Python - Django Framework for Web Development Part 1

Prerequisites:

- Basic Fundamental of Programming in Python (especially Dictionary, Module, and OOP)
- HTML
- Basic knowledge in:
    - Any Templating Engine
    - Commands or Terminal
    - Database SQL Commands (DDL, DML, TCL, DQL, DCL)
    - Schema migration (Database Migration/ Database Version Control)

Requirements:

- Python 3
    - Download at: https://www.python.org/downloads/ and install
    - Python directory after installed:
        - Default: C:\Users\wilfred pine\AppData\Local\Programs\Python\Python38
        - Custom: C:\Python38

          Note: *All built-in modules, classes, methods in python can be found inside Python installation directory Lib folder. Built-in modules like math, time, turtle, decimal, enum, random, glob, os, etc. We can manually install other developed python external libraries using their provided command, it will automatically add inside Lib folder of Python. Django Framework is one of the examples of Python library use for web development.*
- Django Framework
    - Django directory after installed:
        - C:\Users\wilfred pine\AppData\Local\Programs\Python\Python38\Lib\site-packages\django
        - Or C: \Python38\Lib\site-packages\django

          Note: *All built-in modules, classes, methods in Django can be found inside Django installation directory. Like http, urls, forms, shortcuts, utils, template, etc.*

Installing Django

This is the recommended way to install Django.

Install pip. The easiest is to use the standalone pip installer. If your distribution already has pip installed, you might need to update it if it’s outdated. If it’s outdated, you’ll know because installation won’t work.

Enter the command to your terminal or Command Prompt:

```
py -m pip install Django
```

Checking Django Version

```
py -m django --version
```

Creating a project

You can create a project in Django and store it anywhere on your computer. Open a terminal and go to the directory or folder you want to create the project.

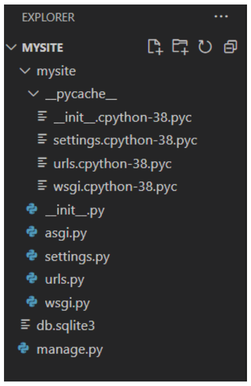
Example: PS C:\Users\mypc\Desktop\django_projects>

Then run the command to create a project "**mysite**":

```
django-admin startproject mysite
```

What is a Project in Django?

A **project** is a web application using Django. There is only ever **one** "project" and **many** "apps" within it. Django would automatically create your project folder and additional directory with your project name and then add the starter files within that directory. Example: our project named "mysite"



```
EXPLORER
MYSITE
  mysite
    __pycache__
      __init__.cpython-38.pyc
      settings.cpython-38.pyc
      urls.cpython-38.pyc
      wsgi.cpython-38.pyc
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
  db.sqlite3
  manage.py
```

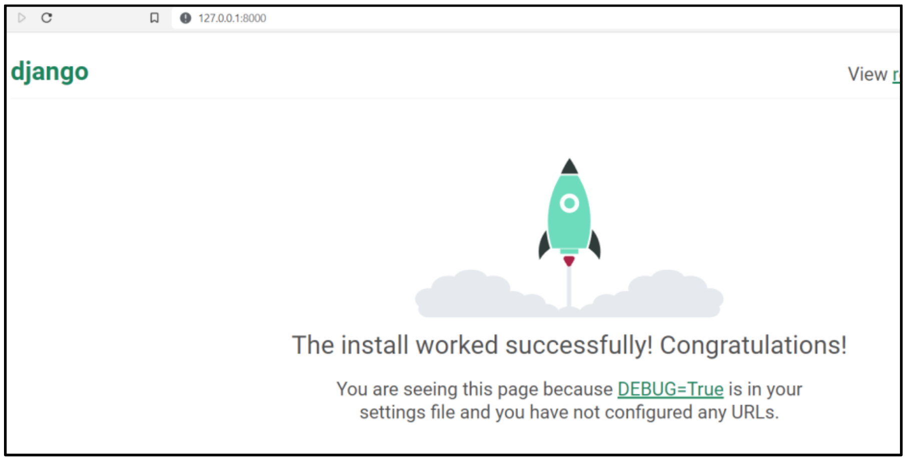
Run project

To run a Django project, open a terminal and point it to your project folder then run the commad:

```
py manage.py runserver
```

Example: PS C:\Users\mypc\Desktop\django_projects\mysite>py manage.py runserver

On your browser, browse **127.0.0.1:8000**. It will display the fresh installation landing page of a Django project.



