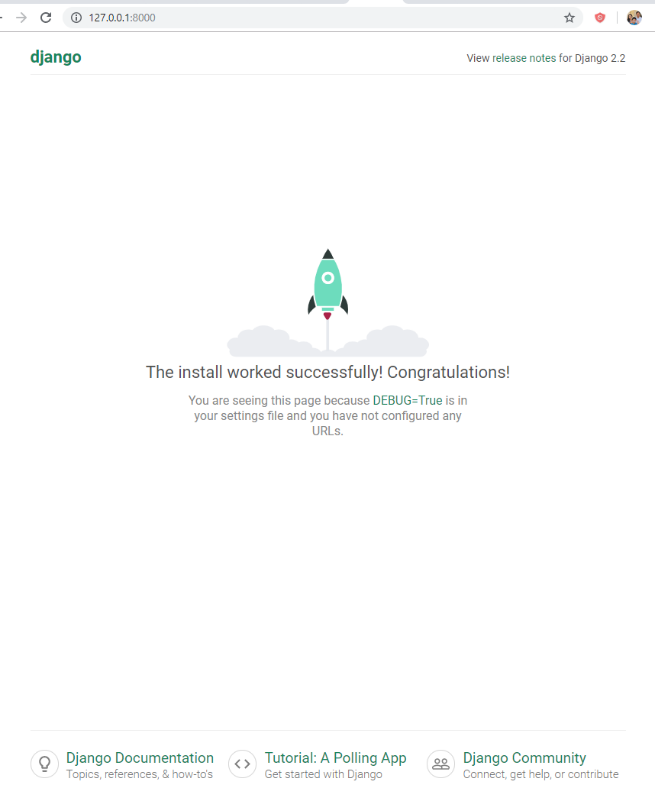
*#Note: Recommend to use environment. So that our project is self-contained and not rin into issues when packages update.*



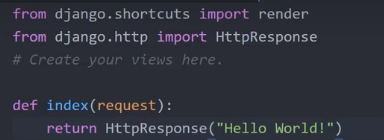
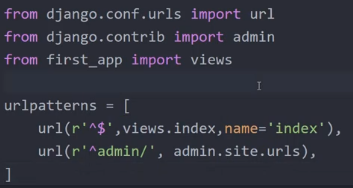
*The sign (myEnv) mean the environment has been activated.*

* + conda info --envs: this code command conda to show us all environments installed in our computer.
  + To choose a specific version for our environment:
* **Notice**: we can also use Python distribution to install Django: pip install django.

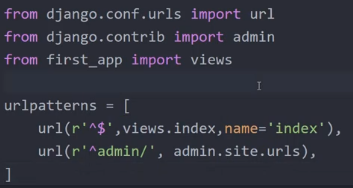
#### b/ Django Project

* When creating Django, there’s a tool was created along with it: *django-admin.*  This’ll be used below.
* To create a new folder: *mkdir folder\_name* (This can be used anywhere, in the environment or Python or command prompt)
* To create frist project: *django-admin startproject first\_project*. And these files, folder are created: 
  + **\_\_init\_\_.py** is a blank file to let Python know and treat this folder as an packages.
  + **settings.py** is the file to store all project settings
  + **urls.py**: Python script that store all the URL patterns for the project. Basically they relate to different pages of my web apps and how it should connect to the end user.  This particular files is going to make a lot of use of regular expression. So remember to review again the regular expression lecture.
  + **wsgi.py**: The Python script that acts like Web Gate Server Way Interface. It’ll help us to deploy the our web app to production on some online server/
  + **manage.py**: This is a Python script that we will use a lot. It’ll associate with many commands as we build to web apps.
* So first, we run the manage.py file by command : *python manage.oy runserver*.
  + At first, I have some issue: 
  + The issues happened due to the Server was killed by Ctrl command. Mean when we get the server **from** command line to enter it in address bar on browser, do not use Ctrl +C *#Ctrl +C is control Break, which is to quit server on command line.* use the mouse instead, or us Python’s alternative : alt +C (This was said in command line but I didn’y understand…).
  + ou are seeing this page because [DEBUG=True](_blank) is in your settings file and you have not configured any URLs. Debug need to turn to False when our wibsite go to production. So that user cannot access debug tools,
  + migrate notication can be ignore at the moment cause it will be back latter in the course. These are files to migrate database **from** one design to other.

#### c/ Django Application

* Django Projects vs Django Applications
* *python manage.py startapp first\_app*.
* Files created in first\_app folder:
  + A “migrations” folder tores database specific information as it relates to the models:
    - \_\_init\_\_.py
    - **admin.py** : You can register your models here which Django will then use them with Django's admin interface.
    - **apps.py**: Here you can place application specific configurations
    - **models.py**: Here you store the application's data *models* ?
    - **tests.py**: Here you can store test functions to test your code
    - **views.py**: This is where you have functions that handle requests and return responses
* Step to create 1st app:
  + - Step 1/ Add a “*first\_app*” to list *INSTALLED\_APP* in **setting.py**
  + Step 2/ Open **view.py** type :
    - **from** django.shortcuts **import** render
    - **from** django.http **import** HttpResponse (**Import** HttpResponse Object)
    - The “*request*” here is a *HttpResponse* object. We can make this whatever we want, not necessary “*request*”. This mean each time user view this, the *HttpResponse* is called.
    - Each view of the application would happen within the **views.p**y file. The **index()**  we created here is **a views**.
    - 
  + Step 3: To see what happen in view.py, we have to map **view.py** to **urls.py**:
    - **from** first\_app **import** views
    - 

#### d/ URLs mapping:

* The include() function allows us to look for a match with regular expressions and link back to our application's own urls.py file.
  + 
* In the app\_two folder, we create a mini urls.py file with the same configuration as the orginal urls.py.
* The syntax to add the urlpatterns list of the original urls.py file:
  + url[r’^app\_two/’,include(app\_two.urls)] (This ‘app\_two/’ actually can be anything you want. It’s the domain name.
  + In the mini urls.py file of app\_two, we write any the output we need for our related web.
* Additional:
  + [r’^$’] mean any link.
  + [r’^app\_two/’] mean anylink contain of ‘app\_two/’

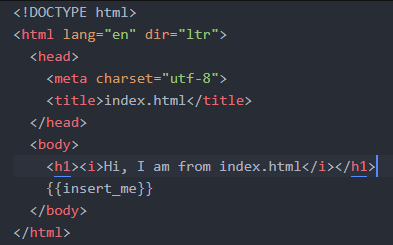
#### e/ Django Template

* Template have two parts:
  + The static part of HTML code.
  + The dynamic content we inject in. Which are template tags.
  + Notice that if we don’t get this clearly. We will get it through the course

1. Create a templates folder which in top directory: second\_app/templates/app\_two
2. **import** os, Use os.path.dirname, os.path.abspath, os.path.join to create link to TEMPLATES folder in settings.py. Set this as an variable. TEMPLATES\_DIR *(for convenient to add to TEMLATES later)* = os.path.join( )

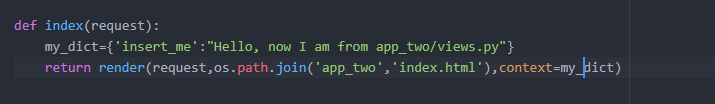
* Edit the key “DIRS”, append the directory to “DIRS”’s list value.

