# Setting up django

## Route the path (if needed)

$condaPath = "C:\Users\<YourUsername>\miniconda3\condabin"

[Environment]::SetEnvironmentVariable("Path", $env:Path + ";$condaPath", "User")

## Ready the powershell if still didnot work after conda activate my\_environment

conda init powershell

## to check the list of env that I created

conda info --envs

## to activate env

conda activate my\_Env

## To deactivate env

conda deactivate my\_Env

# Installing django

## command lines

1. conda install django

Or **for normal python distributions**

2. pip install django

Should use **django-admin tool** which is already installed along with Django in the first place.

**django-admin**

## To create project

django-admin startproject first\_project 🡨project name

# About the files that appear after creating project.

## First file(\_\_init\_\_.py)

This file lets's the python know that this directory can be treated as a package due to it's name \_\_init\_\_.py and it is also blank Python script.

## settings.py

This is where you will store all your project settings

## urls.py(Expect to use a lot of usage of regular expression)

This is a files for storing all the URL patterns for project.So it is just storing all the different pages of web application and how they should connect to the end user.

## wsgi.py

This is a python script that acts as the web server Gateway interface and help us to deploy the application to production.

## manage.py

This is a file that will be using a lot of the time. It will be associates with many commands as we build our web app!

## To run django web project on server

**Use this commands** - python manage.py runserver

And it will show a bunch of stuff and at the buttom it will show someting like this:

Django version (version), using settings

'first\_project.settings'

Starting development server at

http://127.0.0.1:8000/

**And to run in browser copy and paste that url in browser.** http://127.0.0.1:8000/

## Migration warning

This has to do with database and how to connect them to Django

# So what is migration?

It's allow user to mover databases from one design to another and this is also reversible.

So you can "migrate" your database.

## Create app in Django

python manage.py startapp myapp <--- name

**To get http stuff working you need to import below two in view.py of app's diratory**

from django.shortcuts import render

from django.http import HttpResponse

## step by step of letting django know that I have created app.

**step1** go to settings.py of project folder.

**step 2** find install\_app and add 'appname'. eg --'first\_app'

Creating view

### In view.py

from django.http import HttpResponse

### Mapping

Visualizing the Flow

Here’s the request-response flow in simplified steps:

Browser Request → Django URL Dispatcher (urls.py)

Match Found → Django calls the mapped view function (views.py)

View Function → Processes the request and returns a response.

Response → Django sent back to the browser.

User Enters URL:

http://localhost:8000/ → Goes to the home view.

http://localhost:8000/menu/ → Goes to the menu view.

http://localhost:8000/order/ → Goes to the order view.

What Happens:

urls.py maps the request to the appropriate view.

The view function processes the request and sends an HttpResponse.

What the User Sees:

For / → "Welcome to our restaurant! Please see our menu."

For /menu/ → "Menu: Pizza, Pasta, Salad, Dessert."

For /order/ → "Thank you for placing your order!"

Summary

The URL dispatcher (urls.py) is like the restaurant host who routes you to the correct table (view).

The view function (views.py) is like the waiter who takes your request and responds appropriately.

# So the flow of url mapping and views

**userက requestလိုက်ပြီဆိုရင် အဲ့requestကurlဆီအရင်ရောက်တယ် project fileအောက်က url.pyကို**

**အဲ့ကနေမှ url dispatcherကကြည့်ပြီးတော့ request ဝင်လာတဲ့urlနဲ့ သူ့ထဲမှာရှိလားတိုက်စစ်တယ်**

1. **မရှိရင် ၄၀၄ errorပို .**
2. **ရှိရင် သူ့နောက်မှာရှိတဲ့ views functionကိုခေါ်ပေးတယ် app fileအောက်က view.py ကို.**

**ပြီးရင်အဲ့viewထဲက functionကို executeလုပ်ပြီး http response or rendered template ပြန်ပိုပေးတယ်**

**ပြီးရင် အဲ့responseကို djangoက userရဲ့ browserဆီပိုပေးလိုက်တယ်.**

# Mapping url

## Include() function from django.conf.urls

* from django.conf.urls import include

**path('first\_app/', include('first\_app.url')),**

# Common necessary import for url.py in project

* from django.contrib import admin
* from django.urls import path
* from django.conf.urls import include
* from first\_app import views
* urlpatterns = [  
   path('', views.index, name='index'),  
   path('testing/', include('first\_app.url')),  
   path('admin/', admin.site.urls),  
  ]

