Juggling Django - a beginner's guide to getting started with back-end in Python

Welcome! Whether you have decided to be 'productive in pandemic' by learning trending technologies or learnt Python and want to explore the web development domain or explored front-end development and confused between all the back-end options, you have come to the right place.

This document aims to give you a head start in Django - a web framework in Python.

Fun fact - The name Django derived from the famous guitarist Django Reinhardt. He had only three fingers but was a master at it.

Before starting, let's understand what exactly is a web framework. It is a code library that makes web development quicker and easier by giving basic patterns for building reliable, scalable and maintainable web applications. Think of it as a way to create shortcuts that can prevent otherwise overwhelming and repetitive code. For example, you may have written some code from scratch to save data in a database for your web application. Instead of rewriting that code from scratch each time you make a website or web service, you can use existing features in a web framework. A few of the most popular and successful web frameworks include Django, Express, Ruby on Rails.

Now, why should you learn Django?

- **Based on Python** - Python is one of the easiest and most popular programming languages out there supporting coming of age technologies like ML and AI.

- **Developer-friendly Documentation** - Django has been best at documentation from the beginning, from the point it became open source in 2005 till date. It is like a very well-established library for developers constantly updated by the active community.

- **Rapid development** - You will not need expert-level knowledge to make a fully functional website. You will also not need to create separate server files to design and connect the database or create an admin panel for handling the back-end. Django takes care of many such tasks.

- **SEO and security** - Using Search Engine Optimization(SEO), you can add your website to the search engine such that it appears in the top results. Django also covers the security loopholes by default which are generally left open for the back-end developers to complete.

- **ORM -** Django ORM provides an elegant and powerful way to interact with the database. ORM stands for Object Relational Mapper. It is just a fancy word describing how to access the data stored in the database in an Object-Oriented fashion.

Fun fact - Disqus, YouTube, Instagram, Spotify, Dropbox, NASA, Pinterest, Mozilla, Quora, Reddit, Bitbucket, Udemy, National Geographic are some of the countless companies using Django in their back-end.

Now that you are all set to learn Django, here are some prerequisites:

- Knowledge of Python. (basic syntax, data types, writing functions, built-in libraries)
- Basic understanding of Object-Oriented Programming.
- Download pip.
- Set up a virtual environment.
- Download required dependencies.

A **virtual environment** is a tool that helps to keep dependencies required by different projects separate by creating isolated **environments** for them. Use a new virtual environment whenever you work on any Python-based project to ensure that updating the version of any dependency on your system doesn't affect existing projects.
Let us now start by setting up a virtual environment for our project. The commands are different for Mac/Linux users. So check out the video link below. Here are the steps for Windows users. First, open the command prompt(type in the search box on the home screen) and use the cd command to move to a specific location like Desktop or any other folder on your computer. (cd stands for change directory)

$ cd <path_of_the_preferred_location>
You will find the path of a specific location in file explorer.

We will now use the venv module that's already present in the Python standard library. (no need to install separately)

To create a new virtual environment, run the following in the command prompt.

$ python -m venv <name_of_virtual_env>

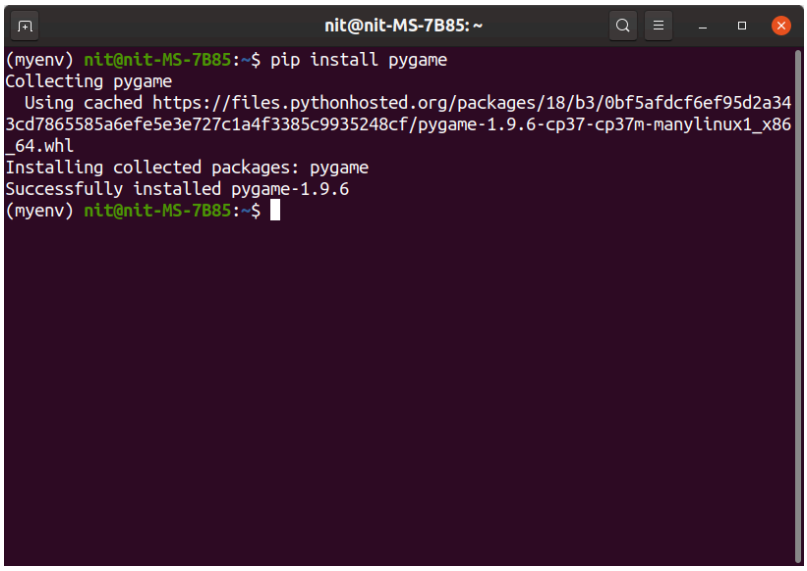
Run the command dir to confirm the creation of our new virtual environment.