1. Create index.html inside templates folder. Put some html code in the *index.html .*Edit the title, add 1 deader “Hello, this is index.html”, add *{{insert\_me}}* ~~template tag~~  This is actually called variable – Django variable.
2. Com back to views. **From** *django.shortcuts* **import** *render* . change the **def** *index()* content to : *my\_dict = { ‘insert\_me’:’Heloo, I am* ***from*** *views.py’*}. return *render(request,’index.html’,context=’my\_dict’)\*

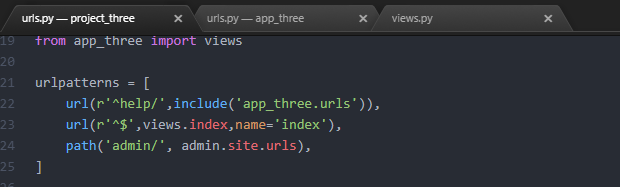


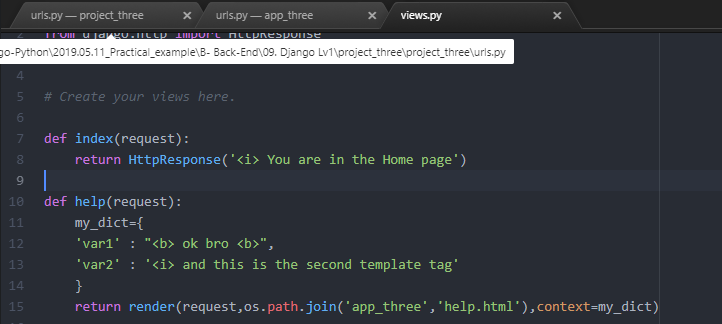
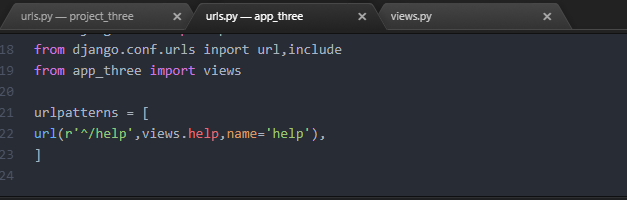
1. Change directory for *index.html* **from** *templates/index.html* to *templates/app\_two/index.html*. This is to distinguish index.html for each app.

* The render() the function here produce a string, not a html code like *HttpResponse(‘html\_code’)*



#### f/ Template Challenge.





#### g/ Django static files

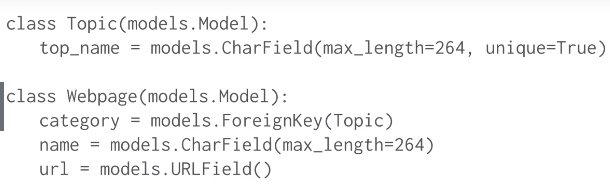
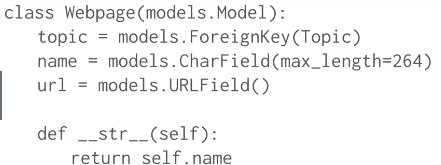
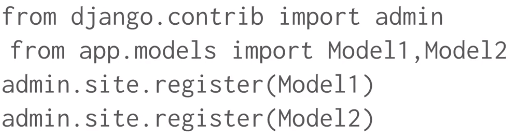
* To create static files, we need to create a new directory name static inside the project folder. The static folder will be linked in the settings.py file with a STATIC\_URL variable, just like template.
* Then if we want to add an image, again create a sub directory name images and put the images inside this images directory. To check if all of this worked, access *localhost/static/images/image.jpg (127.0.0.1.8000/static/images/image.jpg)*
* We use Django-tag:
  + *{% load staticfiles %}* at the top ò
  + *<img src={%static “images/[ic.jpg] %} />* for the img tag
  + *Django tags* - *{%%}* is used for more complex injections and logic
  + 

### 2/ Django level two

#### a/ Introduction

* Review the last Section (Django Level One)

