
# Django Class Notes

Clarusway

# Functional Views & Templates, Static Files

Nice to have VSCode Extentions:

- Djaneiro - Django Snippets

Needs

- Python, add the path environment variable
- pip
- virtualenv

## Summary

- Intro to Views&Templates with additional note
- Create project and app
- Views
  - Request object
  - Interview Question
  - Create a template view
- Templates
  - Variables
  - Tags
  - Filters
  - Comments
  - Advenced examples

    - Interview Question
- Static Files

## Create project and app:

- Create a working directory, name it as you wish, cd to new directory
- Create virtual environment as a best practice:

```
python3 -m venv env # for Windows or
python -m venv env # for Windows
virtualenv env # for Mac/Linux or;
virtualenv env -p python3 # for Mac/Linux
```

- Activate scripts:

```
.\env\Scripts\activate  # for Windows
source env/bin/activate  # for MAC/Linux
```

- See the (env) sign before your command prompt.
- Install django:

```
pip install django
```

- See installed packages:

```
pip freeze

# you will see:
asgiref==3.3.4
Django==3.2.4
pytz==2021.1
sqlparse==0.4.1

# If you see lots of things here, that means there is a problem with your virtual
env activation.
# Activate scripts again
```

- Create requirements.txt same level with working directory, send your installed packages to this file, requirements file must be up to date:

```
pip freeze > requirements.txt
```

- Create project:

```
django-admin startproject project
django-admin startproject project .
# With . it creates a single project folder.
# Avoiding nested folders
# Alternative naming:
django-admin startproject main .
```

- Various files has been created!
- Check your project if it's installed correctly:

```
python3 manage.py runserver  # or,
python manage.py runserver  # or,
py -m manage.py runserver
```

- (Optional) If you have nested project folders with the same name; change the name of the project main (parent) directory as src to distinguish from subfolder with the same name!

```
# optional
mv .\project\ src
```

- Lets create first application:
- Go to the same level with manage.py file:

```
cd .\src\
```

- Start app

```
python manage.py startapp app

# Alternative naming:
python manage.py startapp home
```

# Views

- Go to views.py in app directory
- Check your interpreter on VSCode, and choose the one with virtual env
- Create first view by adding:

```
from django.http import HttpResponse

def home_view(request):
    html = "<html><body>Hello World!</body></html>"
    return HttpResponse(html)
```

- Must include URL path of new app to the project url list.
- Go to urls.py and add:

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

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

- Create urls.py under app, and add:

```
from django.urls import path
from .views import home_view

urlpatterns = [
    path('', home, name="home"),
]
```

- Go to settings.py and add under INSTALLED_APPS:

```
'app.apps.FirstappConfig'  # or
'app'
```

- Run our project:

```
python manage.py runserver
```

- Go to http://localhost:8000 in your browser, and you should see the text “Hello, world.”, which you defined in the index view.

## Some experimentation about request object:

- Go to the views.py, try adding these lines and interact with the web page:

```
def home(request):
    # print(request)
    # <WSGIRequest: GET '/'>
    # gives info about the request object

    # print(request.GET)
    # <QueryDict: {}> an empty dict
    # but after some querry on browser like /?q=abc
    # <QueryDict: {'a': ['3']}>
    # returns values brought by GET method

    # print(request.GET.get("q"))
    # returns the value of q

    # print(request.COOKIES)
    # COOKIES: HTTP cookies are small blocks of data created by a web server while
a user is browsing a website and placed on the user's computer or other device by
the user’s web browser.
    #
    # When you visit a website that uses cookies, a cookie file is saved to your
PC, Mac, phone or tablet. It stores the website's name, and also a unique ID that
represents you as a user. That way, if you go back to that website again, the
website knows you've already been there before.
    #
    # {'csrftoken':
'8RR6TAZe8rBtyQl1H1tbb3umLijiQBAT5QhtTsDOPseM4letTnixfnYryrPcb1ZS'}
    #
    # A CSRF token is a unique, secret, unpredictable value that is generated by
the server-side application and transmitted to the client in such a way that it is
included in a subsequent HTTP request made by the client.
    # Session id if you login the page

    # print(request.user)
    # AnonymousUser

    # print(request.path)
    # /

    # print(request.method)
    # GET

    # print(request.META)

    # if request.method == "GET":
    #     print("This is a GET method")

    return HttpResponse("Hello, Jane")
```

Django uses request and response objects to pass state through the system.

When a page is requested, Django creates an HttpRequest object that contains metadata about the request. Then Django loads the appropriate view, passing the HttpRequest as the first argument to the view function.

Each view is responsible for returning an HttpResponse object.

### Interview Question:

**What are views in Django?**

A view function, or “view” for short, is simply a Python function that takes a web request and returns a web response. This response can be HTML contents of a web page, or a redirect, or a 404 error, or an XML document, or an image, etc.

Example:

```
from django.http import HttpResponse
def sample_function(request):
 return HttpResponse(“Welcome to Django”)
```

There are two types of views:

- Function-Based Views: We import our view as a function.
- Class-based Views: It’s an object-oriented approach.