# Common necessary import for url.py in app

* from django.urls import re\_path
* from first\_app import views
* urlpatterns = [  
   re\_path(r'^$', views.index, name="index")  
  ]

# Common necessary import for views.py in app

* **from django.shortcuts import render**
* **from django.http import HttpResponse**
* # Create your views here.  
    
  def index(request):  
   return HttpResponse("Hello World")  
    
  def test\_include(request):  
   return HttpResponse("Hello World included")

# Templates in Django

* Templates are a key part to understanding how django really works and interacts with my website. Connecting templates with models to display data created dynamically.
* Template will contain the static parts of an html page(parts that are always the same)

## Template tags

They have their own special syntax. These syntax allows you to inject dynamic content that your Django App's views will produce, effecting the final HTML.

## step by step of letting django know that I have created template in setting.py of peoject dir.

**step1** go to settings.py of project folder.

**step 2** find Template dir and edit inside 'DIR key'

## DIR key hard coded path problem solution

**But since DIR key require hard-coded path, it will be really hard for user from different OS can have difficult time. But can be solved with python’s os module which can dynamically generate the correct file path strings, no matter what type of computers.**

**Import OS**

**Print (\_\_file\_\_)**

**Print (os.path.dirname(\_\_file\_\_))**

### Feeding dir key inside of the template dictionary with os module

Step 1 go to setting.py at project folder and import os. Than type:

BASE\_DIR = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))

TEMPLATES\_DIR = os.path.join(BASE\_DIR,"templates")

import os

N2BASE = os.path.abspath(\_\_file\_\_)

N1BASE = os.path.dirname(os.path.abspath(\_\_file\_\_))

BASE\_DIR = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))

print(\_\_file\_\_)

**Output** C:\Users\USER\PycharmProjects\Giraffel\first\_project\first\_project\settings.py

print(N2BASE)

**Output** C:\Users\USER\PycharmProjects\Giraffel\first\_project\first\_project\settings.py

print(N1BASE)

**Output** **C:\Users\USER\PycharmProjects\Giraffel\first\_project\first\_project**

print(BASE\_DIR)

**Output C:\Users\USER\PycharmProjects\Giraffel\first\_project**

## Recommended Modern Alternatives way with pathlib

**from pathlib import Path BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent**

**# Build paths inside the project like this: BASE\_DIR / 'subdir'.**

**TEMPLATES\_DIR = BASE\_DIR / 'templates'**

## After solving template hard coded issue

* Create template file at the same level with view.py inside app folder.
* Inside that create the file with app name again and create html file

**Django Template Variables**

<body>  
 <h1>Hello World</h1>  
 {{insert\_me}}  
</body>

def templates(request):  
 my\_dict = {'insert\_me': 'Now I am coming from first app index.html!'}  
 return render(request, 'first\_app/index.html', context=my\_dict)

These will also inject content into HTML directly from Django.

## Returning other types of media with template

Create folder "Static", inside create images folder. "Static" folder is not only for media but also for css and javascript.

### For static media

**setting mhar add below lines**

STATIC\_DIR = BASE\_DIR / 'static'

After that create variable and list also inside **setting.py which is within project folder not app**

STATIC\_URL = '/static/'

STATICFILES\_DIRS = [

STATIC\_DIR,

]

Always put under Doc type html file.

{% load static %}

<img src="{%static "images/pic.jpg"%}" alt="Uh Oh, didn't show!">

### For static Css

Create folder "Static" inside create CSS/mystyles.css

To use CSS in HTML

<link rel="stylesheet" href="{% static "CSS/mystyles.css" %}"/>

### Summary Level 1 set-up work flow

django-admin startproject first\_project

cd first\_project

python manage.py startapp first\_app

# Django level 2 (models)

Modals are the importants part of allowing django to incorporate with a database into a Django Project. Django comes with SQLite to work for simple examples but it can also connect to PosrgreSQL, mySQL by switching the actual engine.

