Files

settings.py

- [x] **BASE_DIR**
  - [x] Directory where manage.py exists.
  - [x] This allows to work relative to directory.
  - [x] You can try to print BASE_DIR and runserver.
- [x] **SECRET_KEY**
  - [x] Should be unique to each project.
  - [x] Modify few characters if using someone else project.
- [x] **DEBUG**
  - [x] Shows details for debugging
  - [x] Should be changed to False when in production.
- [x] **ALLOWED_HOSTS**
  - [x] Allowed domain names and ips.
  - [x] Used as security measure in production.
- [x] **INSTALLED_APPS**
  - [x] Components used in the whole project.
  - [x] Remember to add all apps you create and also third-party apps you install in this list.
- [x] **MIDDLEWARE**
  - [x] Manages how requests are handled and securities are handled.
- [x] **ROOT_URLCONF**
  - [x] Tells django how to manage routes.
- [x] **TEMPLATES**
  - [ ] Create a templates directory with base.html inside it and respective folders for apps.
  - [x] How are html templates rendered, where are they stored.
  - [x] In DIRS list, add os.path.join(BASE_DIR, "templates").
- [x] **WSGI_APPLICATION**
  - [x] Tells django how to use servers.
  - [x] Sometimes we may need to change it.
- [x] **DATABASES**
  - [x] Which database engine used and where is database stored.
  - [x] By default uses sqlite3 database.
    - [x] Change database name to create new database. Eg: change name to db2.sqlite3.
- [x] **AUTH_PASSWORD_VALIDATORS**
  - [x] Which password validators are applied.
- [x] **STATIC_URL**
  - [x] Talk about later.

models.py

> In docs, arguments given in fields are required arguments. When adding new field, either do null=True or provide some default value(Eg. default="default value").

- [x] **CharField**
  - [x] Must have max_length=120 argument.

- [x] **TextField**
  - [x] blank=False : Makes field as required while taking input.
  - [x] null=True : Makes field nullable in database.
- [x] **DecimalField**
  - [x] decimal_places=2 is required.
  - [x] max_digits=1000 is required.
- [x] **BooleanField**
- [ ] **FileField**
  - [ ] upload_to="images/"
- [ ] **ForeignKey**
  - [ ] First argument must be the other model.
  - [ ] on_delete=models.CASCADE is required
  - [ ] on_delete=models.SET_NULL, null=True can also be set.
- [ ] **ManyToManyField**
  - [ ] through [can have an intermediate model eg TweetLike]
  - [ ] .add
  - [ ] .remove
  - [ ] .set [requires a querySet]
  - [ ] .all
  - [ ] May have to pass other model as a string if it is defined later.
- [ ] **OneToOneField**
  - [ ] Allows you to use . to access related model.
- [ ] To create foreign key to same model, user 'self' as the other model.

Associating Users to Models

- [ ] Eg:

```
# from django.conf import settings

# User = settings.AUTH_USER_MODEL
# from django.contrib.auth import get_user_model
# User = get_user_model()
from django.contrib.auth.models import User

class TweetModel(models.Model):
  content = models.TextField(blank=True, null=True)
  user = models.ForeignKey(User, on_delete=models.CASCADE,
related_name="tweets")
```

- [ ] To access tweets from user_obj, `user_obj.tweetmodel_set.all()` or `user_obj.tweets.all()` (if related_name is provided).
- [ ] Similarly to access TweetLikes by a user, `user_obj.tweetlike_set.all()`

Signals

- [ ] When creation, deletion of one model needs to affect another, we use signals to execute them.