$dir

Now, to activate the new virtual environment, run the following:

$ <name_of_virtual_environment>\scripts\activate.bat

You will now see the new environment's name in the parenthesis in your command line like this.



At this point, if you run the pip list command to view all existing dependencies, you will see only 2 - pip and setuptools. We will download the required dependencies using the pip install command wherever required in the future.

$ pip list

$ pip install <name_of_dependency>

To exit from the virtual environment, use the command deactivate.

$ deactivate

Once you deactivate a virtual environment, all the dependencies installed get deleted. Hence, we need to store all of these dependencies in a file called requirements.txt to install them again.

$ pip freeze > requirements.txt

The pip freeze command displays a list of all dependencies. The above command is a short-hand notation that stores the output of the pip freeze command in a new file requirements.txt created in the same directory displayed in the command line.

To install all the dependencies after activating the virtual environment again, use the following command:

$ pip install -r requirements.txt

To delete a virtual environment, use the following command:

$ rmdir <name_of_virtual_env> /s

Here, rmdir stands for remove directory. /s is to ensure that every subdirectory, empty or not, is deleted.

And that's it. We have learnt to set up our virtual environment. To distinguish different virtual environments from one another in the long run, use a unique name or store the environment directory in the project.
If you had any issues following along, here are some video tutorials for the same.

Windows:
http://youtube.com/watch?v=APOPm01BVrk

Mac/Linux:
https://www.youtube.com/watch?v=Kg1Yvry_Ydk

Fun fact - It cost $7 million to develop Django. After 16 years of release, the Django development community consists of 1200 members today!

Phew! The virtual environment set-up is finally done. Now we install all the required dependencies.

Activate the virtual environment you created. We will now install Django.

$ pip install django

To confirm the installation, run the following command that displays the installed version.

$ python -m django --version

To display the list of all Django-specific commands, run the following command:

$ django-admin

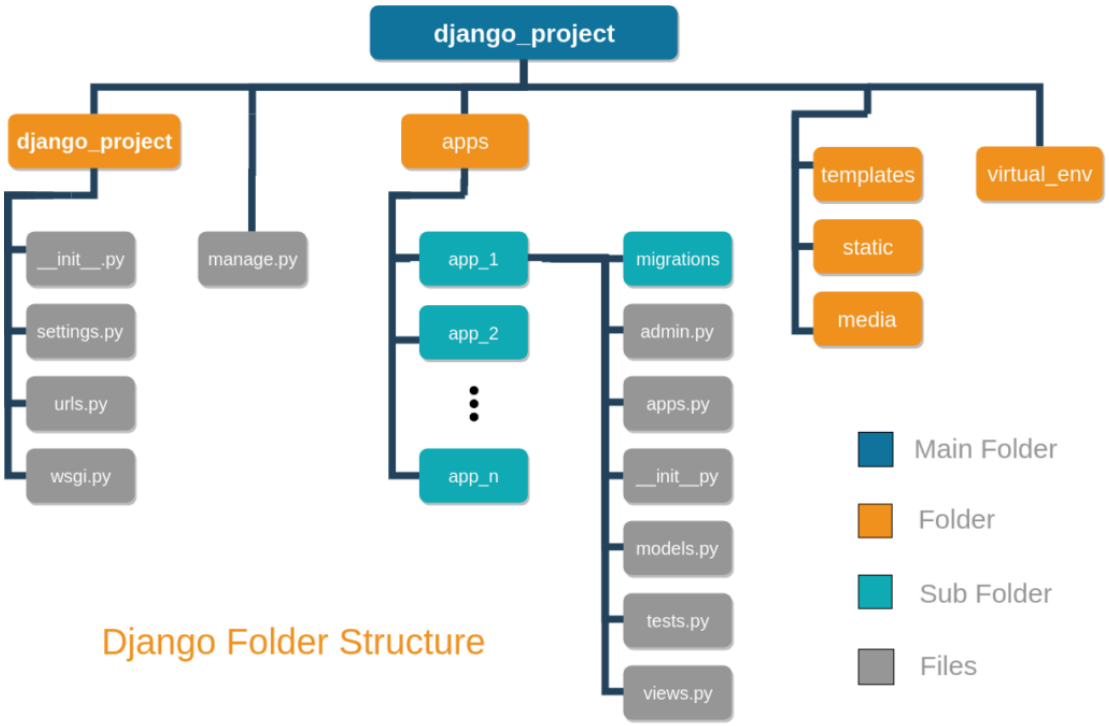
We will use one of these commands to create a brand new Django project! The project name must contain only lower-case letters and underscores to improve readability.

$ django-admin startproject <name_of_new_project>

You will find a new project folder in the current directory (the location you mentioned in the cd command) with the name you specified above.

Open this folder using a code editor of your choice. I recommend Sublime Text or VSCode.

