Table of Contents

Getting started with Django

We assume you have the latest version of python installed. And also Django installed.

Starting a project

Let's create a project called mysite.

```
django-admin startproject mysite
```

This command will create a folder called mysite with all the files required to get our project started.

Let's cd into our directory to start doing some work.

```
cd mysite
```

Open the settings.py file. Modify the TIME_ZONE to wherever you are from.

```
TIME_ZONE = 'Asia/Kolkata'
```

Add a path for static files

```
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

If you are hosting your site on pythonanywhere.com. Make sure it is on the list of allowed hosts.

```
ALLOWED_HOSTS = ['pythonanywhere.com']
```

To create a database

```
python manage.py migrate
```

Starting the web server

```
python manage.py runserver
```

The website can be accessed at http://localhost:8000/

Creating an app

A project is a collection of many applications. We can think of them as modules.

To create a new app

```
python manage.py startapp myapp
```

This creates a new folder within our project folder with all the files required for the app.

We need to add the name of the app in our project settings.py file.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    ...,
    'myapp'
]
```

Django Uses the **Model-View-Template** framework

Model is a way to store our data.

View extracts data from the model, and passes it on to the template

Template displays the data.

The admin interface

We can access the admin interface at http://localhost:8000/admin

It allows to access the models (Data) in our project

But first, we need to create a superuser to be able to access our admin interface

```
python manage.py createsuperuser
```

The admin interface will enable you to perform all the basic CRUD (Create, Read, Update, Delete) operations.

To make sure that we can access our models in our admin interface, we need to register them in the **admin.py** file.

Views

A view is a python function that takes in a request and returns a response.

The request object contains metadata about the request. The first argument to the view function is a request. Each view is responsible for returning a response object.

Create a folder called *templates* inside myapp folder. Write a hello.html file inside templates. (Just any basic HTML will do).

In *myapp/views.py*

```
from django.shortcuts import render

def hello(request):
    return render(request, 'hello.html', {})
```

The final parameter {} is a dictionary containing all the variables that we want to pass to our template. A dictionary is nothing but an unordered collection of key-value pairs.

Create a file called urls.py in myapp folder. This is for all the urls used in myapp.

myapp/urls.py

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

urlpatterns = [
    url(r"^hello/", views.hello, name = 'hello')
]
```

You need to make sure that the main urls.py in your project folder knows all the urls in myapp application.

mysite/urls.py

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

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

Template Language

We have learnt to server up a static page on our Django server.

Let's add some dynamic content. This is where we interpolate our HTML file with some python variables. We pass the python variables though a dictionary. A dictionary is nothing but an unordered collection of key-value pairs.

myapp/views.py

```
from django.shortcuts import render
import datetime

def hello(request):
    today = datetime.datetime.now().date()
    return render(request, 'hello.html', {'today': today})
```

In the *myapp/templates/hello.html*

We can interpolate the variables within curly braces.

```
<p> The date today is {{ today }} </p>
```

We can perform logic and loops inside our templates also.

Models

A model is an object that is stored in the database.

In *myapp/models.py* file

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    title = models.CharField(max_length = 100)
    content = models.TextField()
    created_at = models.DateTimeField(default=timezone.now())
    author = models.ForeignKey('auth.User')

    def publish(self):
        self.save()

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

We have created a template called Post for a simple object that will have a title and some content. The publish calls the save method which saves the object in the database. The str method, is a way to identify our objects. In this case, the Post objects will be referenced by their title.

Now we need to add our new model to our database,

```
python manage.py makemigrations myapp
```

Now, python has prepared a migration file, that we need to apply to our database,

```
python manage.py migrate myapp
```

To make sure we can access this model through the **admin interface**,

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

Now let's display the posts in our webpage, we do through views and templates.

In myapp/views.py

```
from .models import Post

def posts(request):
    posts = Post.objects.all()
    return render(request, 'posts.html', {'posts': posts})
```

In myapp/templates/posts.html