- [ ] Eg

```
from django.db.models.signals import post_save #, pre_save, post_delete ...

...

def function_to_execute(sender, instance, created, *args, **kwargs):
  if created:
    # Executes only if the sender is created for first time.
    Profile.objects.get_or_create(user=instance)
  # Executes each time sender is saved
  # have creation statements here if you have already created users and
save each of them from admin panel. After that you can remove the statement
from here.

post_save.connect(function_to_execute, sender=User) # sender=SenderModel
```

Commands

manage.py

- [x] **runserver**
  - [x] Starts a development server.
  - [x] You can allow the server to keep running and do all changes in another terminal, including migrations.
- [x] **makemigrations** and **migrate**
  - [x] Updates database.
  - [x] Both commands are run together in sequence.
  - [x] Run these upon any change in models.py.
  - [x] To reset database,
    1. Delete all files in migrations folder (except __init__.py)
    2. Delete __pycache__ folder in migrations directory.
    3. Delete db.sqlite3 file.
- [x] **createsuperuser**
  - [x] Allows to create a superuser to login into admin page (urls/admin).
- [x] **startapp appname**
  - [x] Creates new app (component in project).
  - [x] An app does one thing very good.
  - [x] An app does one thing very good.
  - [x] You need to add it in INSTALLED_APPS list.
- [ ] **test**
  - [ ] To run tests in tests.py.
  - [ ] test [app_name] can be used to test a specific app.
- [x] **shell**
  - [x] Allows you to import models and manipulate data to database using the model.
  - [x] Eg. >>> from products.models import Product >>> Product.objects.all() >>> Product.objects.create(name="Watch", price=22)

views.py

Functional Views

- [x] Need to add views in urls.py.
- [x] Takes a request object as argument.
- [x] Conventionally, functions end with _view.
- [x] Add *args, **kwargs also as arguments in function definitions.
- [x] Returns HttpResponse(html_string), render(request, template_name, context_dictionary), JsonResponse(data) or redirect(url)
- [x] Convention is to pass model objects as 'object' in context, and then access the attributes from it.
- [ ] You can use **require_login** decorator.
- [ ] You can add redirect("login?next=/profile/update")
- [x] To use forms, Eg:

```
from .forms import ProductForm
def product_detail_view(request):
  form = ProductForm(request.POST or None)
  if form.is_valid():
    obj = form.save(commit=false)
    # Play with objects
    obj.save()
  context['form'] = form
```

- [x] form.cleaned_data can be used to clean data.
- [x] form.errors can be used to view errors.

request Object

> Request object is also accessible in html templates.

- [x] .user
  - [x] Gives username of user logged in.
  - [x] If no one is logged in, it gives AnonymousUser.
  - [x] .is_authenticated (in template)
- [x] .method
  - [x] can have value 'GET', 'POST' or few other methods.
- [x] .GET dictionary that contains data sent through get request.
- [x] .POST dictionary contains data sent through post request.
- [ ] .is_ajax() : Tells if the request is ajax or not.

ModelName.objects

- [x] .get(id=[number])
  - [ ] This must return exactly one object.
- [x] .create(**dictionary) or .create(attribute1=value1, attribute2=value2 ...)
- [ ] .get_or_create() : Does what it says.

- [ ] .none() : To create an empty querySet.
- [ ] .filter(attr1=value1, attr2=value2)
    - [ ] returns a list of objects.
    - [ ] **.filter(foreign_model__foreign_attr="value")**
    - [ ] **.filter(attr__in=[list, of, values])
    - [ ] To filter with **or** condition:

```
from django.db.models import Q
qs.filter(Q(attr1="value1") | Q(attr2="value2"))
```

    - [ ] Every querySet (returned by filter/all functions) has .count() method.
    - [ ] querySet.orderBy('?') : randomly orders the querySet.
    - [ ] querySet.values_list("list", "of", "values", "to", "return", flat=True)
    - [ ] querySet.aggregate(django.db.models.Sum('field_name'))
    - [ ] querySet.annotate // Look in docs it's better than aggregate.
    - [ ] querySet.distinct() # Useful when using Q.
    - [ ] .filter(attr__iexact="VaLUe") [Ignores exact match]
- [ ] model_object.save() can be used to save the model_objects.
- [ ] model_object.delete()
- [ ] .all().delete(), .filter(user__username="manas").delete()
- [ ] To render error if id is incorrect:

```
def tweet_detail_view(request, tweet_id,  *args, **kwargs):
  try:
      obj = TweetModel.objects.get(id=tweet_id)
  except :
      raise Http404(f"TweetModel with id={tweet_id} not found.")
  return HttpResponse(f"<h1>Testing {tweet_id} {obj.content}</h1>")
```