Once you open the project folder, you will notice many files Django already created for us. Do not worry. Let us see them one by one.



If the name of the project is django_project. Then,

- The outer django_project/ is the root directory. Django root directory is the default app that Django provides you when you use the startproject command. It is a container for your project. Its name does not matter to Django; you can rename it to anything you like.
- manage.py is a command-line utility that lets you interact with this Django project like when you have to run, deploy, debug and test the project.
- The inner django_project/ directory is the actual Python package for your project. Its name is the Python package name you will need to use to import anything in the project. It has the configuration files of project settings like __init__.py, settings.py, wsgi.py and urls.py.
- The __init__.py file is empty and it exists in the project only for one purpose, which is conveying to the Python interpreter that the inner django_project folder is a package(a collection of modules or code blocks for frequently used operations).
- The settings.py file is the main file where we will be adding all our project applications, middleware applications and configure default settings.
- The urls.py file contains the project level URL information. The purpose of this file is to connect the web apps(distinct features) within the project.

- And finally, the wsgi.py file. This file is necessary if you want to deploy the project on a specific server. Else, we don't use it much during the development of the project.

Even if all of this seems confusing to you at this stage, don't worry. You will have much better clarity once we start writing code.

Now back to the command prompt. If you run the command,

$ python manage.py runserver

It will display the IP address of our project - http://127.0.0.1:8000/

You can replace 127.0.0.1 with localhost.

Type the address in your browser search box and it will display the default Django website. Now let us start writing code to display specific information of our choice.

Now, think about why we created this project. What are its features? Let's consider YouTube. It has a homepage with video content, several buttons leading to the trending videos section, our profile, settings etc. In Django terminology, we call each of them - distinct pieces of code performing a specific task - an app. Depending on your requirement, follow the procedure below and create an app for your project.

$ python manage.py startapp <name_of_your_app>

Once you run that, you will see another folder in your outer project folder called the <name_of_your_app>

You will now see a couple of other files called __init__.py, admin.py, apps.py, models.py, tests.py, views.py and a migrations folder.
Yes! I understand that it is overwhelming to see so many files even before writing a single line of code. But please be patient. These files make the code base very organized.

Let us start by displaying some text on our home screen instead of the default Django website.

First, open the views.py file. You will notice an import in the file. The pre-existing import statements are functions most commonly used by developers. views.py file holds code that is responsible for displaying HTML or generating HTTP Response on the screen.
Include this code snippet in the file.

```
from django.http import HttpResponse
def home(request):
    return HttpResponse("Home Page")
```

Here, the home function takes a compulsory argument request. The HttpResponse function takes HTML code/plain text as an argument, which you will see on the home screen.

Open <name_of_your_app>/urls.py and include the snippet below

```
from . import views
urlpatterns = [
    path("", views.home, name='home'),
]
```

To open the home page, we must let the interpreter know its specific location within the app. urls.py file in the app folder holds all the paths to different pages.

Open the urls.py in the root directory and include the below snippet,

```
urlpatterns = [
```

```
    path('home/', include('<name_of_your_app>.urls')),
    path('admin/', admin.site.urls),
]
```

To direct the interpreter to the app, we specify the urls.py of the app in this file. The home/ indicates that http://127.0.0.1:8000/home is the URL for the app home page.

Now open the command prompt once again and execute the runserver command. If you now open the web server http://127.0.0.1:8000/home, you will see the text Home Page displayed on the screen. It may not seem like much, but give yourself a pat on your back for taking small strides into the spectacular world of backend development using Django.

To summarize, you understood the features of Django, set up your own virtual environment, created your first Django project and displayed some text on your app home screen! That's all I had in store for you. Stay tuned for more.

Recap Time:

In our previous post, we tried to explain the basics of Django and its significance in the web development field. We have seen how to create a virtual environment and install all the required modules to create a Django project and we have learnt the importance of all the files that are automatically created once we begin a project. After all the setup, we created an app for our project and used a HttpResponse which will display the text “Home Page” when we go to our website.

After finishing all the setup processes, if you’re still doubtful about how much of it you remember, you have absolutely no need to worry! Everything will start making sense once we start building up the website. All the fun stuff starts now!