You just need to edit the ENGINE parameter used for DATABASES in setting.py file to change database engine.

## Creating model

To create model we use class structure inside of the relevant application models.py file

This class object will be a subclass of Django's built-in class:

**django.db.models.Model**

Each attribute of the class represents a field, which is just like a column name with constraints in SQL.

### Eg of creating models in model.py of app

# Create your models here.

class Topic(models.Model):

    top\_name = models.CharField(max\_length=264,unique=True)

    def \_\_str\_\_(self):

        return self.top\_name

class Webpage(models.Model):

    topic = models.ForeignKey(Topic,on\_delete=models.CASCADE)

    name = models.CharField(max\_length=264,unique=True)

    url = models.URLField(unique=True)

    def \_\_str\_\_(self):

        return self.name

class AccessRecord(models.Model):

    name = models.ForeignKey(Webpage,on\_delete=models.CASCADE)

    date = models.DateField()

    def \_\_str\_\_(self):

        return str(self.date)

## Working with manage.py to migrate

**python manage.py migrate**

After that

python manage.py makemigrations **app1** <--App name

Then

**python manage.py migrate**

So now we can use the shell from the manage,py file to play around with the models

### Testing this work or not with the shell from the manage,py file (in commands)

1. **Python manage.py shell**
2. **from first\_app.model import Topic <--class which is in model.py of app**
3. **print(Topic.objects.all())**
4. **<QuerySet []>**
5. **t = Topic(top\_name = "Social Network")**
6. **t.save()**
7. **print(Topic.objects.all())**
8. **<QuerySet [<Topic: Social Network>]>**
9. **quit()**

## Registering these models to our application's admin.py file.

To using more convenient with the admin interface with the models. We need to register them to our application's admin.py file.

You can do that with this code

from django.contrib import admin

from app name -->app.models import Model1,Model2,Model3 etc eg. Topic,Web

admin.site.register(Model1)Topic

admin.site.register(Model2)Web

admin.site.register(Model3)AccessRecord

**In order to fully use the database and the Admin, we will need to create a "superuser"**

## Creating superuser command

**python manage.py createsuperuser**

# Common necessary import for model.py in app

**from django.db import models**

### Example creating database with class in model by using OOP

# Create your models here.

class Topic(models.Model):

    top\_name = models.CharField(max\_length=264,unique=True)

    def \_\_str\_\_(self):

        return self.top\_name

class Webpage(models.Model):

    topic = models.ForeignKey(Topic,on\_delete=models.CASCADE)

    name = models.CharField(max\_length=264,unique=True)

    url = models.URLField(unique=True)

    def \_\_str\_\_(self):

        return self.name

class AccessRecord(models.Model):

    name = models.ForeignKey(Webpage,on\_delete=models.CASCADE)

    date = models.DateField()

    def \_\_str\_\_(self):

        return str(self.date)

# Using fake data to populate models in Django with faker Library

pip install Faker (can read documantation in Faker.readthedocs.io)

Some clarification for below code

topics = ['Search', 'Social', 'Marketplace', 'News', 'Games']

def add\_topic():

t = Topic.objects.get\_or\_create(top\_name=random.choice(topics))[0]

t.save() # Saves the record (if it was created).

return t

get\_or\_create make the data return with tuple form which is (<Topic: News>, True) that why these is [0] at the end. True if data does not exist in data base so new column data needed to create. False if the data is already exist.

To run python populatefilename.py 🡨--

### Example:

* **Initial State**: The Topic table is empty.
* **First Call to add\_topic()**:
  + Suppose random.choice(topics) selects 'Games'.
  + Since no Topic with top\_name='Games' exists, create adds a new record: {id: 1, top\_name: 'Games'}.
* **Subsequent Calls to add\_topic()**:
  + If random.choice(topics) picks 'Games' again, get retrieves the existing record instead of creating a duplicate.
  + If it picks 'Search', it creates a new record: {id: 2, top\_name: 'Search'}.