## Create a template view

- Create app/templates/app directory and create a home.html file under it:

```
Hello World!
```

- Change your views to comply with the template:

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

def home(request):
    return render(request, "app/home.html")
```

- Add a context:

```
def home(request):

    context = {
        'first_name': 'Rafe',
        'last_name': 'Stefano',
    }

    return render(request, "app/home.html", context)
```

- Keep going wiht the examples below.

# Templates

### Variables: {{ variable }}

https://docs.djangoproject.com/en/3.2/topics/templates/#variables

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

- With a context of {'first_name': 'John', 'last_name': 'Doe'}, this template renders to: My first name is John. My last name is Doe.
- Add a context:

```
def home(request):
    context = {
        'title': 'clarusway',
        'dict1': {'django': 'best framework'},
        'my_list': [2, 3, 4]
    }
    return render(request, "app/home.html", context)
```

### Tags: {% tag %}

- Some tags require beginning and ending tags:

```
{% if %}
{% endif %}


{% if my_list %}
    print("List is not empty")
{% endif %}


{% for num in my_list %}
    <li>{{ num }}</li>
{% endfor %}
```

### Filters

```
- Filters transform the values of variables and tag arguments: {{ variable|filter
}}

```html
{{ dict1.django|title }}
```

## Comments:

- Single line: {# this won't be rendered #}

{# This is a single line comment #}

- Multi line: {% comment %}

{% comment %} This is a multi line comment, you can't see it on the page! {% endcomment %}

Examples:

```
<h1>Hello, this is home page</h1>
    {{ title }} <br> <br>
    {{ dict_1 }} <br> <br>
    {% for i in my_list %}
    {{ i }} <br>
    {% endfor %}
    {{ my_list }}
```

- A different div example:

```
<a href="{% url 'pet_detail' pet.id %}">
```

## Advenced examples:

```
{% extends "base.html" %}

{% load static %}

{% block blockname %}

{% endblock blockname %}

<script src="{% static 'main.js' %}"></script>
```

```
{% extends "base.html" %}
{% block content %}
```
```

```
<div>
    {% for pet in pets %}

        <div>
            <a href="{% url 'pet_detail' pet.id %}">
            <h3>{{ pet.name | capfirst }}</h3>
            </a>
            <p>{{ pet.species }}</p>
            {% if pet.breed %}
            <p>Breed: {{ pet.breed }}</p>
            {% endif %}
            <p class="hidden">{{ pet.description }}</p>
        </div>

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

### Interview Question

**What are templates in Django or Django template language?**

Templates are an integral part of the Django MVT architecture. They generally comprise HTML, CSS, and js in which dynamic variables and information are embedded with the help of views. Some constructs are recognized and interpreted by the template engine. The main ones are variables and tags.

A template is rendered with a context. Rendering just replaces variables with their values, present in the context, and processes tags. Everything else remains as it is.

The syntax of the Django template language includes the following four constructs :

- Variables
- Tags
- Filters
- Comments

---

# Static Files (images, JavaScript, CSS)

Static Files reference

Websites generally need to serve additional files such as images, JavaScript, or CSS. In Django, we refer to these files as “static files”. Clients download static files as they are from the server.

Django provides `django.contrib.staticfiles` to help you manage them, this collects static files from each of your applications (and any other places you specify) into a single location that can easily be served in production.

- Settings on settings.py about static files:

```
# Default place which Django will look for static files in everywhere:
STATIC_URL = 'static/'

# Base static files, or files from different directories:
STATICFILES_DIRS = [
    BASE_DIR / "static",
]
# or;
STATICFILES_DIRS = (os.path.join(BASE_DIR, 'static'))


# The absolute path to the directory where collectstatic will collect static files
for deployment.
# Example: "/var/www/example.com/static/"
STATIC_ROOT = "staticfiles-cdn"
#  or;
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

- It is possible to search for related static file path:

```
python manage.py findstatic css/base.css --first --verbosity 2
# return the first match for each relative path, use the --first option
# by setting the --verbosity flag to 2, you can get all the directories which were
searched
```

- A basic example for a template using static files is:

```
{% load static %}

<head>
    <link rel="stylesheet" href="{% static 'app/style.css' %}">
</head>

<h1>Hello World!</h1>
<img src="{% static 'app/example.jpg' %}" width="300" alt="Example">

<video width="400" controls>
    <source src="{% static 'app\kittens.mp4' %}" type="video/mp4">
    Your browser does not support HTML video.
  </video>
```

---

## Additional note about the differences between static files variables

- **Development**

STATIC_ROOT is useless during development, it's only required for deployment.

While in development, STATIC_ROOT does nothing. You don't even need to set it. Django looks for static files inside each app's directory(myProject/appName/static) and serves them automatically.

This is the magic done by manage.py runserver when DEBUG=True.

- **Deployment**

When your project goes live, things differ. Most likely you will serve dynamic content using Django and static files will be served by Nginx. Because Nginx is incredibly efficient and will reduce the workload off Django.

This is where STATIC_ROOT becomes handy, as Nginx doesn't know anything about your django project and doesn't know where to find static files.

So you set STATIC_ROOT = '/some/folder/' and tell Nginx to look for static files in /some/folder/. Then you run manage.py collectstatic and Django will copy static files from all the apps you have to /some/folder/.

- **Extra directories for static files**

STATICFILES_DIRS is used to include additional directories for collectstatic to look for. (Not necessarily app directories.) For example, by default, Django doesn't recognize /myProject/static/. So you can include it yourself.

- MEDIA_ROOT is the folder where files uploaded using FileField will go.
- STATICFILES_STORAGE

Default: 'django.contrib.staticfiles.storage.StaticFilesStorage'

The file storage engine to use when collecting static files with the collectstatic management command.

- STATICFILES_FINDERS

Default:

[ 'django.contrib.staticfiles.finders.FileSystemFinder', 'django.contrib.staticfiles.finders.AppDirectoriesFinder', ]

The list of finder backends that know how to find static files in various locations.

---

## (Optional - Advenced) How to deploy static files

Guide for "Serving static files in production"

☺ **Happy Coding!** ✍

Clarusway