The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

What is INSTALLED_APPS?

Within the newly created settings.py file inside your project folder (mysite\mysite\settings.py) is a configuration called INSTALLED_APPS which is a list of Django apps within a project. Django comes with six built-in apps that we can examine.

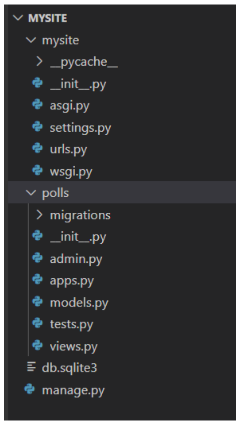
```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Creating "apps" in a Django project

We can add an app by using the startapp command so let's add the polls app now.

```
py manage.py startapp polls
```

That'll create a directory "**polls"**, which is laid out like this:



```
MYSITE
  mysite
    __pycache__
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
  polls
    migrations
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  db.sqlite3
  manage.py
```

What is an "**Apps"**?

A Django **app** is a *small library* representing a discrete part of a *larger project*. For example, our project "**mysite**" might have an app for "**polls**".

The **polls** app has been created and comes with its own associated files. Often you'll want to also add a urls.py file here, too.

We **also** must add the app to our INSTALLED_APPS settings of our project or else the Django project won't recognize it.

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'polls', # new
]
```

Note: we can create many "**apps"** in one project and activate it by adding it to our INSTALLED_APPS settings

3rd Party Packages

A 3rd party package is a plain old Django application that has been designed to be pluggable into any existing project with the Python packaging tools. You can see a tutorial on this here. It takes just a few additional steps.

This is a case where we can see the power of separating out functionality into smaller apps. It's far easier to share them either within a larger project, within a company, or to make public like a 3rd Party Package.

Naming Conventions

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent -- nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

https://www.python.org/dev/peps/pep-0008/#naming-conventions

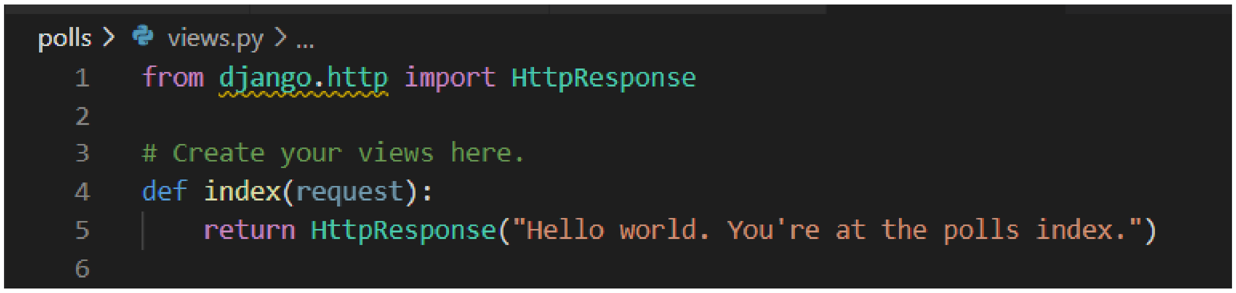
App naming conventions

An **app**'s name should follow Pep 8 Guidelines, namely it should be short, all-lowercase and not include numbers, dashes, periods, spaces, or special characters.

Views

Write your first View

Let's write the first view. Open the file **polls/views.py** and put the following Python code in it:



polls > views.py > ...