#### b/ Django Models Overview

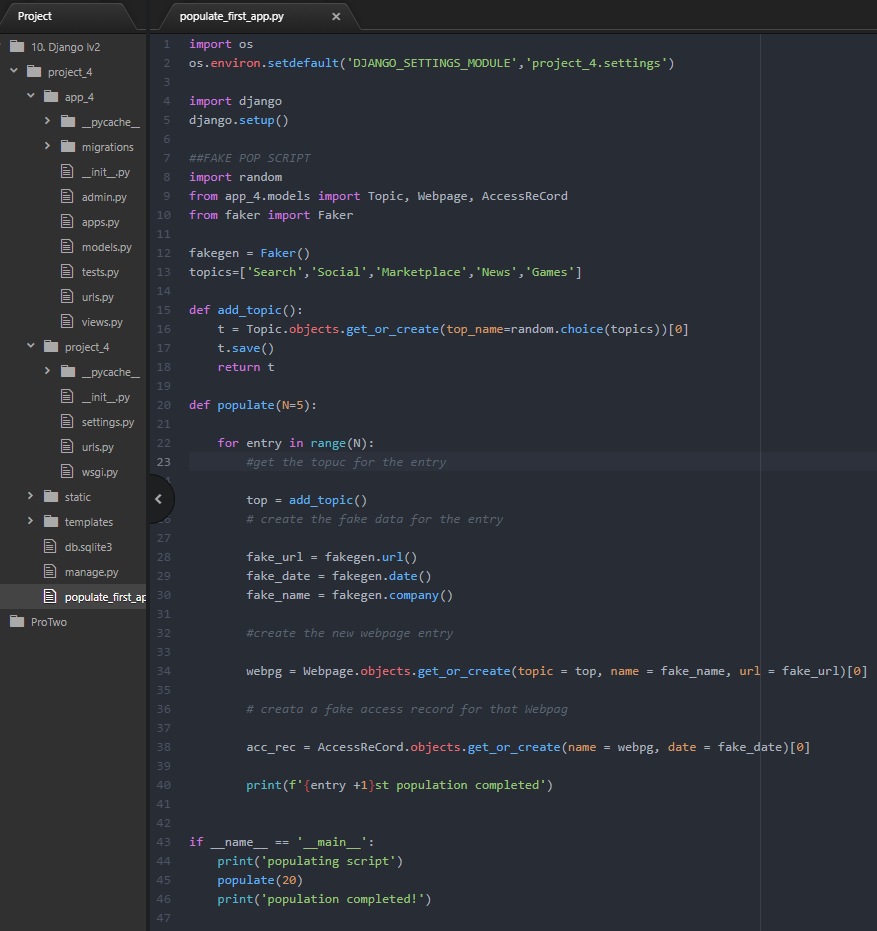
* This is about accepting information **from** user and retrieving information **from** database to generate content for the user.
* *Models* is used to incorporate with database in *Django Project*.
* *Django* come equipped with *SQLite* – which will work for simple example. *Django* can also connect to a variety of *SQL* engine backend.
* In settings.py we can edit the *ENGINE* parameter user for *DATABASES*
* To create a model, we use a **class** structure inside models.py of app:
  + This **class** object is a sub **class** of a Django’s built in **class** *django.db.models.Model*
  + Then each attribute of the **class** represents a field, which is just like a column name with constraints in *SQL*
* About *SQL*:
  + *SQL* operate like a giant table with each column representing a field and each row representing an entry.
  + Each column has a type of field such as *CharField, IntegerField, DataField*, etd.
  + Each field has constraints, for instant: *max-length constraint* indicating the maximum number of characters allowed.
  + *The tables (or models) relationship*: Models often reference each other.
  + *Foreign Keys* and *Primary Keys:*
    - *Tables*/*models* which reference each other have at least 1 common column with the same data.
    - The common column in the current *model*/*table* is *Primary* *Keys*. The common column in the other models/ table is the *Foreign* *Keys*.
    - Each *Primary* *key* is a unique identifier for each row in the models.
    - Each *Foreign* *Key* just denotes that the data coincides with a *Primary* *Key*.
* For instant of creating model:
  + First we setting up *models* classes:
    - 
    - 
    - Notice: **From** Django 2.x and above, there is one required \*\*kwargs: *on\_delete = \*argument* . This is a SQL Standard
      * *CASCADE*: When the referenced object is deleted, also delete the objects that have references to it (When you remove a blog post for instance, you might want to delete comments as well). SQL equivalent: *CASCADE*.
      * *PROTECT*: Forbid the deletion of the referenced object. To delete it you will have to delete all objects that reference it manually. SQL equivalent: *RESTRICT*.
      * *SET\_NULL*: Set the reference to NULL (requires the field to be nullable). For instance, when you delete a User, you might want to keep the comments he posted on blog posts, but say it was posted by an anonymous (or deleted) user. SQL equivalent: SET NULL.
      * *SET\_DEFAULT*: Set the default value. SQL equivalent: SET DEFAULT.
      * *SET(...)*: Set a given value. This one is not part of the SQL standard and is entirely handled by Django.
      * *DO\_NOTHING*: Probably a very bad idea since this would create integrity issues in your database (referencing an object that actually doesn't exist). SQL equivalent: NO ACTION.
  + After setting up *models*, we can migrate database to app:
    - Django can do this entire process with a single command: *python manage.py migrate*. (Entire process means creating the database with SQL)
    - Then, register the change to ( I think this should have been “from” (?) ) our app: *python manage.py makemigrations app1*
    - Next, run the command *python mange.py migrate* again to update the migration.
    - (Additional **from** internet): *Migration* are how Django store changes to our models – files.
  + To use the more convenient Admin interface with *models*, we need to register them to our application’s admin.py file:
    - 
    - *admin.site.register(model\_name)*
    - (Model1, model2 it name of the models we created, they can be any other name we set up)
  + To fully use the database and the Admin interface, we need to create a “*super user*”:
    - *python manage.py createsuperuser*
    - We need to provide a name, email and password to the *superuser*. We should write down or save them somewhere because we need to remember this them

#### c/ Creating Models

* Step in this lecture:
  + 1/ create first models
    - - Topic
    - Webpage (Have ForeignKey in Topic)
    - Accessrecord (Have ForeignKey name in Webpage)
  + 2/ Use shell to simulate Python P: *python manage.py shell*
  + 3/ Access the Admin site on browser

#### d/ Population Scrips

* This is some kind of dummy data which is fake data
* To do this populating, we need os, django, random, faker library. Faker wasn’t installed with Django or anaconda so we have to install it by ourself by:
  + Pip/conda install Faker
* Step 1 – Set up:
  + Os.environ.setdefault(‘DEJANGO\_SETTINGS\_MODULE’,’project\_name.settings’)  This code is to let Django know where the settings.py exists.
  + django.setup()  to load and populate Django’s application registry when some code is standalone
  + import faker.Faker or from faker import Faker
  + import all needed models
* Step 2 – Create populating fuction:
  + Class, Functions and methods to use:
    - Faker()  Class with own method which create fake date. Eg: fakegen =Faker(), fakegen.date()  generating fake date.
    - random.choice(list\_name)  chose 1 in the list
    - Model\_name.objects.get\_or\_create(field\_name = field\_value)[0]  Get\_or\_create return a tuple of 2 item, the first item is the field value we need  indexing[0]
    - Models\_instance\_name.save()  to save data injected to database. Actually we don’t need to use this as the get\_or\_create already have a built-in save()



*(?) Tại sao chỉ có Topic là phải t.save() còn những models khác lại không?*  *Answered*

#### e/ Models-Templates-Views Paradigm

* *MTV: Django operates on Models-Templates-Views*, which is called “*MTV*”.
* To serve dynamic content to user:
  + First: **Import** models we need to *views.py*
  + Second: Use the view to query the model for data that we will need
    - Use Model\_name.objects.order\_by(“Field\_name”) or Model\_name.object.all() to get all objects of a model.
    - Assign those object to a list variable then make that variable a value in dictionary.
    - Use dictionary to be context of render
  + Third: Paste results from the model to the template
    - First: Put an {% if list\_variable\_key\_name %} on top of data block
    - Use Variable tag {{ list\_variable\_key\_name.Field }}
    - To show each item of the objects list, use {% for acc in list\_variable\_key\_name \_name %}
    - Don’t forget {% endfor %} and {% endif %}
  + Fourth: Edit the template
  + Fifth: Map an url to view in *urls.py*