First, lets see what exactly is happening when we open our browser and search in the address bar for:

127.0.0.1:8000/home

Let's say the name of the app we have created is "home".
When we open the urls.py file in the root directory of our project, it says :

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

This file is telling that whenever the "127.0.0.1:8000/home" is called, it needs to be handled by the 'home.urls' file, which is basically the urls.py file located inside the home folder in the root directory. This file says :

```
from . import views
urlpatterns = [
    path('', views.home, name='home'),
]
```

This file tells the compiler to call a function in the views.py file whose name is 'home'. The 'home' function inside the views.py has the following code:

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

# Create your views here.
def home(request):
    return HttpResponse("Hello World!")
```

And Voila! This is how the text "Hello World!" is generated whenever we open "127.0.0.1:8000/home" in our browser.

But this Home page doesn't really do anything. We'd want our home page to have multiple texts, print an image, maybe even have an amazing UI style too. All these stylings can be achieved by using a HTML file. But we know that HTML files are static, and once we specify the content, we cannot really change it. We don't want this for our website. We'd like our website to be dynamic. This is where Django brings in the amazing concept of "Templates"! Using templates, Django helps us to have dynamic content in HTML web pages, keeping the layout of the web page constant. This is possible by DTL or Django Template Language.

Let's get started with **Templates**. There are two steps that we'll have to do now :

First, create a new folder in the root directory of our project, with the name "templates". This folder will contain all the HTML files that we will be using for our website.

Secondly, we'll need to let the settings.py file of our project know where to look for the templates. In the settings.py file, navigate to "TEMPLATES", here, change the following line:

```
'DIRS': []
```

To the following :

```
'DIRS': [os.path.join(BASE_DIR,'templates')],
```

Now we can start using templates! Create a home.html file inside the templates folder and include the following line in it :

```
<h1>Hello World!!!</h1>
```

Now in the views.py file of our app, instead of returning a HttpResponse, we can call this html page. For this we'll have to import a function called render from the 'django.shortcuts' library.

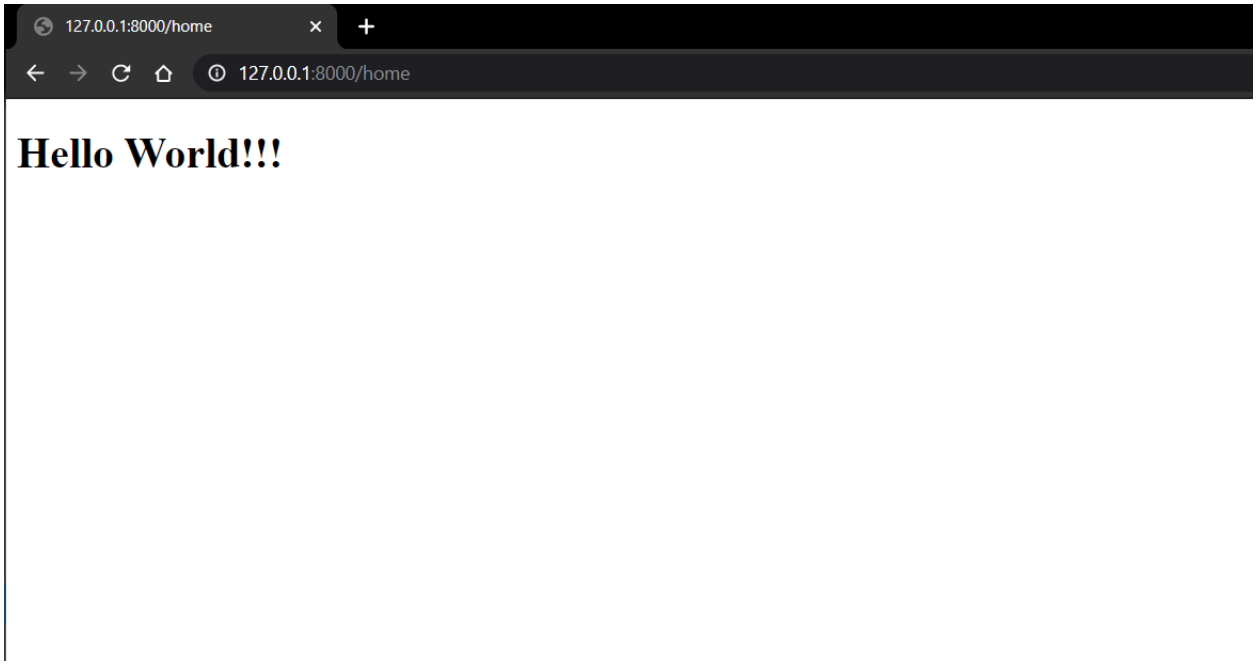
The render function takes a minimum of two arguments, request and the name of the HTML file we'd like to call.

So now our views.py file will look like this :