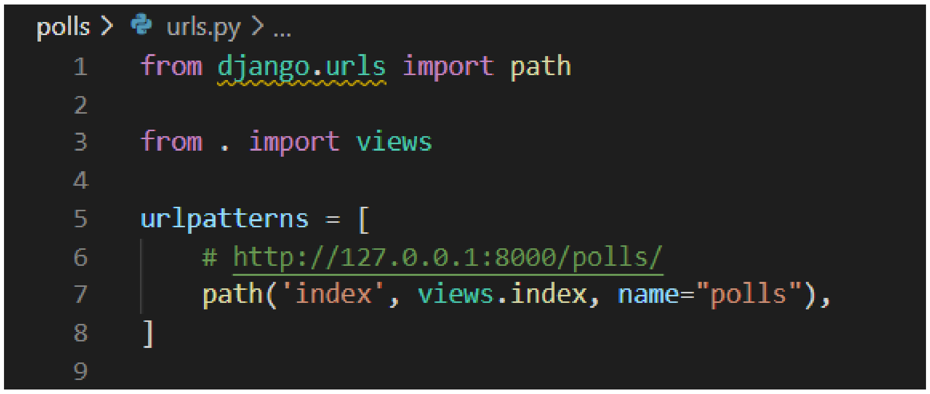
```python
from django.http import HttpResponse

# Create your views here.
def index(request):
    return HttpResponse("Hello world. You're at the polls index.")

```

This is the simplest view possible in Django. To call the view, we need to map it to a URL-and for this we need a URLconf.

To create URLconf in the **polls** directory, create a file called “**urls.py”**. Your app directory should now look like:

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    urls.py
    views.py
```

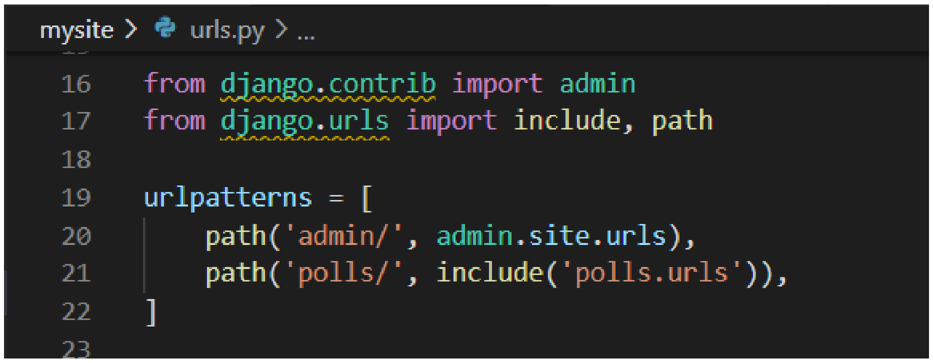
In the **polls/urls.py** file include the following code:



polls > urls.py > ...

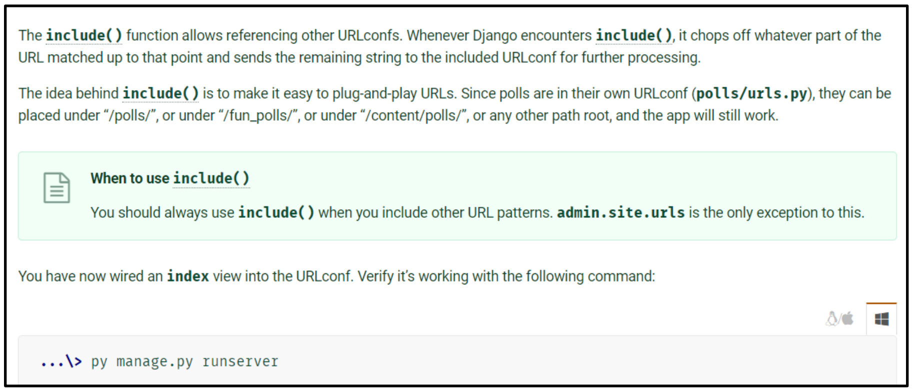
```python
from django.urls import path

from . import views

urlpatterns = [
    # http://127.0.0.1:8000/polls/
    path('index', views.index, name="polls"),
]
```

The next step is to point the root URLconf at the **polls.urls** module. In **mysite/urls.py**, add an import for **django.urls.include** and insert an **include()** in the **urlpatterns** list, so you have:



mysite > urls.py > ...