#### f/ Django level two project

* Requirements:
  + I/ Add a new model called User. It should have these fields:
    - 1/ First Name
    - 2/ Last Name
    - 3/ Email
  + II/ Create a view for website for the domain */users*:
    - This page will be an HTML list of user names and emails

### 3/ Django level Three:

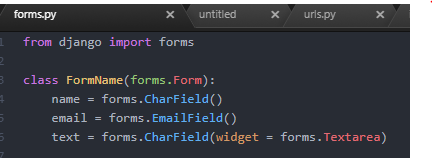
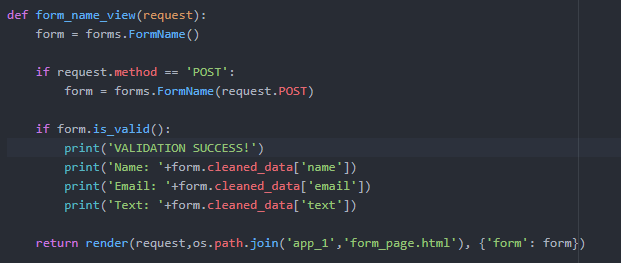
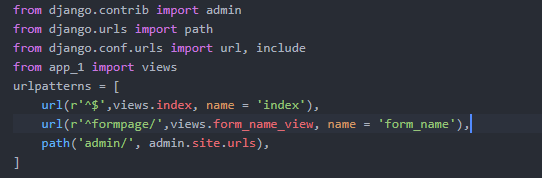
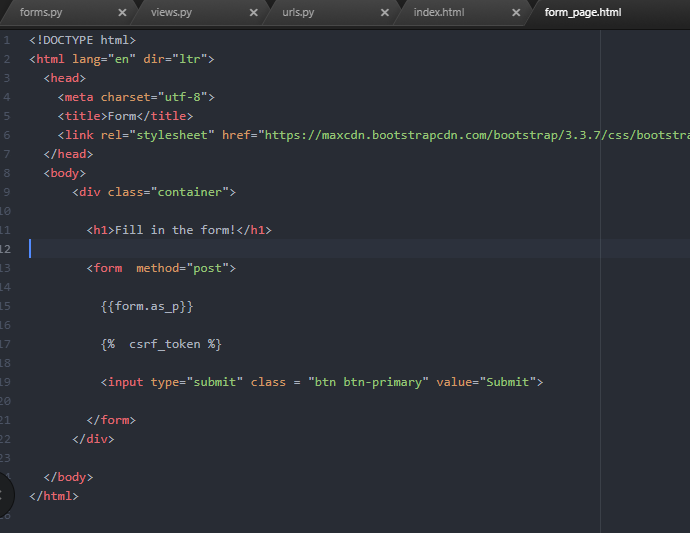
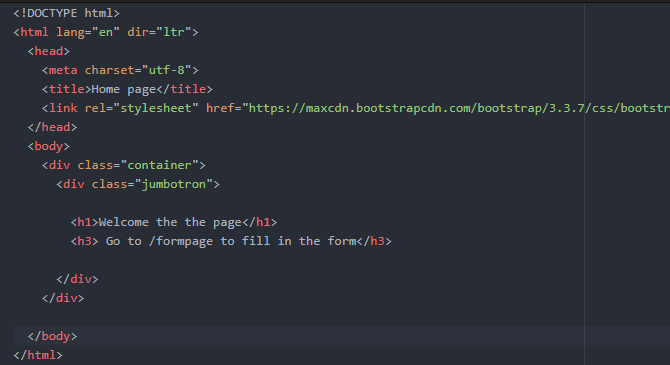
#### a/ Django Form

* Forms.py file
* Form.html
* Concept:
  + HTTP (Hypertext Transfer Protocol)
  + GET (Request data **from** resource)
  + POST (Submit data to be processed to a resource)
* *{% csrf\_token %}* Cross-site Request Forgery (Forgery- giả mạo):

#### 

* Note: Malware site’s script doesn’t have access to the anti- CSRF token because of the HTTP access control. So some user may still be able to be attacked as the allow control for every website just because they can’t use the API from another website. (?) *The last sentence is so abstract.*

#### b/ Forms code along

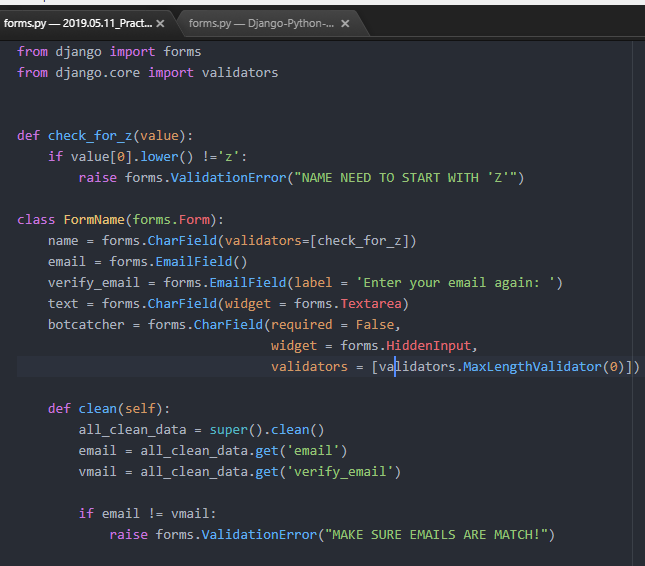
* Steps:
  + 1/ Set-up project, app, templates, a dir inside template for the app with index.html (home page) and form\_page.html
  + 2/ Create a *forms.py* inside app, **import** *forms* **from** *Django*, just like models
  + 3/ Create a new form :
    - 
    - New view: 
* Note for *request.method*:
  + request.method.- Contains the HTTP method used for the request (e.g. GET, POST).
  + request.GET or request.POST.- Contains parameters added as part of a GET or POST request, respectively. Parameters are enclosed as a django.http.request.QueryDict[[4]](https://www.webforefront.com/django/viewmethodrequests.html" \l "footnote-4)instance.
  + request.POST.get('name',default=None).- Gets the value of the name parameter in a POST request or gets None if the parameter is not present. Note default can be overridden with a custom value.
  + request.GET.getlist('drink',default=None).- Gets a list of values for the drinkparameter in a GET request or gets an empty list None if the parameter is not present. Note default can be overridden with a custom value.
  + request.META.- Contains HTTP headers added by browsers or a web server as part of the request. Parameters are enclosed in a standard Python dictionary where keys are the HTTP header names -- in uppercase and underscore (e.g. Content-Length as key CONTENT\_LENGTH).
  + request.META['REMOTE\_ADDR'].- Gets a user's remote IP address.
  + request.user.- Contains information about a Django user (e.g. username, email) linked to the request. Note user refers to the user in the django.contrib.auth package and is set via Django middleware, described later in this chapter.
    - Map to *urls.py*, (?) the name =’’ is quite different – Solved: This name is not really matter here, it will make impact in the next section which related to url Django-tag. 
    - form\_page.html: 
    - *index.html:* 
* .*is\_valid()*:
  + is\_valid is a method that check if some data is valid (có hiệu lực – acceptable) and return Boolean value.
  + automatically call the *clean()* function. *clean()* is used in the forms.py file while *.is\_valid()* is used in views.py. This *clean()* function return the *cleaned\_data*.
  + By default, all Django model fields have required parameter value = True. So, sometime we have to overwrite it to required = False or blank = True for fields that may not have data input by user.