# Model.Templates-View Paradigm(MTV)

**First**: In the views.py file we import any models that we will need to use.

**Second**: Use the view to query the model for data that we will need

**Third**: Pass results from the model to the template

**Fourth**: Edit the template so that it is ready to accept and display the data from the model

**Fifth**: Map a URL to the view

# .objects is the default manager

In Django, .objects is the **default manager** that is automatically added to each model class, allowing you to interact with the corresponding database table.

**AccessRecord.objects** is the default manager for the AccessRecord model.

### ****Summary of Common Operations****:

1. **Query**: .all(), .filter(), .exclude(), .get()
2. **Order**: .order\_by()
3. **Limit**: .first(), .last(), .count(), .exists()
4. **Aggregate**: .aggregate(), .annotate()
5. **Create**: .create()
6. **Update**: .update()
7. **Delete**: .delete()
8. **Raw Queries**: .raw()
9. **Retrieve Specific Fields**: .values(), .values\_list()

# Django Level 3

# Common necessary import for form.py in app

From Django import forms

Class formname(forms.Form):

name = forms.CharField()

email = forms.EmailField()

text= forms.CharField(widget=forms.Textarea)

## Forms in views of App

From . import forms 🡨 form.py in app or from Appname.form import formname

def form\_name\_view(request):

form = forms.FormName()

return render(request,'form\_name.html',{'form':form})

when in HTML

{{form.as\_p}}

{%csrf\_token%}

# Using post to grab the data from form

def form\_name\_view(request):

    form = FormName()

    if request.method == 'POST':

        form = FormName(request.POST)  # Corrected typo here: FormName

        if form.is\_valid():

            print("VALIDATION SUCCESS!")

            print("NAME: ", form.cleaned\_data['name'])  # Fixed concatenation

            print("EMAIL: ", form.cleaned\_data['email'])

            print("TEXT: ", form.cleaned\_data['text'])

    return render(request, 'AppThree/form.html', {'form': form})

### Step by step Explanation of above code

Note: request from def form\_name\_view itself is a GET method.

### ****GET Request (Initial Page Load)****

* When the user first **accesses the URL** (e.g., /form/), Django receives an **HTTP GET request**.
* In your view function:

python

Copy code

def form\_name\_view(request):

form = FormName() # Empty form instance

return render(request, 'AppThree/form.html', {'form': form})

* + request.method is "GET", so the condition if request.method == 'POST': **does not execute**.
  + A new **empty form instance** (FormName()) is created.
  + This form is sent to the template and rendered as empty fields for the user to fill out.

### ****POST Request (Form Submission)****

* When the user fills out the form and **clicks Submit**, the browser sends the form data back to the same URL using an **HTTP POST request**.
* Now, in the same view function:

python

Copy code

if request.method == 'POST':

form = FormName(request.POST) # Bind submitted data to the form

* + request.method becomes "POST", so this block **executes**.
  + The submitted form data (request.POST) is passed to the form class FormName() to **bind** the data.
  + Django automatically validates the form fields.

### ****Validation and Response in views of App****

* If the form is valid:

python

Copy code

if form.is\_valid():

print("VALIDATION SUCCESS!")

print("NAME: ", form.cleaned\_data['name'])

* + The submitted data is cleaned, validated, and stored in form.cleaned\_data.
* Regardless of success or failure, the view **re-renders** the form:

python

Copy code

return render(request, 'AppThree/form.html', {'form': form})

* + If there are errors, they are displayed alongside the form fields.

Difference between Django project and Django Application

# Form Validation

## Bot catcher

Step 1 Adding a trap for bot to fill the value in one input by adding hidden input box in form.py file in app

class User\_input(forms.Form):

    name = forms.CharField(max\_length=245, required=False)

    email = forms.EmailField( required=False)

    text = forms.CharField(widget=forms.Textarea)

    botcatcher = forms.CharField(required=False,widget=forms.HiddenInput)

    def clean\_botcatcher(self):

        botcatcher = self.cleaned\_data['botcatcher']

        if len(botcatcher)>0:

            raise forms.ValidationError("GOTCHA BOT!")