```python
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('polls/', include('polls.urls')),
]
```


The `include()` function allows referencing other URLconfs. Whenever Django encounters `include()`, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

The idea behind `include()` is to make it easy to plug-and-play URLs. Since polls are in their own URLconf (`polls/urls.py`), they can be placed under "/polls/", or under "/fun_polls/", or under "/content/polls/", or any other path root, and the app will still work.

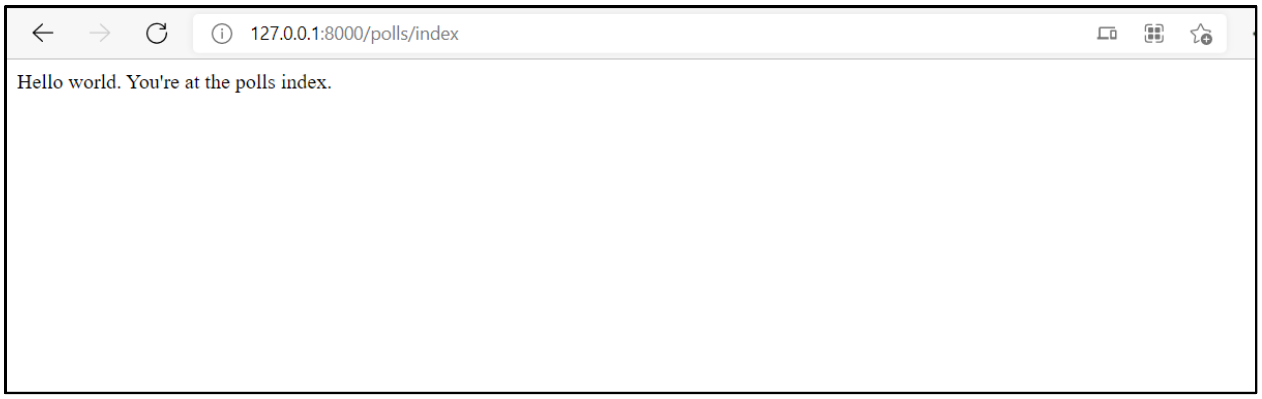
When to use `include()`

You should always use `include()` when you include other URL patterns. `admin.site.urls` is the only exception to this.

You have now wired an `index` view into the URLconf. Verify it's working with the following command:

```
...\> py manage.py runserver
```

Run the project and go to polls index page



```
Hello world. You're at the polls index.
```

What is View?

A **View** is a "type" of web page in your Django application that generally serves a specific function and has a specific template.

Example, in our **poll** application, we'll have the following four views:

- Question "index" page – displays the latest few questions.
- Question "index" page – displays the latest few questions.
- Question "detail" page – displays a question text, with no results but with a form to vote.
- Question "results" page – displays results for a particular question.
- Vote action – handles voting for a particular choice in a particular question.

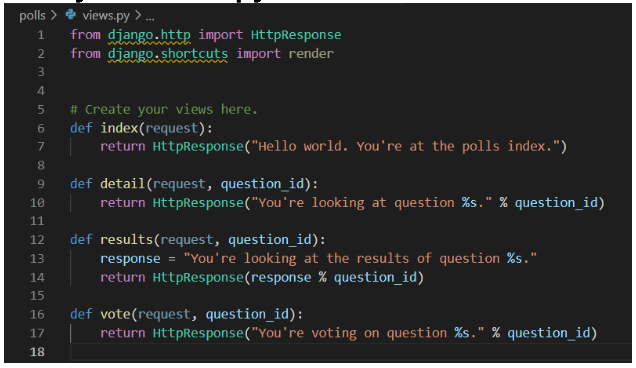
In Django, web pages and other content are delivered by views. Each view is represented by a Python function (or method, in the case of class-based views). Django will choose a view by examining the URL that's requested (to be precise, the part of the URL after the domain name).

Writing more views

Now let’s add a few more views to polls/views.py. These views are slightly different, because they take an argument:

```python
def detail(request, question_id):
    return HttpResponse("You're looking at question %s." % question_id)

def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)

def vote(request, question_id):
    return HttpResponse("You're voting on question %s." % question_id)
```

Now your views.py looks like this:



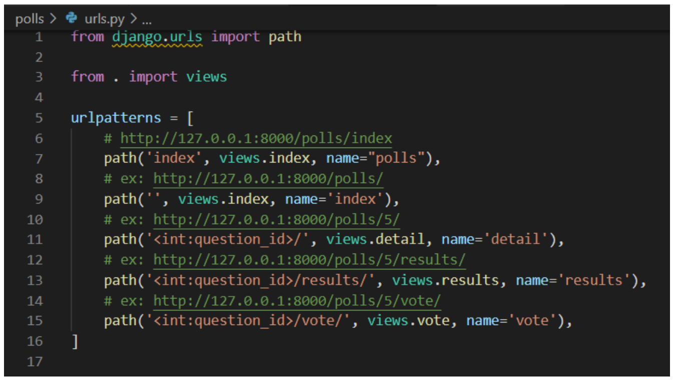
```
polls > views.py > ...
```

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


# Create your views here.
def index(request):
    return HttpResponse("Hello world. You're at the polls index.")

def detail(request, question_id):
    return HttpResponse("You're looking at question %s." % question_id)

def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)

def vote(request, question_id):
    return HttpResponse("You're voting on question %s." % question_id)

```

Wire these new views into the **polls.urls** module by adding the following path() calls:

```python
     # ex: http://127.0.0.1:8000/polls/
    path('', views.index, name='index'),
    # ex: http://127.0.0.1:8000/polls/5/
    path('<int:question_id>/', views.detail, name='detail'),
    # ex: http://127.0.0.1:8000/polls/5/results/
    path('<int:question_id>/results/', views.results, name='results'),
    # ex: http://127.0.0.1:8000/polls/5/vote/
    path('<int:question_id>/vote/', views.vote, name='vote'),
```

Now the polls’ urls.py looks like this:



```
polls > urls.py > ...
```

```python
from django.urls import path

from . import views

urlpatterns = [
    # http://127.0.0.1:8000/polls/index
    path('index', views.index, name="polls"),
    # ex: http://127.0.0.1:8000/polls/
    path('', views.index, name='index'),
    # ex: http://127.0.0.1:8000/polls/5/
    path('<int:question_id>/', views.detail, name='detail'),
    # ex: http://127.0.0.1:8000/polls/5/results/
    path('<int:question_id>/results/', views.results, name='results'),
    # ex: http://127.0.0.1:8000/polls/5/vote/
    path('<int:question_id>/vote/', views.vote, name='vote'),
]

```

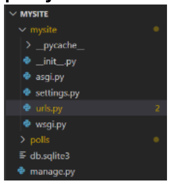
Urls (routing)

path()

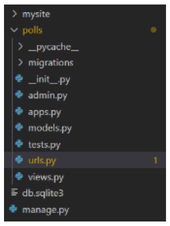
`path`(*route*, *view*, *kwargs=None*, *name=None*)

- Returns an element for inclusion in **`urlpatterns`**.

- **Project's Urls** (mysite/urls.py) – The default urls or the config urls or the main urls of our project. This will be the one who rendered and can be call at our web browser.



- **Apps' Urls** (polls/urls.py) – The urls of our specific apps. It cannot be call or activated unless we include it in our project's config/main urls (mysite/urls.py).



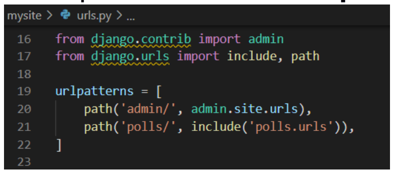
- **Importing apps' urls into project's main urls**

  Inside **mysite/urls.py** add the code:

```
path('polls/', include('polls.urls')),
```

  The first argument of path() method is the path you called in browser.