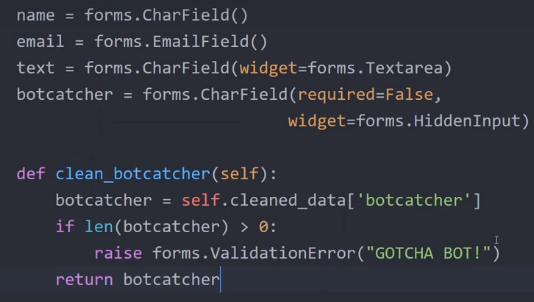
#### c/ Form Validation

* The form we set up is not only available for user but also for bots. So Django have some built-in validators which can be conveniently used to validate our forms (or check for bots)
* Everything above happen within the forms.py file. We will have below:
  + Adding a check for empty fields
  + Adding a check for a “bot”
  + Adding a clean method for the entire form
* Content:
  + 1/ Create a bot catcher function?
  + 2/ Replace the bot catcher by a validators inside
  + 3/ Create a validator that check whether a str starts by ‘z’. This is a function to be added to validators attributes.
  + 4/ Create a validator that check for verification of email by the clean\_ method. *clean()* lone will for the whole form. Those clean\_r’^anything’ will be found by Django and automatically be run.

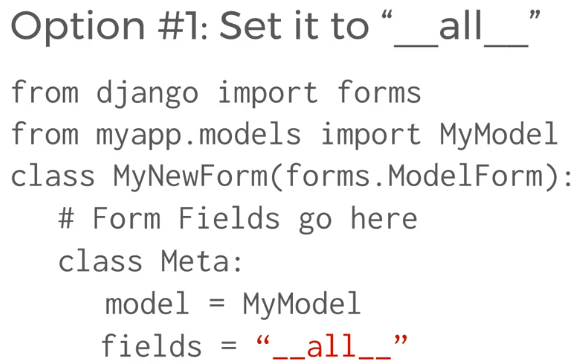
*(?) 1/ What does “clean” mean? – Just like confirmation of data submitted ? I am still confused on this context.*

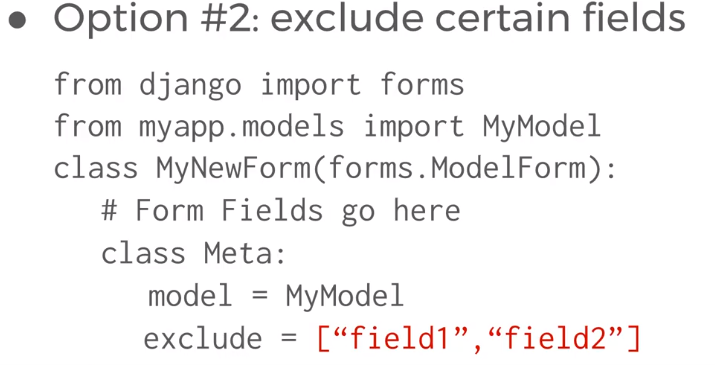
*(?) 2/ What is super()*  *this function create a instance of the parent* ***class*** *of the current class. So that we can use the parent class’s methods.*

* + Notice: This lasted version of Django have another different in syntax : *all\_clean\_data.get()* instead of *all\_clean\_data[]*
  + 
  + Another way to create validator: *clean\_whatever\_name\_of\_field\_we\_are\_checking(self)*  Django will automatically look for the *clean\_$* method and look for the *$* if it match any of the form’s field, then check it



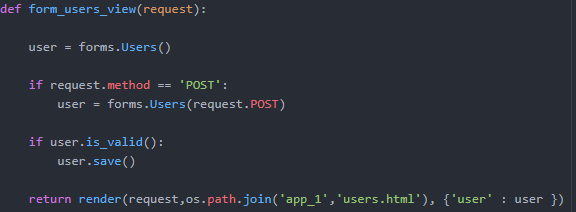
#### d/ Model Forms (forms.ModelForm):

* Instead of using inheritance form *forms.Form*, we use *forms.ModelForm* sub class
* Meta **class** –inline class?
* The field is match up **from** model. But is we want to use some custom validators, then we have to paste in those field where we want to use custom validator
* Notice: Check out the documentation for h
* ow to connect forms to models.
* 





#### e/ Django exercise – Save forms data to models

* 
* Is this exercise, we are creating a form that can collect data **from** user input and paste it into models, save the models data to database.

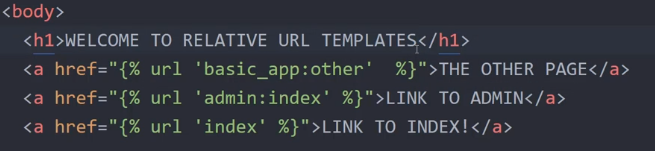
### 4/ Django level four

#### a/ Relative URLS with Templates

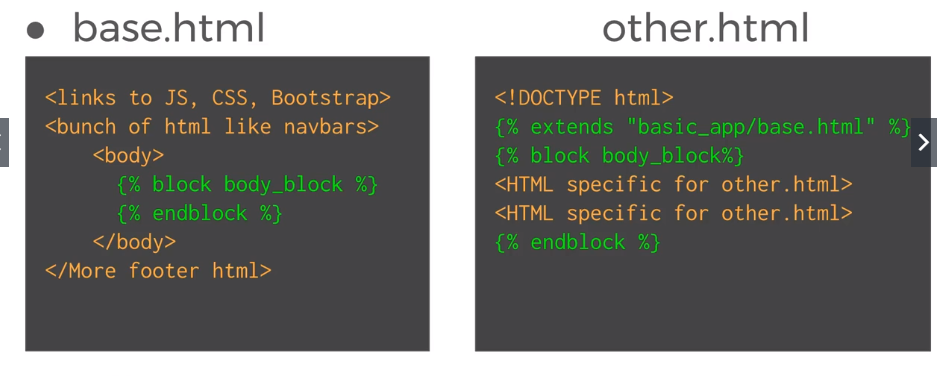
* Usually we have Django-variable and Django-tag for models, forms, static file/link, view. This URLs Template Tag is for calling url **from** urls.py

(?) Name is actually in the urls.py?