```
{% for post in posts %}
    <h1>{{ post.title }}</h1>
    <p>{{ post.content }}</p>
    <p>{{ post.created_at }}</p>
    <p>{{ post.author }}</p>
{% endfor %}
```

In myapp/urls.py

```
urlpatterns = [
    url(r"^hello/", views.hello, name = 'hello'),
    url(r"^posts/", views.posts, name = 'posts'),
]
```

Regex

URLs are mapped using Regex (Regular expressions)

Regular expressions start with r and are contained within " " (quotes)

Regular expressions follow certain rules

```
^ signifies starting of the text
$ signifies ending of the text
\d signifies a digit
+ signifies that the previous item should be repeated
() is used to capture part of the pattern
```

Imagine we individual urls for posts such as *http:localhost/post/123/*

Now to write a view for each url would be cumbersome, so we map views with urls created by regex, and we can extract the number from the url, and pass it to the view.

```
r"^post/(?P<val>\d+)/$"
```

The above expression, would match something like http://localhost/post/123/

The value '123' is captured and stored in the variable named val, and passed to the view.

Query sets

We can start the django shell with the below command:

```
python manage.py shell
```

This starts the console >>

```
>> from myapp.models import Posts
>> Post.objects.all()
```

The above command returns all the Posts saved in the database.

We can filter the objects based on a criteria

and We can also order/sort the objects based on a criteria

```
post = Post.objects.get(pk = num)
```

The above line, retrives a post according the primary key. We can pass it on the template, to display it.

```
me = User.objects.get(username='rishi')
Post.objects.filter(author=me)
```

URL Mapping

Each URL is mapped to a view.

Let's suppose we want to show our post in detail on it's own page/url.

Let's put a 'See more' link below each post, which will show the entire post on it's own page.

In *urls.py*

```
url(r"^post/(?P<num>\d+)/$", views.post_detail, name = 'post_detail')
```

We capture the pattern from the url, and name it num, and pass it on the view.

In *posts.html*

```
{% for post in posts %}
    <h1> {{ post.title }} </h1>
    <a href={% url 'post_detail' pk=post.pk %}"> See more </a>
{% endfor %}
```

Here, pk refers to primary_key. We didn't explicitly create a primary_key, Django created one for us automatically. It is the column that increases by 1 for each record. 1, 2, 3 and so on. It is sort of like an index number for our posts.

In *views.py*

```
def post_detail(request, num):
    post = Post.objects.get(pk = num)
    return render(request, 'post_detail.html', {'post': post})
```

In *post_detail.html*

```
<title> Post #{{post.pk}} </title>
<h1> {{post.title}} </h1>
<a href="{% url 'posts' %}"> Go back </a>
```

Django Forms

Create a file called *forms.py* in myapp folder

We can create a form from scratch, but here we will use a ModelForm which will save the result of the form to the model.

myapp/forms.py

```
from django import forms
from .models import Post

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['title', 'content']
```

In templates folder, create a file called post_new.html

```
<h1> New Post </h1>
<form method="POST">
    {% crsf_token %}
    {{ form.as_p }}
    <button type="submit"> Save </button>
</form>
```

In *myapp/views.py*

```
from django.shortcuts import render, redirect
from .forms import PostForm

def post_new(request):
    if request.method == "POST":
        form = PostForm(request.POST)
        if form.is_valid():
            post = form.save()
            return redirect('post_detail', pk=post.pk)
    else:
        form = PostForm()
    return render(request, 'post_new.html', {'form': form})
```

request.POST is a dictionary like object containing the form data.

In *myapp/urls.py*

```
url(r"^new/", views.post_new, name = 'new')
```

```
if form.is_valid():
    post = form.save(commit=False)
    post.author = request.user
    post.published_date = timezone.now()
    post.save()
```

In the above example, we don't save the form data to the database right away. We add certain additional info, before saving.

If the form has image or file upload then, we need to set