  Example: 127.0.0.1:8000/**polls/**



```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('polls/', include('polls.urls')),
]
```

  When you call the first argument '**polls/**' , the system will execute the second argument which is the **urls.py** of "**polls"** app.
  Inside the **'polls/urls.py',** you will see another path**.** When this file executed, the system finds the next path after **"polls/"** in "127.0.0.1:8000/**polls/"**

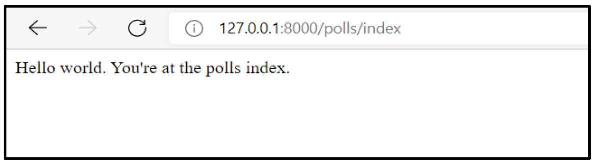
```
from django.urls import path
from . import views
urlpatterns = [
    # http://127.0.0.1:8000/polls/index
    path('index/', views.index, name="polls"),
]
```

Example: 127.0.0.1:8000/polls/**index**/
In short, when you call the **"index/"** path, it will execute the **index**() method inside the **views.py** file (views.index).

polls/views.py

```
from django.http import HttpResponse
from django.shortcuts import render
# Create your views here.
def index(request):
    return HttpResponse("Hello world. You're at the polls index.")
```

Output:



Hello world. You're at the polls index.

Templates

Being a web framework, Django needs a convenient way to generate HTML dynamically. The most common approach relies on templates. A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

A Django project can be configured with one or several template engines (or even zero if you don't use templates). Django ships built-in backends for its own template system, creatively called the Django template language (DTL), and for the popular alternative Jinja2. Backends for other template languages may be available from third-parties.

Template Engine

- *Jinja (Jinja2)*

  Jinja, also known and referred to as "Jinja2", is a popular Python template engine written as a self-contained open-source project. Some template engines, such as Django templates are provided as part of a larger web framework, which can make them difficult to reuse in projects outside their coupled library.

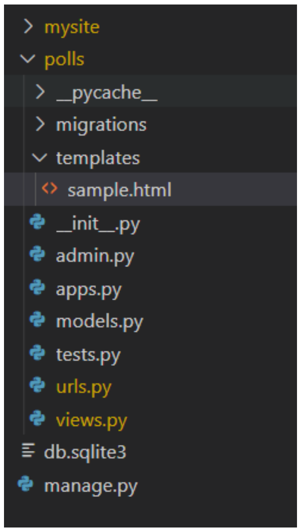
- *Django templating*

  Django comes with its own template engine in addition to supporting (as of Django 1.9) drop-in replacement with other template engines such as Jinja.

Creating Templates

1. Create a “template” folder inside your “**polls”** apps
2. Create a sample html file inside “**template**” folder

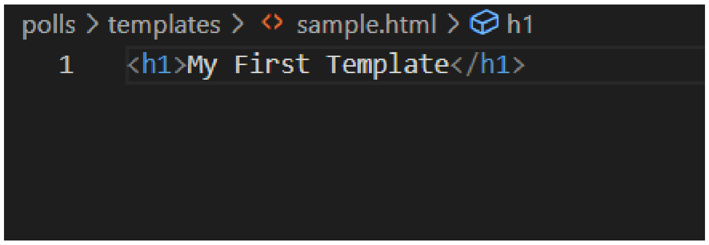
Your **polls** directory looks like this:



```
> mysite
v polls
  > __pycache__
  > migrations
  v templates
    sample.html
  __init__.py
  admin.py
  apps.py
  models.py
  tests.py
  urls.py
  views.py
db.sqlite3
manage.py
```

Rendering HTML Template

Open the html template and write the following code:



polls > templates > sample.html > h1

```html
<h1>My First Template</h1>
```

To render this html template, create this method in your **polls/views.py**

```python
def sample(request):
    return render(request, 'sample.html')
```