#### b/ Code:

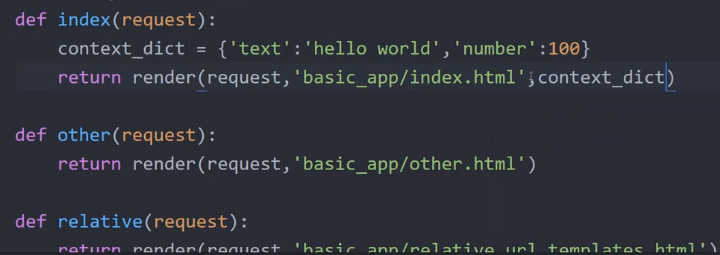
* 2 key things to remember here are that we need to a variable for our app\_name and name it lo create (?)ok for in urls.py file

#### c/ URL template inheritance

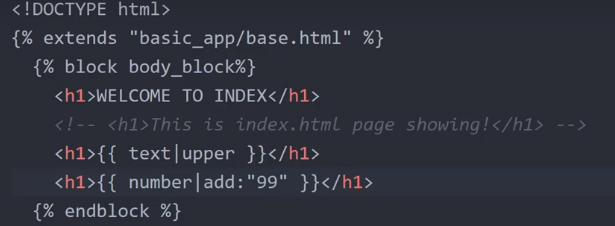
* This is sometime known as template extending
* 
* (?) Can we have multiple block in the html file?  Try search inclusion-tags, maybe their air something usefull

#### d/ Filter:

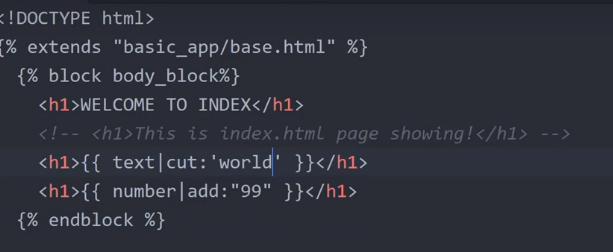
* Use built-in filter and build an own filter
* First: Add a context dictionary to call data in some Django-variable – {{}}



* Second: Call string we created in Django-variable. Also call some filter here:



* Third: Create a cut filter by original way (the commented code in the next screenshot.



-Fourth: Use the decorators (?) I haven’t feel the sense of this yet

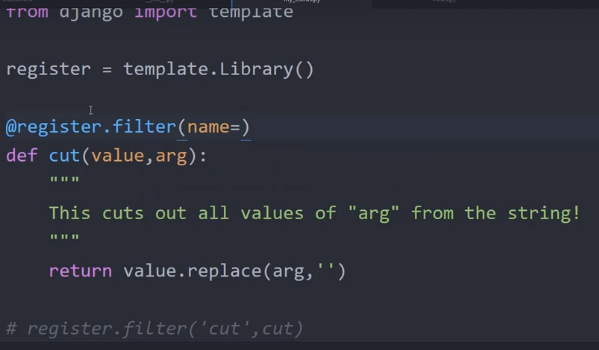
*@new\_decorator*

***def*** *decorator\_func():*

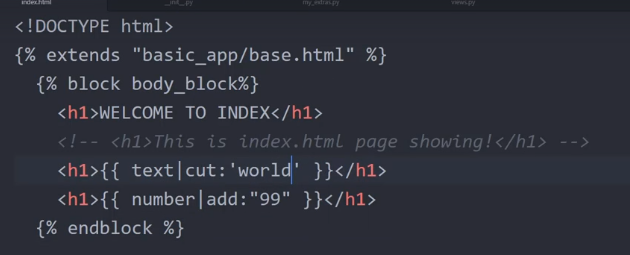
*content*

Is equivalent to:

*decorator\_func = new\_decorator(decorator\_func))*



* Fifth: Again, check if the new filter we created work



* (!) Don’t forget to check out the other filters in documentation, Jose said that even we can do most of those stuff with Python code in the back end but we should check it out.

### 5/ Django Level 5:

#### a/ Django passwords

* In this part, we are talking about setting up for User Authentication.
* First thing we need to take care of is setting up our ability to authenticate a User.
* There’re some app would help us, make sure they are under the INSTALLED\_APPS list in the settings.py:
  + “django.contrib.auth:
  + “django.contrib.contenttypes”
* Next thing is to make sure we save the password safely, (!) **never save them as plain text**. We use a PBKDF2 algorithm with SHA-256 hash that built in to Django.
  + This SHA-256 is developed by National Security Agency- or NSA -. WE can try getting some hash code on <https://www.xorbin.com/tools/sha256-hash-calculator>.
  + The overall understanding is that whenever we pass in a data, SHA-256 encrypts it to a hash code. We save the hash code and then check if the hash code is matched, not check the direct password.
  + Like any security methods else, those are not totally secured but it’ll take a massive computing power to crack it.
  + But if we want more security, we can upgrade even more secure hashing algorithms, just have a google search, it’s not difficult
* So we will use 2 libraries: bcrypt and Argon2 by:
  + pip install bcrypt
  + pip install django[argon2]
* We need to pass in the list PASSWORD\_HASHERS to try in the order we want to try them. If for some reason we don’t have the library support, eventually we’ll fail back on the original PBKDF2. :

PASSWORD\_HASHERS = [

'django.contrib.auth.hashers.Argon2PasswordHasher',

'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',

'django.contrib.auth.hashers.BCryptPasswordHasher',

'django.contrib.auth.hashers.PBKDF2PasswordHasher',

'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',

]

* + Those are list of Hasher which Django is gonna try. If the first have error then Django will try the second, if the second have error, etc. (?) which error?, checking fail or ?
* We can also add in validator to prevent user from using too weak password such as “abc123”. Each validator will check for a condition. But be notice that it’ll be very annoying for password have too much rule like having to be change every 6 months, with a massive combination of character and symbol.