- [ ] **Sending JSON response**:
    - [ ] Eg of one data:

```
  def tweet_detail_view(response, tweet_id, *args, **kwargs):
    data = {
      'id':tweet_id
    }
    status = 200
    try:
      obj = TweetModel.objects.get(id=tweet_id)
      data['content'] = obj.content
    except:
      data['message'] = "Not found"
      status = 404
    return JsonResponse(data, status=status)
```

    - [ ] Eg of list:

```
def tweet_list_view(response, tweet_id, *args, **kwargs):
  data = {
    'response':[{'id':x.id, 'content':x.content} for x in
TweetModel.objects.all()]
  }
  return JsonResponse(data)
```

urls.py

- [x] Best practice is to create a urls.py for each app and include it in the main project urls.py.
- [x] Copy paste main project urls.py to create apps urls.py.
- [x] Adding urls is given in the starter page.
- [ ] To render a template directly,
    - [ ] TemplateView.as_view(template_name='[template.html]')
- [ ] Add an optional character: re_path(s"profiles?/", some_view) # last s is optional.
- [ ] To add dynamic urls,:

```
# In urls.py
  path('tweets/<int:tweet_id>', tweet_detail_view)
# In views.py
  def tweet_detail_view(request, tweet_id, *args, **kwargs):
    return HttpResponse(f"tweet_id={tweet_id}")
```

- [ ] Instead or path, re_path can be used to add paths with regular expressions.

templates

- [x] Django first looks at the DIRS list for templates, then in installed apps templates directory (in sequence).
- [x] Create a base.html with common headers and other things. Add {% block body %}{% endblock body %} In all other html pages, {% extends 'base.html' %} {% block body %} Then content here will be placed between body block in base.html {% endblock body %}
- [x] To create components separately, create html documents separately and add {% include 'component.html' %}
    - [ ] You can pass variables: {% include 'component.html' with form=form btn_label=btn_label %}
- [x] Context variables can be used inside template with {{ variable }} format.
- [x] To render a list, use for loop:

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

- ☑ To check for conditions, use

```
{% if variable == "some_value" %}
  <h4> variable is 'some value'<h4>
{% elif variable == "some_other_value" %}
  <h4>variable is some other value<h4>
{% endif %}
```

> Refer builtin template tags in docs to know about more tags.

- ☑ comment

```
{% comment "Comment title" %}
<tag>Commented text</tag>
{% endcomment %}
```

- ☑ cycle:

```
{% for item in items %}
<tr class="{% cycle 'row1' 'row2' %}"></tr>
```

- ☑ To render forms, use

```
<form action="[url]" method='POST'>
{% csrf_token %}
{{ form.as_p }}
<input type="submit" >
```

> forms.as_ul is also a valid method. Default action sends request to current url. You can put action='.' to get same effect as default. To perform google search from your website,

```
<form action='http://www.google.com/search' method='GET'>
  <input type='text' name='q' placeholder='Google Search'/>
  <input type='Submit' value='Search'/>
</form>
```

Filters

- ☑ Filters are used in {{ }} this type of syntax.
- ☑ Filters can be used one on top of other. {{ variable|capfirst|upper }}
- ☑ See docs for builtin filters.
- ☑ Custom filters can be created.