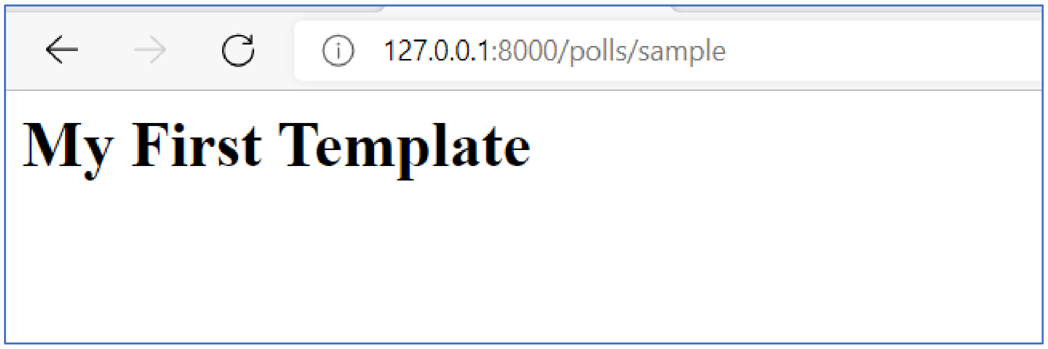
In order to use the **render()** method, you need to import the render method from **django.shortcuts** Module at the top of **polls/views.py**

```python
from django.shortcuts import render
```

Then add path method in your **polls/urls.py** to call the views in our browser

```python
    #http://127.0.0.1:8000/polls/sample
    path('sample', views.sample, name='sample'),
```

Now, go to your browser and call the 127.0.0.1:8000/polls/sample url


127.0.0.1:8000/polls/sample

My First Template

Passing data from views to template

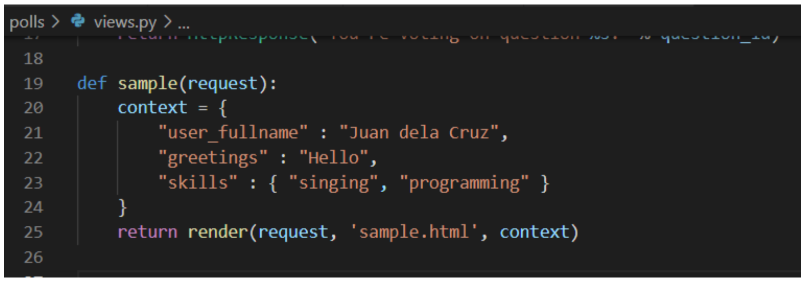
Now, we are going to edit the *sample()* method in our **polls/views.py** and insert inside the method this dictionary before the return statement:

```
context = {
        "user_fullname" : "Juan dela Cruz",
        "greetings" : "Hello",
        "skills" : { "singing", "programming" }
    }
```

Now add the "**context"** variable as the 3rd argument in **render()** method

```
    return render(request, 'sample.html', context)
```

The **sample()** method in our **polls/views.py** is now looks like this:


polls > views.py > ...

```

def sample(request):
    context = {
        "user_fullname" : "Juan dela Cruz",
        "greetings" : "Hello",
        "skills" : { "singing", "programming" }
    }
    return render(request, 'sample.html', context)

```

Displaying the data from views into template using the "Template Engine"

The syntax of the Django template language involves four constructs.

1. Variables
2. Tags
3. Filters
4. Comments

Variables

A variable output a value from the context, which is a dict-like object mapping keys to values.

Variables are surrounded by {{ and }} like this:

```
<h5>{{ greetings }} </h5>
<p>{{ user_fullname }}</p>
```

Tags

Tags provide arbitrary logic in the rendering process.

This definition is deliberately vague. For example, a tag can output content, serve as a control structure e.g. an “if” statement or a “for” loop, grab content from a database, or even enable access to other template tags.

Tags are surrounded by {% and %} like this:

```
{% csrf_token %}
```

Most tags accept arguments:

```
{% cycle 'odd' 'even' %}
```

Some tags require beginning and ending tags:

```
<ul>
    {% for skill in skills %}
      <li>{{ skill }}</li>
    {% endfor %}
</ul>
```

Filters

Filters transform the values of variables and tag arguments.