```
from django.shortcuts import render

# Create your views here.
def home(request):
    return render(request,'home.html')
```

So now when we go to 127.0.0.1:8000/home we will see the following :



The Django template language

A Django template is a text document or a Python string marked-up using the Django template language. Some constructs are recognized and interpreted by the template engine. The main ones are variables and tags.

A template contains variables, which get replaced with values when the template is evaluated, and tags, which control the logic of the template.

Variables

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

Variables look like this: **{{ variable_name }}**. When the template engine encounters a variable, it evaluates that variable and replaces it with the result. Variable names consist of any combination of alphanumeric characters and the underscore (**"_"**) but may not start with an underscore, and may not be a number.

Here is a small example on how you can check the working of variables :

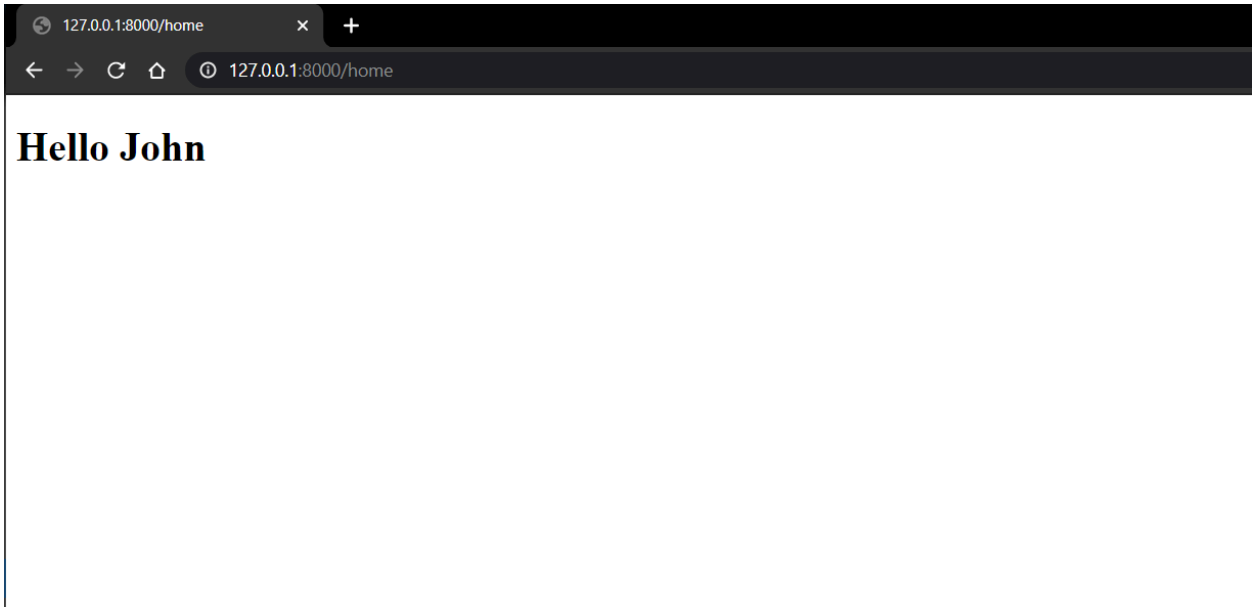
In our views.py file inside the home app, in the return statement, we can add another argument to the function call of the function render which is a dictionary that consists of keys, which are the variable names and the values being the variables we'd like to pass to our HTML page. For example, If we change the return statement to:

```
return render(request, 'home.html', { 'name' : 'John' })
```

So now we can access the string 'John' by using the variable name, 'name'.
Now change the home.html file to the following:

```
<h1>Hello {{name}}</h1>
```

So now when we refresh our web page, we will see the following :



This is how we can have dynamic content on our HTML pages. Every variable we pass from views.py to the HTML page should be inside one dictionary, that is being passed as an argument when we call the render function.

Note :
If you are using Visual Studio Code as your IDE, you will have to instal the Jinja extension from the marketplace. This extension adds language colorization support for the Jinja template language to VS Code. Jinja is a Django built-in Backend. Backend is a dotted Python path to a template engine class implementing Django's template backend API.

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 %}. Let's look at a few examples to understand tags in detail.

- One of the fundamental concepts that we come across while learning a programming language are the if-else statements. DTL allows us to use this if-else logic in our templates, just like we do in python. Here's a look at it :

```
{%  if condition_is_true %}
        <h1>Execute the html commands</h1>
{% endif  %}
```

  You can also have multiple elif blocks, just like in python. Try them out and get yourself used to it!

- Let's see another important fundamental concept that we see in programming languages, a For loop! Again, we can use For loops in our templates using DTL, just like we do in python. Let's see an example where we would like to print the elements of a list as an unordered list, inside a template :

