Django

- Follows the **MTV** (Model-Template-View) Design Pattern. The ‘view’ is the business logic layer and acts more or less like the controller. A bridge between the model and the template.
- High level (Flask would be more low level)
- Usually helps you do the things the best way possible. Less freedom compared to Flask but less chances to screwing up

Despite having its own nomenclature, such as naming the callable objects generating the HTTP responses "**views**",the core Django framework can be seen as an MVC architecture. It consists of an object-relational mapper (ORM) that mediates between data models (defined as Python classes) and a relational database ("**Model**"), a system for processing HTTP requests with a web templating system ("**View**"), and a regular-expression-based URL dispatcher ("**Controller**").

Also included in the core framework are:

- a lightweight and standalone web server for development and testing
- a form serialization and validation system that can translate between HTML forms and values suitable for storage in the database
- a template system that utilizes the concept of inheritance borrowed from object-oriented programming
- a caching framework that can use any of several cache methods
- support for middleware classes that can intervene at various stages of request processing and carry out custom functions
- an internal dispatcher system that allows components of an application to communicate events to each other via pre-defined signals
- an internationalization system, including translations of Django's own components into a variety of languages
- a serialization system that can produce and read XML and/or JSON representations of Django model instances
- a system for extending the capabilities of the template engine
- an interface to Python's built-in unit test framework

The Concept of Apps

Another unique thing about django is it uses the concept of ‘**apps**’. Usually we think of an entire project as an app but within django a single project or website can have multiple apps, for example a blog app, a client app to handle client functions, a store app and so on. Of course we could also have a single app for the website.

We can also move apps from one project to another. Using other peoples apps easily. Its easy, flexible and scalable, reusable.

Python Virtual Env

This gives us a virtual environment for all of our python instances. This is recommended way to develop Django apps. Creates isolated environments with their own directories. Separates your Django project instances.

Installation

1. I somehow downloaded **pip**. Then installed **virtualenv** with pip. Then installed **django**.
2. Create a directory, change to it. Create a new virtual enviroment there called "**env**"

   virtualenv env
3. Now, activate the virtual environment with the following command:

   . env/bin/activate
4. If virtualenv is set to a wrong Python version we can set it up like this:

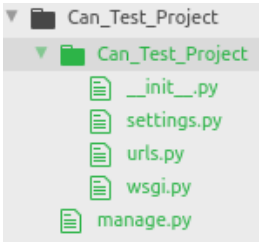
   virtualenv --python=python3.6 myvenvname
5. Create a new django project :

   django-admin startproject Can_Test_Project

also changed the manage.py first line to **#!/home/can/Desktop/django-app/env/python3.6 python**

sudo apt-get install python3-pip [I guess django is installed to each specific python version]

pip3 install Django [this solved some problems for me, installed it to python3.6 (?)]



→ In our project folder we have another folder which contains all of our code and a **manage.py** file

→ **manage.py** is the CLI (Command Line Interface) client that we will use for doing different things. It's basically a wrapper of **django-admin**. (We used django-admin to create our application). Manage.py is basically the same thing but localized for this project. With the actual code in that file we dont need to do anything.

→ In our folder we have a couple files. **__init__.py** : This file is empty by default. It is required to be there for some things. Having it here gets rid of some confusion that has to do with file names and modules and things like that.

→ **settings.py** has all of our settings. Things like secrey key (needs to be kept secret), base directory etc. Any app that we create we need to put it in INSTALLED_APPS array. Settings about databases, atm it is set to sqlite3 (not preferable for production apps)

→ **urls.py**, this is basically routing. For the url patterns we will use regular-expressions. Each app will have its own urls file. Setting things like if a url starts with '**admin/**' go to admin.site.urls. Each app will have its own urls file(?)

→ **wsgi** is the primary development platform for django(?). It's basically the common standard for web-servers and applications. Its a simle default wsgi configuration created when we created our project.