They look like this:

```
<p>{{ user_fullname|title }}</p>
```

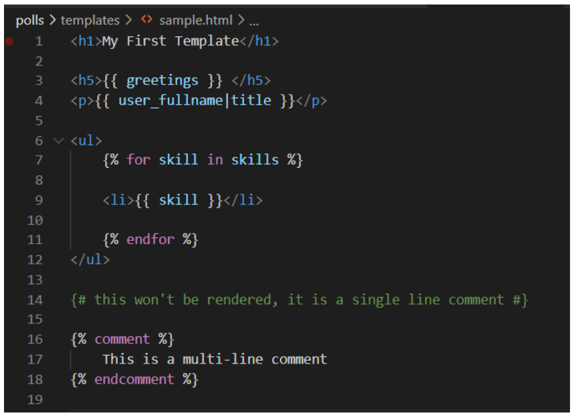
If the **user_fullname** is **“juan dela cruz”,** now it is converted to **“Juan Dela Cruz”**

Comments

Comments look like this:

```
{# this won't be rendered, it is a single line comment #}

{% comment %}
    This won’t be rendered, this is a multi-line comment
{% endcomment %}
```

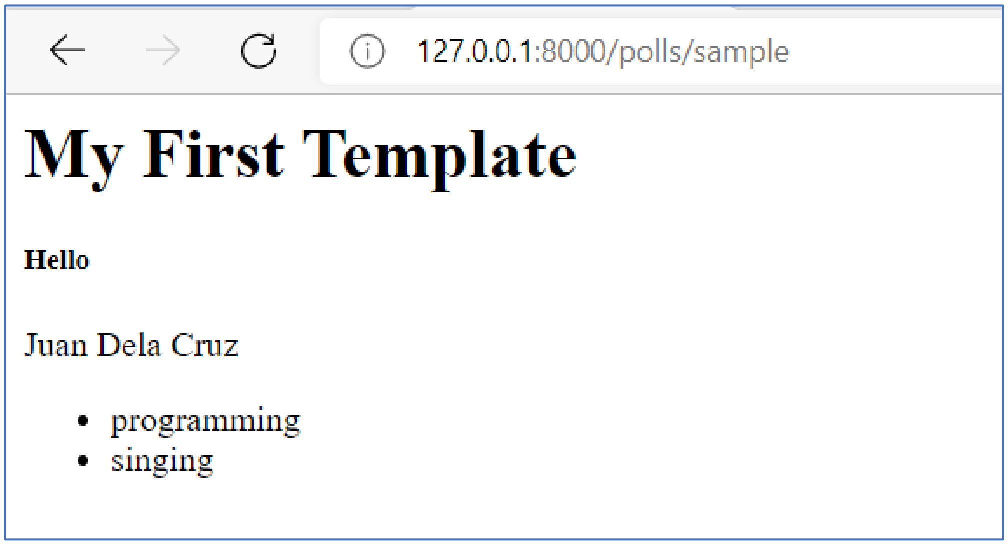
File: polls/templates/sample.html



polls > templates > sample.html > ...

```
<h1>My First Template</h1>

<h5>{{ greetings }} </h5>
<p>{{ user_fullname|title }}</p>

<ul>
    {% for skill in skills %}

    <li>{{ skill }}</li>

    {% endfor %}
</ul>

{# this won't be rendered, it is a single line comment #}

{% comment %}
    This is a multi-line comment
{% endcomment %}

```

Output:



127.0.0.1:8000/polls/sample

My First Template

Hello

Juan Dela Cruz

- programming
- singing

REFERENCES:

https://docs.djangoproject.com/en/4.0/

https://learndjango.com/tutorials/django-best-practices-projects-vs-apps#:~:text=Apps%20A%20Django%20app%20is%20a%20small%20library,another%20app%20called%20payments%20to%20charge%20logged-in%20subscribers.