        return botcatcher

# Common necessary import for built-it validation from Django in views.py in app

from Django.core import validators

## Adding a check for empty fields

def clean\_botcatcher(self):

        botcatcher = self.cleaned\_data['botcatcher']

        if len(botcatcher)>0:

            raise forms.ValidationError("GOTCHA BOT!")

        return botcatcher

## Adding a check for a ‘bot’

class User\_input(forms.Form):

    name = forms.CharField(max\_length=245, required=False)

    email = forms.EmailField( required=False)

    text = forms.CharField(widget=forms.Textarea)

    botcatcher = forms.CharField(required=False,widget=forms.HiddenInput,

validators=[validators.MaxLengthValidator(0)])

## Custom validation example with check\_for\_z method

This is the own validation method done by user which check the name start with z or not.

def check\_for\_z(value):

    if value[0].lower() != 'z':

        raise forms.ValidationError("Name needs to start with z")

class User\_input(forms.Form):

    name = forms.CharField(validators=[check\_for\_z], required=False)

    email = forms.EmailField( required=False)

    text = forms.CharField(widget=forms.Textarea)

check\_for\_z is a customize method by user instead of using built in django validators.

## Adding a clean method for the entire form

**Same with above customize method but this one is not only for specific field. This will apply on the entire form with super().clean() which in this case is for checking email are match?.**

from django.core import validators

# def check\_for\_z(value):

#     if value[0].lower() != 'z':

#         raise forms.ValidationError("Name needs to start with z")

class User\_input(forms.Form):

    name = forms.CharField(required=False)

    email = forms.EmailField( required=False)

    verify\_email = forms.EmailField(label='Enter your email again!', required=True)

    text = forms.CharField(widget=forms.Textarea)

    # botcatcher = forms.CharField(required=False,widget=forms.HiddenInput)

    def clean(self):

        all\_clean\_data = super().clean()

        email = all\_clean\_data['email']

        vemail = all\_clean\_data['verify\_email']

        if email != vemail:

            raise forms.ValidationError("Make sure emails match!")

### What does super().clean() do?

* super().clean() **calls the parent class’s clean() method** from the Django Form class.
* This ensures that **default validation and cleaning** for each field defined in the form are performed.
* It collects the validated data for all fields in the form into a dictionary called **cleaned\_data**.

### ****Order of operations in Django form validation****

When a form is validated (e.g., by calling is\_valid()), Django follows this sequence:

1. Each field’s clean() method is called to validate and clean individual field values.
2. The cleaned values for those fields are added to cleaned\_data.
3. After field-level cleaning is complete, the form’s clean() method is called.
4. super().clean() ensures the cleaned data from previous steps is available.
5. Custom validation logic in clean() can access cleaned\_data.

# Processing Data that user input via form into model which is database

**First create model in the model.py for database. Then, import that model into form.py. After that create Form.model in form.py. inside that Form.model you can place validations above class meta and assigned the model that we create first into model like in green highlighted line below.**

class NewUserForm(forms.ModelForm):

    #You can add validators here

    class Meta:

        model = User

        fields = '\_\_all\_\_'

**As the next step import that form.model into views.py (from .forms import NewUserForm)**

**And than work like ususal in below code.**

def users(request):

    form = NewUserForm()

    if request.method == "POST":

        form = NewUserForm(request.POST)

        if form.is\_valid():

            form.save(commit=True)

            return index(request)

        else:

            print('ERROR FORM INVALID')

    return render(request,'appTwo/users.html',{'form':form})

**Relative URLs with templates**

**In url.py file**

**path('sign\_up/',views.users, name='sign\_Page'),**

**In html you can refer that name like that.**

**<a href=”{% url ‘sign\_Page’%}> Sign up Page</a>**

## Best method

app\_name = ‘basic\_app’ 🡨 app name

**<a href=”{% url ‘basicapp:sign\_Page’%} >Sign up Page</a>**

**So what happen is app\_name is the global name and it need to be assigned with actual app name. To use this you need to pair with name attribute from path.**

**Template inheritance (Like require\_one in php)**

**In both child and parents have Blocks**

**{% block body\_block %}**

**{% endblock %}**

To inheritance use {% extends "basicapp/base.html" %} it’s perform like require once in PHP

### Block in parent

* Blocks in the **parent template** serve as placeholders for content that child templates can override.
* If a block is **not overridden** in the child, the parent’s content is used.
* If a block **is overridden**, only the child’s content for that block is used (unless you include the parent’s content using {{ block.super }}).