Starting the Server

Running the command **`python manage.py runserver`** will run the server. (If it will say django not found then retry it with python2 manage.py runserver). If everyting works we will be able to access our site from the port 8000. http://127.0.0.1:8000/

- when we run that command we see this message : You have 13 **unapplied migration(s)**. Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions. Run 'python manage.py migrate' to apply them.

Migrations are basically files that will do things to our database (creating tables, add data etc.).

Unapplied migrations means there are migrations waiting that have to do with admin, auth, ... We haven't created them in our database yet. We could apply them now by running the command above but that will create them in the sqlite database which we dont want. We want to use MySql.

Installing MySql client for Python and setting a superuser account

While in our enviroment (env), we can install it with **`pip install mysqlclient`**

I was getting errors and first I ran this command **sudo apt-get install python3.6-dev libmysqlclient-dev**, after that I did pip install mysqlclient and it worked.

→ I started lampp, opened phpmyadmin and created a new database called djangoproject

→ Not we can go to our settings.py folder and change the DATABASES setting. Everyting should be setup so we can run our migration and all those 13 default migration should go into this database.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'djangoproject',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': '127.0.0.1',
        'PORT' : ''
    }
```

→ **`python3.6 manage.py migrate`** did the trick. When we now look at our database we will see the different tables are added.

'HOST' : 'localhost' was problematic, '127.0.0.1' fixed it as usual.

→ Now when we run our server again with python3.6 manage.py startserver, and go to localhost:8000/admin we can see a new admin backend interface.

→ Right now we dont have any user to login. We can use manage.py to create a superuser so we can login here. **python3.6 manage.py createsuperuser --username can --email ozbek14@yahoo.com**, then enter a password. And a superuser is created

→ We can login as superuser to our server and configure groups and users, see activities and so on. We didn't even write any code and we got all of this functionality out of the box == rapid development

Creating Apps

We will create a blog/post app. To create a new app called 'posts': **`python3.6 manage.py startapp posts`**. We have a new folder in our project structure now.

Admin.py has to do with adding models to your admin interface.

Apps.py is app specific settings.

Models.py is where we create our models, we will have a post model.

Test.py is for testing.

Views.py is basically our controller, where we can load views or templates, interact with our model and so on.

- We have to add this app to the INSTALLED_APPS array in the settings.py folder. Adding 'posts' to that array
- Also adding it to the **root/urls.py** folder (on the right)
- We need to create **urls.py** in the posts directory

```
from django.conf.urls import url, include
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^posts/', include('posts.urls')),
]
```

The main urls.py file says that anything starting with posts/ will be passed off the to the urls file in the posts app.

In that **posts/urls.py** file we say if the URL is .../posts and if nothing is after that then call the index function from the views file.

When we call localhost:8000/posts, we get back "Hello from posts".

This was just for testing, in the next step we will render a template

urls.py

```
from django.conf.urls import url
from . import views

urlpatterns = [
    # ^ start with,  $ end with : nothing
    # look in our views.py file for a function called index
    url(r'^$', views.index, name='index')
];
```

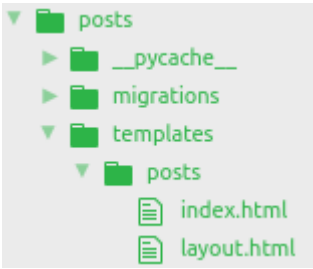
views.py

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

# Create your views here.
def index(request):
    return HttpResponse('HELLO FROM POSTS')
```

Rendering a Template

We want to render a html template and return it back, not just static text.



- The **render** function looks to folders with the name 'templates'. In order to prevent conflicts, it is good practice to create a templates folder for each app and in those folders another folder with the app name, putting the templates inside that.

Here we created '**templates**' folder in our '**posts**' directory and in templates another new directory called '**posts**' again. This is also suggested in the django documentation.

- Now we want to return an html file called 'index.html', we pass it to the render function. Now when we request .../posts, we get index.html back as a response

views.py

```
def index(request):
    #render looks into all folders
    # with the name 'templates'
    return render(request, 'posts/index.html')
```

```
<h1>INDEX HYTTTTML</h1>
```

Templates

But we want to generate dynamic html pages. This is where html templates comes into play. Here we will use **Jinja**, it is **a template engine** for **Python**.