```
<ul>
      {%  for item in given_list %}
            <li>{{ item }}</li>
      {% endfor %}
</ul>
```

Head over to this page to check the official documentation of all the built-in tags that are available in Django.

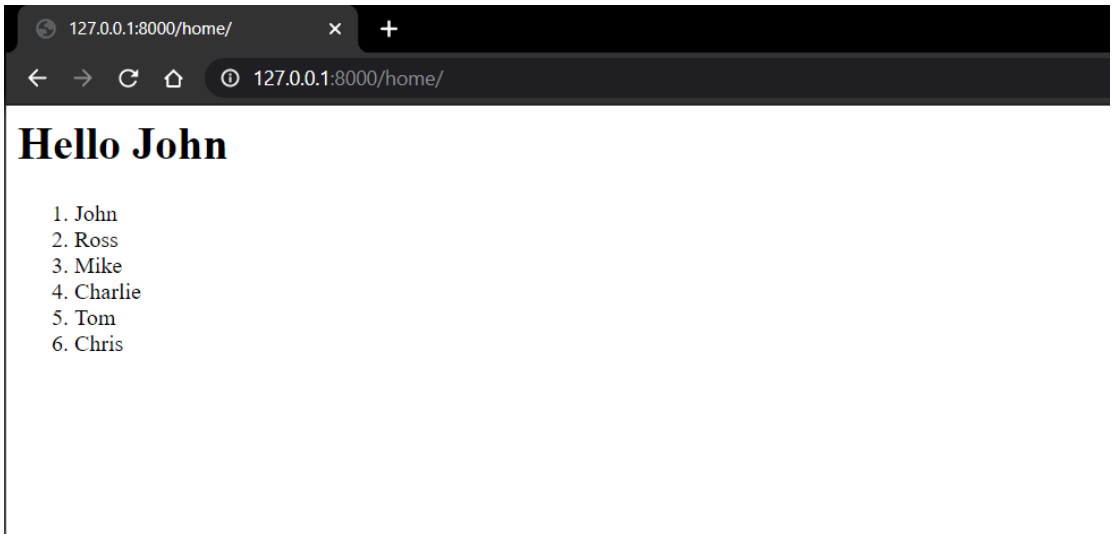
Let's use these above concepts to generate an ordered list on our webpage.
First, we'll have to pass a list to the template, from the home function in the home.views file. So lets change our home function to the following :

```
def home(request):
    student_names = ['John', 'Ross', 'Mike', 'Charlie',
'Tom','Chris']
    return render(request, 'home.html',{'name' : 'John','names' :
student_names})
```

Here, we are passing a list of names of students in a class as a variable, when we are rendering the ‘home.html’ template. This list can be accessed from the template using ‘{{ names }}’. Let’s use this list to generate text on our web page. In the home.html template, add the following :

```
<h1>Hello {{name}}</h1>
<ol>
    {% for item in names %}
            <li>{{ item }}</li>
    {% endfor %}
</ol>
```

This code will print the names in the list we have passed, as an ordered list. Just like this :



These are two basic cases where we have used Variables and Tags that are two important things the Django Template Language has to offer. There are still Filters and Comments which are a part of DTL.

Filters

Filters transform the values of variables and tag arguments.

They look like this:

```
{{ django|title }}
```

With a context of **`{'django': 'the web framework for perfectionists with deadlines'}`**, this template renders to:

```
The Web Framework For Perfectionists With Deadlines
```

Some filters take an argument:

```
{{ my_date|date:"Y-m-d" }}
```

Here is the link for the documentation of built-in Filters in the DTL.

Comments

Comments look like this:

```
{# this won't be rendered #}
```

A **`{% comment %}`** tag provides multi-line comments.

A Django template is a text document or a Python string marked-up using the Django template language. Variables, Tags, Filters and Comments are the four constructs that are recognized and interpreted by the template engine. Do check out the official documentation of the Django Template Language here .

Template Inheritance

Usually, we'd like our website to have multiple webpages, which follow the same theme and have the same layout and color scheme as well. Since this is fairly common in most of the websites, Django has a feature which would allow us to have single HTML and CSS files for all the webpages following the same layout. We can achieve this using **block** and **extends tags**. Let's have a look :

- First lets create a HTML file which will have the basic layout. This file will be called the ‘Parent’ template. Create a ‘base.html’ file inside the templates folder we have created earlier.
- Inside this ‘base.html’ file, add the following code:
  (While writing html code, be sure to use the ‘Emmet’ feature, which will give us the option to generate the basic structure of a HTML file, just by typing ‘!’ inside a HTML file and selecting the option from the drop-down list that appears.)