AUTH\_PASSWORD\_VALIDATORS = [

{'NAME': django.contrib.auth.password\_validation.UserAttributeSimilarityValidator', },

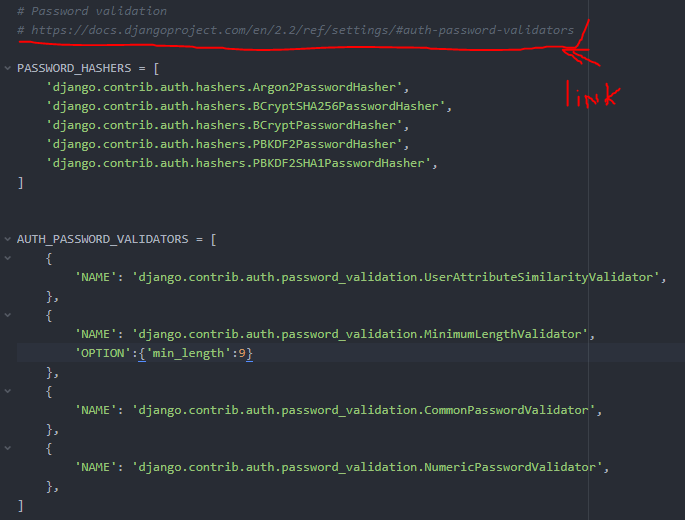
{'NAME': 'django.contrib.auth.password\_validation.MinimumLengthValidator', },

{'NAME': 'django.contrib.auth.password\_validation.CommonPasswordValidator', },

{'NAME': 'django.contrib.auth.password\_validation.NumericPasswordValidator', },

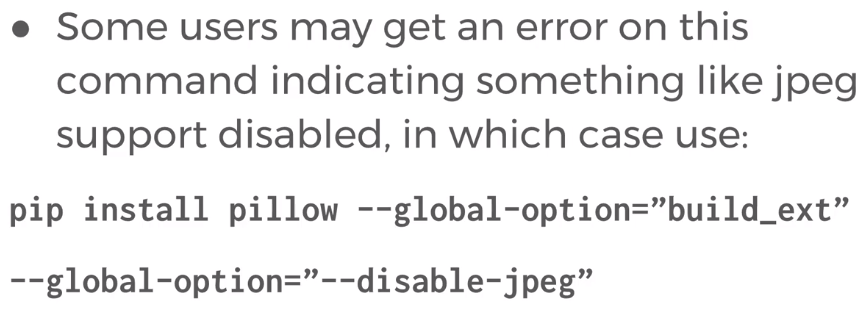
]

* We add specific parameter for each option by adding as below: ‘’OPTION’: {‘min\_length’: number},}.
* (!) Check for more about password in the link

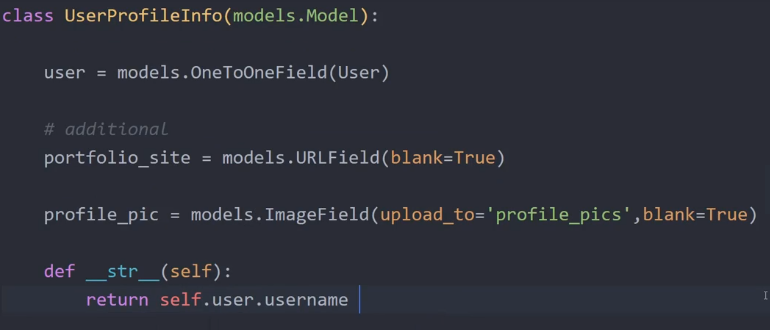
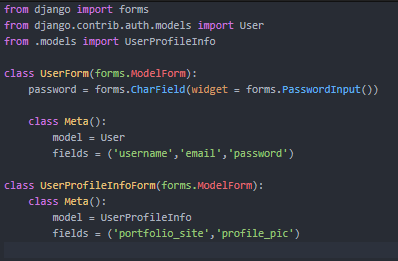


* In this lecture, Jose mentioned a new directory which is media directory, beside template, static directory.

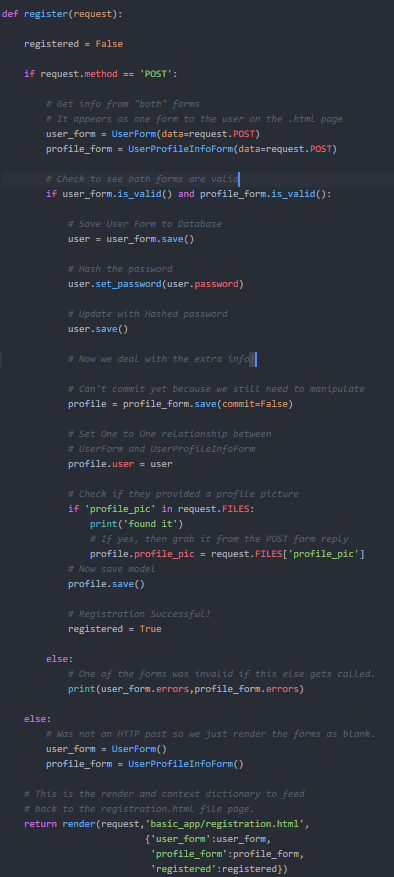
#### b/ User Models

* In this lecture we are going to discuss about User Authorization Models and how to set up media files in our project.
  + Note that this User Authorization is different from the previous example we created in section 3.
  + Django have 2 built-in models which are User and Group. They are the Authentication and Authorization Models, try not to over write them.
    - (?) Dig deeper about User model and Group model.
  + This User object have a few key features:
    - Username
    - Email
    - Password
    - Frist name
    - Surname
  + A few attribute:
    - is\_active:
      * **Active** users are users that you can identify (by an id, email or username) who interact with your business during a specified period.
    - is\_staff
    - is\_superuser
    - etc (? I do it have more built-in attribute?)
    - We can also create more attribute by creating other classes that have relationship with this User class in models.py
  + New syntax/code:
    - New Field: OneToOneField()
    - URLField(blank = TRUE) (?)- Blank ?
* In order to work with images in Python, we have to install the Python Imaging library – pillow library:
  + Pip install pillow
  + Sometime we may get an error say indicate something like jpeg support disable, let’s try this
    - 
* So thing like CSS, JS, Image, etc are stored by us in static\_root. Then things uploaded by users will be stored in media\_root after uploading. And as this is uploaded form user, we need to set up a form to support the procedure.