We create a template page called '**layout.html**'. This will be the layout for our posts pages. It will be standard html with css and js. Just the tags from Jinja will be added, it will make it dynamic.

- Lets assume that we created a new blog-post and we want to put it to that placeholder on our layout page. Lets put index.html there.

layout.html

```
<body>
  <header class="container center-align">
    <h1>TraversyMedia Blog</h1>
  </header>

  <div class="container">
    {% block content %}
    {% endblock %}
    <br>
```

This way we can pass dynamic data into our templates.

We are now describing what will go into the block in the layout file. Starting with some logic. Telling this extends a particular template, giving a path to the template which will be extended, starting from the 'templates' directory. In this example '**posts/layout.html**' file will be extended.

index.html

```
{% extends 'posts/layout.html' %}

{% block content %}
<h1>Latest Posts</h1>
{% endblock %}
```

- Note that even when we will have many apps. Other apps can extend this layout.html template even though it is not included in the templates directory of that specific app, it **DOES NOT MATTER**. Django is going to load them all, considering them all in one big **/templates** directory.

When we call **localhost:8000/posts** now. Again everything will be the same, django will first look to **root/urls.py**, from there it will go to **posts/urls.py**, then to **posts/views.py** calling the **index function** there. In our layout we will include the things in **index.hml** and return it as a response. So at the end we will see a page, TraversyMedia Blog, under that 'Latest Postst'.

→ For short snippets of code, another jinja logic name '**includes**' can be used, also need to create a directory called 'includes'. Makes it more organized and cleaner.

Extends: Generally we will extend pages that will be present all over our website. We would extend a page that is like our header and footer of our website, something that will be there all the time.

Includes: Can also be present in a lot of places but not necessarily everywhere.

Passing dynamic data to our templates from our Model

Adding another parameter to the render function, a dictionary. It will output the same page. Variable

views.py

```
def index(request):
    #render looks into all folders
    # with the name 'templates'
    return render(request, 'posts/index.html', {
    'title': 'Latest Posts1'
    })
```

index.html

```
{% extends 'posts/layout.html' %}

{% block content %}
<h1>{{title}}</h1>
{% endblock %}
```

This is how we insert our blog posts. We will create a model, get the data from the model and then we will pass it into the view like this.

models.py

```
from django.db import models
from datetime import datetime

# Create your models here.
class Posts(models.Model):
    title = models.CharField(max_length = 200)
    body = models.TextField()
    created_at = models.DateTimeField(default = datetime.now, blank = True)
```

```
python3.6 manage.py makemigrations posts
```

```
Migrations for 'posts':
  posts/migrations/0001_initial.py
    - Create model Posts
```

→ We created our Model in posts/models.py and migrated it. It created a migrations file. This is just a file, it didnt create the database table yet.

→ To create the database table we need to enter the following command:

```
python3.6 manage.py migrate
```

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, posts, sessions
Running migrations:
  Applying posts.0001_initial... OK
```

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
☐	1	**id** 🔑	int(11)			No	*None*		AUTO_INCREMENT
☐	2	**title**	varchar(200)	utf8mb4_general_ci		No	*None*		
☐	3	**body**	longtext	utf8mb4_general_ci		No	*None*		
☐	4	**created_at**	datetime(6)			No	*None*		

Adding Post Functionality to our Admin Panel

Right now when we login as admin, we only see 'Groups' and 'Users'. But our goal is to also add 'Posts' and manage our posts from there. Adding new ones, modifying and removing it. We have our model class it is mapped to our table in our MySql database.

→ To do this we have to go to our **posts/admin.py** file.

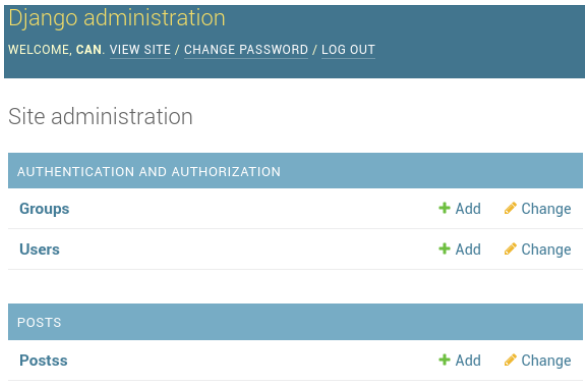
admin.py

```python
from django.contrib import admin
# Register your models here.