```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Base Template</title>
</head>
<body bgcolor="cyan">
    {% block content %}

    {% endblock %}
</body>
</html>
```

This ‘Parent’ template will add a background color ‘cyan’ to all the ‘Child’ templates which will be inheriting this template.

- As we can see, we have used `{% block content %}` and `{% endblock %}` inside which the code from the child templates, which are inheriting this parent template, will be inserted. Let’s see how to achieve that:

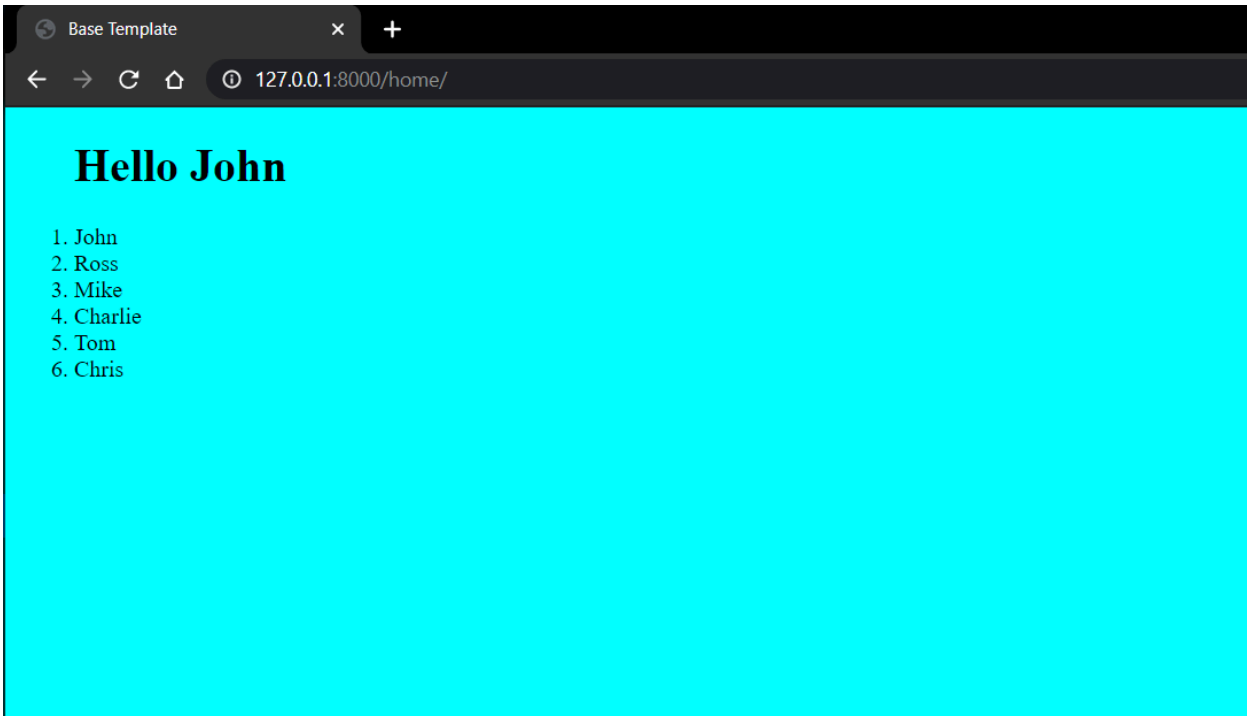
  All we need to do is to go to the template that we’d like to have this layout and add the line {% extends ‘<name of parent template>.html’ %} as the first line

of the file, and we also need to have the code we'd like to be used in the block, between the `{% block content %}` and `{% endblock %}` tags as well. Let's do this to our 'home.html' template, which should now look like this :

```
{% extends 'base.html' %}
      {% block content %}
      <ol>
          <h1>Hello {{name}}</h1>
          {% for item in names %}
                  <li>{{ item }}</li>
          {% endfor %}
      </ol>