```
<form enctype="multipart/form-data">
```

To receive the files in the view, we need to use REQUEST.FILES

```
form = PostForm(request.POST, request.FILES)
```

If a field is optional, then in the forms.py

```
class PostForm(forms.ModelForm):
    #The picture field is not mandatory
    picture = forms.ImageField(required=False)
    class Meta:
        model = Post
        fields = ['picture','question']
```

Authentication and Authorization

Authentication verifies if a user is who they claim to be. Authorization determines what an authenticated user is allowed to do. The both are somewhat couped and collectively called Authentication.

django.contrib.auth is the built in module to include these features in Django. It comes with all the defaul t auth properties.

Only one class of users exists in Django, superusers are just users with special attributes set.

User object has primarily: username, password, email, first_name, last_name

Inside models, we can do,

```
author = models.ForeignKey('auth.User')
```

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    #A genuine user
else:
    #NOT a genuine user
```

We can use decorators to protect our views, such that only logged-in users can access them

In views.py

```
from django.contrib.auth.decorators import login_required

#Add this line on top of the view
@login_required
```

To login users, we can use the default view that django.auth provides

In mysite/urls.py

```
from django.contrib.auth import views

url(r"^accounts/login/$", views.login, name='login')
```

Create a directory called registration inside templates folder. And create a template called login.html

```
 <form method="post" action="{% url 'login' %}">
    {% csrf_token %}
        <table>
        <tr>
            <td>{{ form.username.label_tag }}</td>
            <td>{{ form.username }}</td>
        </tr>
        <tr>
            <td>{{ form.password.label_tag }}</td>
            <td>{{ form.password }}</td>
        </tr>
        </table>

        <input type="submit" value="login" />
        <input type="hidden" name="next" value="{{ next }}" />
    </form>
```

In settings.py

```
LOGIN_REDIRECT_URL = "/"
```

Inside we can check for user login

```
{% if user.is_authenticated %}
  #Some code
{% else %}
  #Some code
{% endif %}
```

To logout, add the url

```
<a href="{% url 'logout' %> Logout </a>
```

In urls.py

```
url(r'^accounts/logout/$', views.logout, name='logout', kwargs={'next_page': '/'}),
```

The User model is stored in **django.contrib.auth.models**

Template Extending

Suppose you have a lot of HTML that is repeated in all the html files (templates). We can avoid writing repeating pieces of HTML code by template extending.

Create a base.html file in the templates folder

templates/base.html

```
<html>
    ...Some HTML content...
    {% block content %}
    {% endblock %}
</html>
```

Whatever goes between block content and endblock is the content that is unique to a page.

templates/home.html

```
{% extends 'base.html' %}
{% block content %}
    ...Some unique content...
{% endblock %}
```

File Upload

```
file = models.FileField(upload_to='uploads/')
picture = models.ImageField(upload_to='uploads/')
```

These are stored in the database as a string field, as a reference to the actual file. If you delete a record in the database, Django will only destroy the reference and not the actual physical file.

In *settings.py*

```
MEDIA_URL = '/media/
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

In *urls.py*

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    ...Your urls info...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

We need to make sure the form has **enctype = "multipart/form-data"**

To display the image in the template

```
{% if post.picture %}
<img src="{{ post.picture.url }}">
```

To receive the files in the view, we need to use REQUEST.FILES

```
form = PostForm(request.POST, request.FILES)
```

request.FILES is a dictionary like object containg all the uploaded files.

Note: To deal with images, python requires a library called Pillow.

```
sudo pip install pillow
```

Request Object

The request object contains a lot of information that we can access in the view.

For example request.META gives all the header information. It is a python dictionary.

```
def home(request):
    test = request.META
    return render(request, 'home.html', {'test': test})
```

```
<div>
  {% for key,value in test.items %}
    <p>{{key}} : {{value}}</p>
  {% endfor %}
</div>
```

test.items() gives us a list of tuples in the dictionary test. In a template, we don't need to use the () to call a function