# Importing the class 'Posts'
# from the models.py file
from .models import Posts

admin.site.register(Posts)
```

→ Just writing this line, we can see that our admin panel has changed.

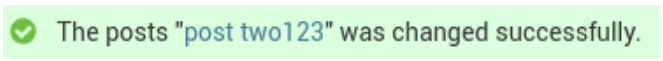


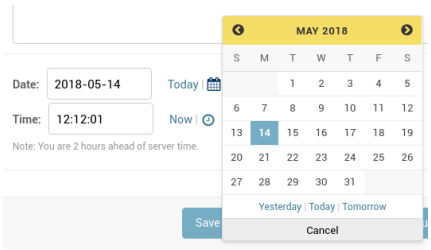
We can now **Add new posts** from this panel, **change the existing posts** and **delete posts** easily from the admin interface. All will be executed directly in our database.

→ New posts all have the name '**Posts Object**'. We can change it by overriding the **__str__** function in our model. So now our in our Posts tab in the admin interface all of the posts are organized showing the name of each one.

```python
def __str__(self):
    return self.title
```

➢ This is one of the perks of Django, all of the interface components are already there out of the box. Thats why it is great for rapid development. The admin interface offers a lot of funtionality, with other frameworks we would have to write it our own. Like add posts, remove, delete it, the date and time checkboxes where a calendar pops up. The flash messages above after a successful operation and so on...





A little problem is that django automatically adds an 's' at the end of our model, thats why we have 'Postss'. It's really easy to fix.

→ Go to our **models.py** file

```python
from django.db import models
from datetime import datetime

# Create your models here.
class Posts(models.Model):
    title = models.CharField(max_length = 200)
    body = models.TextField()
    created_at = models.DateTimeField(default = datetime.now, blank = True)
    class Meta:
        # notice how this Meta is indented!!!
        # We can add different meta values
        # Defining what our plural name for this model will be
        verbose_name_plural = "Posts"
```

Displaying Posts on our Website

Going to our **posts/views.py** file

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

# Create your views here.
def index(request):
    # Getting back the first 10 Posts
    posts = Posts.objects.all()[:10]
    context = {
        'title': 'Latest Posts',
        'posts': posts
    }
    return render(request, 'posts/index.html', context)
```

index.html

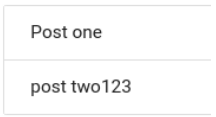
```
{% extends 'posts/layout.html' %}

{% block content %}
<h1>{{title}}</h1>

<ul class="collection">
{% for post in posts %}
  <li class="collection-item">{{post.title}}</li>
{% endfor %}
</ul>
{% endblock %}
```

→ İlk 10 postu databaseden geri aldık, onu context dictionarysine 'posts' anahtarı ile verdik, o dictionaryi de render fonksiyonuna argüman olarak verdik.

→ index.html e gidicek bu context dictionarysi. Orda ise yeni bir unordered list oluşturduk, ve jinja logic ile posts'un içindeki her bir post için yeni bir list item oluştur ve onun içindexi text de, o an baktığın postun title'ı olsun.