### Block in child

Block in child can override the parent’s content which are inside block in parent.

# Built in filter in Django

## Adding multiple injection with dictionary in just one function inside views.py to HTML

def index(request) :

context\_dict = {‘text’:’Hello World’,’number’ :100}

return render(request, basicapp/index.html, context\_dict)

**You can just do that injections along with filter when you pass into the html template. I can apply all sort or calculation, small letter, capital letter there in HTML template.**

<!DOCTYPE html>

{% extends "basicapp/base.html" %}

    {% block body\_block %}

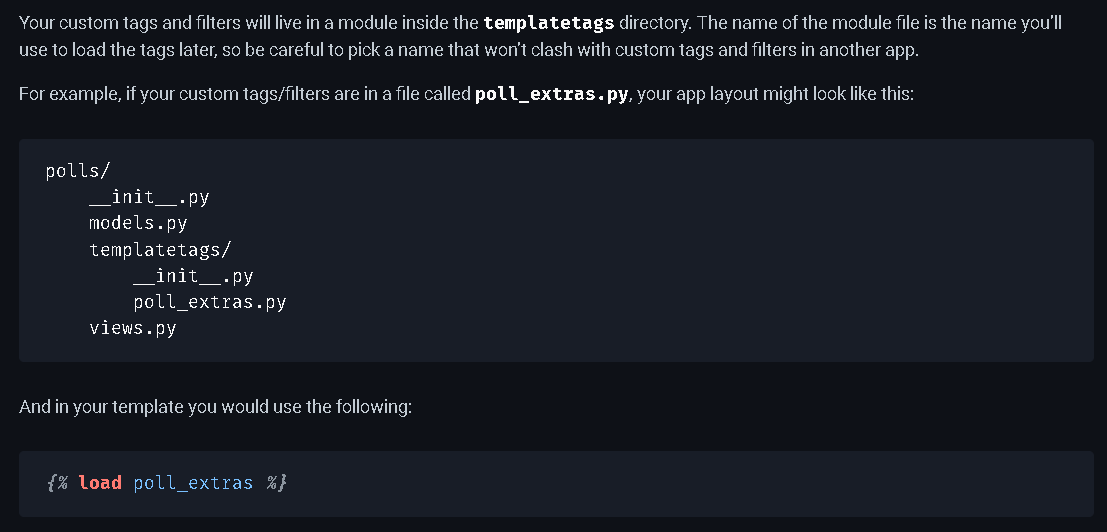
    <h1>{{text|upper}}<h1>

    <h2>{{number|add:”99”}}<h2>

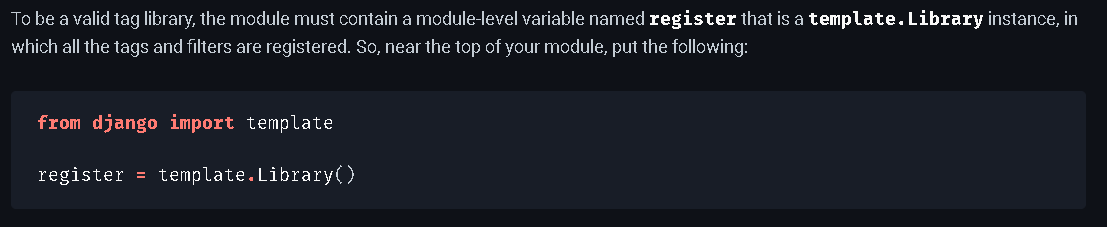
    {% endblock %}

# Custom template tags and filters

The app should contain a template tags directory, at the same level as models.py, views.py, etc.



1. To create custom filter
2. Create template tags directory, at the same level as models.py, views.py, etc.
3. Create a new file called \_\_init\_\_.py inside that directory.
4. Create another file again which is the place to write your custom temple inside template tags directory.
5. After that register your template



1. Create function for the filter. Filter can be with argument or without argument.