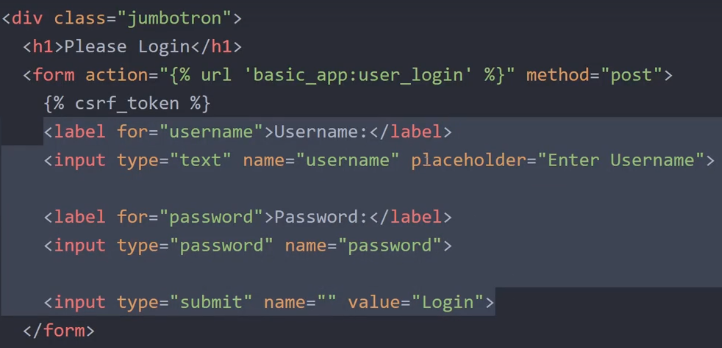
#### c/ Creating User Model and User ModelForm:

* Actually, we’ll have 2 model:
  + User with built-in attributes and
  + User\_AdditionName to add some additional attribute that we need.
  + The 2 models connect to each other by user = OneToOneField(User) as a field in User\_something.
* There’re 2 forms match-up with those model which are
  + UserForm and
  + User\_somethingForm
  + Don’t forget to inherit fields we need from User by fields = Tupple ()
* Step 1:
  + In models.py:
    - import *models* from *Django.db*
    - import *User* model from *Django.contrib.auth.models*
    - Create additional fields for by creating *UserProfileInfo*:
      * We sometime want to add more fields beside the existed field like username, password, etc…
        + 
        + Remember that the User form name should be different from the Model. For instance: UserProfileInfoFrom. Jose admit made a mistake here:
        + OneToOneField: as described above. Google for more information.
        + blank = TRUE mean, this field is able to be blank (Not re quired to be provided)
        + upload\_to = ‘profile\_pics” mean the content will be uploaded to profile\_pics subfolder of media folder
        + Don’t forget to register this new model UserProfileInfo in admin.py
    - Create UserForm and UserProfileInfoForm by inheriting from the above:
      * 
      * Widget: (!)
      * (?) why do we need to redefine password? Is it because
      * forms.PasswordInput() ? <<< because originally, password is just an attribute and was shows. We need to redefine to add attribute like widget = forms.PasswordInput() <<< to hide character when user is typing password
* Step 2: Create html files to prepare for latter registration and login logout:
  + I think we just should create the file and the inherit code in base.html. The other code related to registration and login we should crate after creating registration, login, logout necessary codes.
  + base.html:
    - This html does not appear anywhere in our url mapping.
    - The purpose of base file with inheritable blocks of html code to inherit to the other html files.
    -  I have the answer for the question that if we can have multiple blocks. And it’s so easy... fuck me. Fuck my laziness.
  + index.html: Our home page.
  + login.html: Our login page
  + registration.html: the Registration page.
* Step 3:
  + Create Registration view, url mapping.
  + See in the below part d/ Registration

#### d/ Registration

* Step: Create register view:
  + Mình sẽ test kỹ chỗ natfm tại sao lại phải lưu 2 variable mới vào 2 form.save() ? Tại sao sau khi save rồi mới gắn user field(OneToOneField)?. (ok)  Vì we can only interact with database through models
  + Theo lý thuyết mihf phân tích thì chỉ cần save là variable cũ có thể truy xuất được rồi cơ mà ?? (ok)  Vì we can only interact with database through models
  + Và theo lý thuyết thì variable mới chỉ để cho ngắn lại chứ nó hoàn toàn dung để truy xuất giá trị saved. (ok)  Vì we can only interact with database through models
  + À sao lại gán với giá trị form.save() chứ không gán vô form luôn khi mà save là đã truy xuất được rồi. (ok)  Vì we can only interact with database through models
  + Commit = False tức là sao )). Thấy bảo là để còn edit được. Tức là edit rồi nó tự động save, hay không save luôn ?
  +  data = request.POST would be what is called an assignment statement in Python. It is assigning the value of request.POST to an object called data. The other merely the object itself. If you want to know what they mean in the context of Django we need to know what lecture you are referencing. Please provide the lecture and time. If this is from the lecture's code please provide the full path, filename, and line number that you are referring to.  Still confused with data = request.POST. I deleted the data = part and everything remain the same result with no exception.
  + For the twice apprar of .save() method: Can not take date directly from Form. Form data is used to upload to server only. .save() return the models that can be callable, as well as some attribute, method that form does not have, ig set\_password (?)
  + The set\_password encrypts password in database to hash code. Without set\_password, password data will be an error and are not saved in database.
  + profile.profile\_pic = request.FILES(‘profiles\_pic”)  everytime we have a form that takes in any form of file input such as an Image or a file, it automatically stores in a dictionary with a key given earlier
  + Remember to update understanding about image in UserInfoForm.  I was struggling as the picture wasn’t saved in my code. They were saved well in Jose’s files Ok solved. In Html file, every space count :v. Detail as below:
    - Note that request.FILES  will only contain data if the request method was *POST* and the *<form>* that posted the request has the attribute *enctype="multipart/form-data".*  handling form with file upload Otherwise, request.FILES will be empty.
  + .errors attribute: show the error we are having if some data is not valid:
    - 
  + 

#### e/ Logins

* Creating a login/logout functionality invovles:
  + Setting up the login view:
  + Using built-in decoratiors for access
  + Adding the LOGIN\_URL in settings.py
  + Creating the login.html
  + Edditing the urls.py files
* Django 2.0 removed the django.core.urlsresolvers module and replace by the django.urls. Some features in django.core.urlresolver was also removed.
* LOGIN\_URL:
  + This is the URL or named URL pattern where requests are redirected for login when using the @login\_required() decorator.
  + 
* In login.html:
  + Why do we need to use <form action ={% url ‘appPassword:user\_login‘ %} ? Isn’t it automatically run as the page is run from view in views.py? I’ll test this (!). 
  + 
* In views.py file:
  + We import below:
    - **from** django.contrib.auth **import**
      * authenticate  just verifies the login information
      * login  take the user(aboved-authenticated) object and set the cookies.
      * Logout  take in a request.
    - **from** django.http **import**
      * HttpResponseRedirect  redirect browser to an URL (not including hosting name)
      * HttpResponse  return some html code
    - **from** django.urls **import**
      * reverse: to call the url in urls.py by take in value of attribute “name” (reversing name to url)
    - **from** django.contrib.auth.decorators **import**
      * login\_required  a decoration to require logined. This concept is related to LOGIN\_URL in settings.py
  + Create user\_login view:
    - 
    - request.POST.get(‘data\_name’) <<< get the data.
  + Create user\_logout view:
    - Don’t forget to put in the decorator @login\_required
    - 
  + Add the login/logout button in base.html’s navbar:
    - {% if user.is\_authenticated %}  To check if the user is authenticated or not. We can also uses {% if user.is\_active %}. is\_authenticated is a read only boolean attribute return True if then user has been authenticated
    - 