Bu tamam, şimdi list itemlerine tıklayarak postun içeriğini göstermek istiyoruz.

urls.py

```python
from django.conf.urls import url
from . import views

urlpatterns = [
    # ^ start with,  $ end with : nothing
    # look in our views.py file for a function called index
    url(r'^$', views.index, name='index'),
    # ?P<id> represents a parameter called id,
    # \d+ atleast one or more digits
    # if this URL pattern is called: call the details function in views.py
    url(r'^details/(?P<id>\d+)/$', views.details, name='details')
    ];
```

views.py

```python
def details(request, id):
    # We want to fetch just one post
    post = Posts.objects.get(id=id)

    context = {
        'post' : post
    }
    # loading our template, passing our wanted data in it.
    # jinja will get our data and construct the html file for us.
    return render(request, 'posts/details.html', context)
```

details.html

```
{% extends 'posts/layout.html' %}

{% block content %}
<h1>{{post.title}}</h1>

<div>
    {{post.body}}
  </br></br>
    {{post.created_at}}
</div>
{% endblock %}
```

→ **posts/urls.py** da dedik ki: eğer **localhost/posts/details/12** gibi bir request gelir ise, views.details fonksiyonunu çağır.

→ views.py da yeni bir fonksiyon yazdık, request ve girilen id'yi parametre olarak alıyor. Databaseden o id olan postu alıyor ve o verileri templatetimize yüklüyor/gönderiyor.

→ Jinja sayesinde template'imizin içine logic yazabiliyoruz. Burada dediğimiz ise, h1'in içine verilen postun titleını yaz, bir tane divin içine de postun bodysini ve altına ne zaman oluşturulduğunu yaz.

What is WSGI?

WSGI is the **Web Server Gateway Interface**. It is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request. WSGI is a **Python standard** described in detail in PEP 3333.

Django and Flask are examples of frameworks that run on WSGI.

How to use Django with Apache and **mod_wsgi**

Deploying Django with Apache and mod_wsgi is a tried and tested way to get Django into production.

mod_wsgi is an Apache module which can host any Python WSGI application, including Django. Django will work with any version of Apache which supports mod_wsgi.

The official mod_wsgi documentation is your source for all the details about how to use mod_wsgi. You'll probably want to start with the installation and configuration documentation.

Setting up with Nginx

https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-16-04

Adjust the Project Settings: The first thing we should do with our newly created project files is adjust the settings. Open the settings file in your text editor: **nano ~/myproject/myproject/settings.py**

Start by locating the ALLOWED_HOSTS directive. **This defines a list of the server's addresses or domain names may be used to connect to the Django instance**. Any incoming requests with a Host header that is not in this list will raise an exception. Django requires that you set this to prevent a certain class of security vulnerability.

In the square brackets, list the IP addresses or domain names that are associated with your Django server. Each item should be listed in quotations with entries separated by a comma. If you wish requests for an entire domain and any subdomains, prepend a period to the beginning of the entry. In the snippet below, there are a few commented out examples used to demonstrate:

~/myproject/myproject/settings.py

```
. . .
# The simplest case: just add the domain name(s) and IP addresses of your Django ser
# ALLOWED_HOSTS = [ 'example.com', '203.0.113.5']
# To respond to 'example.com' and any subdomains, start the domain with a dot
# ALLOWED_HOSTS = ['.example.com', '203.0.113.5']
ALLOWED_HOSTS = ['your_server_domain_or_IP', 'second_domain_or_IP', . . .]
```

→ Also a lot of other things has to be configured. Things with **Gunicorn** and **Nginx**

The **Gunicorn** "Green Unicorn" is a Python Web Server Gateway Interface (WSGI) HTTP server. It is a pre-fork worker model, ported from Ruby's Unicorn project. The Gunicorn server is broadly compatible with a number of web frameworks, simply implemented, light on server resources and fairly fast.[2]

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Deployment

→ Sentdex logged in his newly created server Ubuntu + Django installed on VPS (with Digital Ocean). When he first logs in the message he gets :

Nginx listens on public IP (162.243...) port 80 and forwards requests to Gunicorn on port 9000. Nginx access log is in */var/log/nginx/access.log* and error log is in */var/log/nginx/error.log*. Gunicorn is started using an Upstart script located at */etc/*init/gunicorn.conf. To restart your Django project, run : sudo service gunicorn restart.