      {% endblock %}
```

- So now when we run our server, we will see the following webpage :



This feature is called Template **Inheritance .** This is the most powerful and the most complex part of Django's Template Engine. This feature helps us maximise 'Code Reuse'. This is the link where you can check the official Django documentation of the template inheritance concept.

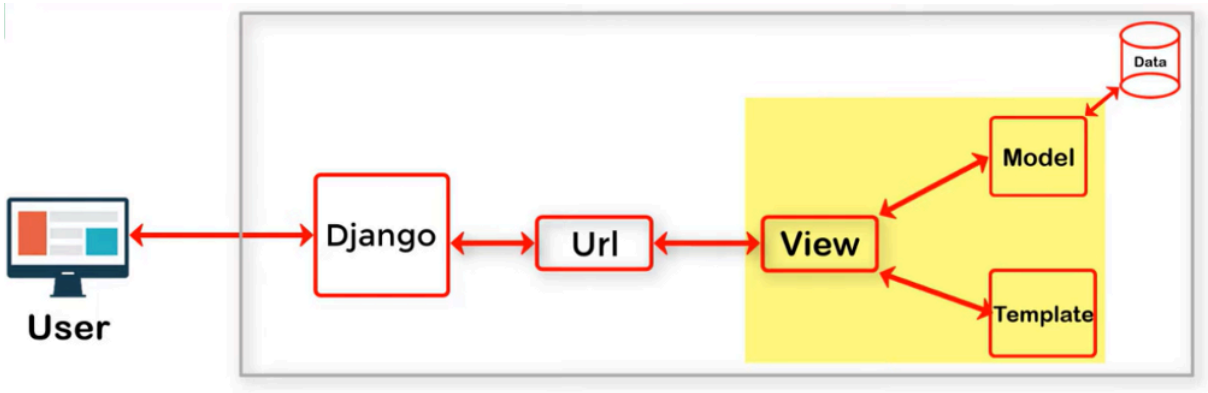
This is where we come to the end of this week's post. To summarise, you have seen how the content on a webpage is generated, talked about the Django Template Language, and ended it with the concept of Template Inheritance. I have also provided the links of official documentation of concepts, wherever possible, be sure to go through them and try and execute the topics I may have skipped over.

Django Application

Let's make a quick recap of what happens when the user sends a request to fetch the data. This request will go to the web application. Since we are using Django, the request will go to the Django framework. Whenever we build a project using the Django framework we get a specific file called URLs. In the urls.py file, we do the mapping right from where to get the request and where to send the request, and this navigation will go to views. You may get a doubt, What are views ?. So the business logic that we want to write is done in views and then these views use the model object to fetch the data and the fetched data is displayed by the template.

You will get more clarity by looking at the following MVT diagram.

Model View Template



Let's make a web application using Django for the sum of two numbers, where we create a home page consisting of a form with two input fields and a submit button and we display the addition of two numbers in the result page.

To create our project run the command:

```
django-admin startproject django_application
```

After Creating the project move into the project directory and run the server by using the following command, and You will find a successful message

```
python manage.py runserver
```

To create our application run the command:

```
python manage.py startapp addition
```

After creating the application we need to Configure our application by adding it to the INSTALLED_APPS list in the settings.py file.

```
INSTALLED_APPS = ['addition.apps.AdditionConfig',
'django.contrib.admin','django.contrib.auth','django.contrib.contenttypes','dja
ngo.contrib.sessions',django.contrib.messages','django.contrib.staticfiles',]
```

Then create a templates folder in the project directory with two Html files

1. home.html (for displaying the form)
2. result.html (for displaying the addition of two numbers)

Then add the path of the templates folder to the TEMPLATES list in the settings.py file as follows

```
TEMPLATES = [{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [os.path.join(BASE_DIR,'templates')],

'APP_DIRS': True,

'OPTIONS': {

'context_processors': [

'django.template.context_processors.debug',

'django.template.context_processors.request',
```

```
'django.contrib.auth.context_processors.auth',
'django.contrib.messages.context_processors.messages',],},},
]
```

Create a form in home.html as follows

```
<div class="form-action"  style="width:300px">
<form action="result" method="POST" >
{% csrf_token %}
<label for="num1">Enter the Number 1 </label>
<input type="text" class="form-control" name='num1'>
<br>
<label for="num2">Enter the Number 2</label>
<input type="text" class="form-control" name='num2'>
<br>
<input type="submit" value="Add">
</form>
</div>
```

Note: Whenever we create a form in Django we need to use the csrf token inside it and the method must be POST.

To the result.html file add the following jinja code.

```
Result={{result}}
```

Create a urls.py file in addition directory and add the path of views to it as follows

```
from . import views
from django.urls import path
from django.urls.conf import include
urlpatterns = [
path('',views.home,name='home'),
path('result/',views.result,name='result'),
]
```

Now Let's Create views for both template files in the views.py file of the addition folder as follows

```
def result(request):
    num1=int(request.POST['num1']);
    num2=int(request.POST['num2']);
    return render(request,'result.html',{'result':num1+num2});
def home(request):
    return render(request,'home.html');
```

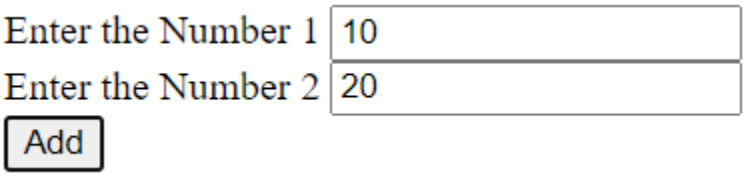
As we have seen earlier in Model View Template Design, After URL mapping request is sent to views and now these views are used to render the template files.

we need to add our application URLs path to the URL patterns list in the project urls.py file as follows. Since we need our application as the home page of our project we use empty codes.

```
from django.contrib import admin
from django.urls import path
from django.urls.conf import include
urlpatterns = [
path('admin/', admin.site.urls),
path('',include('addition.urls')),
]
```

Finally, we have done with the application

Home Page



Result Page