- [x] Common ones are:
  - [x] safe : To render text as html (this can be done in view using *mark_safe*).
  - [x] title : Capitalizes first letter of each word.
  - [x] striptags : Removes all html tags.
  - [x] slugify : Replaces spaces with '-'.
  - [x] add:[number] : Adds a number.

forms.py

- [x] Create this file in the app.
- [x] Inbuilt forms Eg.

```
from django import forms
from .models import Product
class ProductForm(forms.ModelForm):
  class Meta:
    model = Product
    fields = [
      'title',
      'description',
      'price'
    ]
```

- [x] Raw django forms. Eg:

```
from django import forms
class RawProductForm(forms.Form):
  title = forms.CharField()
  description = forms.CharField()
  price = forms.DecimalField()
```

- [x] Raw django forms
  - [x] By default, all fields are required, to change required=False.
  - [x] Search for django form fields for more info.
    - [x] Core field arguments in docs tell about defaults.
  - [x] Arguments in a FormField
    - [x] required=False
    - [x] label='New Label'
    - [x] initial=199.99 (in DecimalField)
    - [x] widget=forms.Textarea(attrs={"class":"class1 class2", "id":"some-id", "rows":20, "cols":120})
    - [x] widget=forms.TextInput(attrs={"placeholder":"A placeholder"})

> All widgets can be found in docs.

- [x] Modifying PreBuilt Forms
  - [x] Add the formFields like in raw django form to overwrite them.

- [x] To validate data, create functions with name clean_[field_name]:

```
def clean_title(self, *args, **kwargs):
  title = self.cleaned_data.get('title')
  if 'CFE' not in title:
    raise forms.ValidationError("Title must contain CFE")
  if 'NEWS' not in title:
    raise forms.ValidationError("Title must contain 'NEWS'")
  return title
```

admin.py

- [ ] Register models to be viewed from admin page.
  - [ ] admin.site.register(ModelName)
- [ ] Search by fields and show fields.
  - [ ] Eg:

```
class TweetAdmin(models.ModelAdmin):
  list_display = ['__str__', 'user']
  search_fields = ['content', 'user__username', 'user__email']
  class Meta:
    model = Tweet
admin.site.register(Tweet, TweetAdmin)
```

Creating user

- [ ] get_user_model().objects.create_user(username='something', password='somethingElse')

tests.py

- [ ] It is a good practice to have all declarations above and have all assert statements at the end of test_ function.

```
UserModel = get_user_model()
from rest_framework.test import APIClient

class TweetTestCase(TestCase):
  def setUp(self):
    self.user = UserModel.objects.create_user(username='cfe',
password='somepassword')

  def get_client(self):
    client = APIClient()
    client.login(username='username', password='password')
    return client
```

```
  def test_user_created(self):
    self.assertEqual(self.user.username, "cfe")
```

- [ ] assertEqual
- [ ] assertNotEqual

Creating custom ModelManagers

```
# tweets -> models.py
class TweetQuerySet(models.QuerySet):
  def feed(self, argument):
    return self.filter(any complex query with argument)

class TweetManager(models.Manager):
  def get_queryset(self, *args, **kwargs):
    return TweetQuerySet(self.model, using=self._db)

  def feed(self, argument):
    return self.get_queryset().feed(argument) # this is to avoid .all()
method call in between.

class TweetModel(models.Model):
  ...
  objects = TweetManage()

# to access,
TweetModel.objects.all().feed(user)
# or if feed method is also defined in TweetManager
TweetModel.objects.feed(user)
```

Using Images in database (in development server)

models.py

- [ ] add field models.ImageField(upload_to='images/')

settings.py

- [ ] MEDIA_URL = '/media/'
- [ ] MEDIA_ROOT = BASE_DIR / 'media' # directory where media files will be stored.

urls.py

- [ ] from django.conf import settings
- [ ] from django.conf.urls.static import static
- [ ] urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

Adding MySQL to django

- [ ] To install, refer https://www.youtube.com/watch?v=TG6WAnyeDRw
- [ ] To change password properly, refer https://dailydoseoftech.com/solved-error-1698-28000-access-denied-for-user-rootlocalhost/

Adding PostgreSQL to django

- [ ] Refer : https://djangocentral.com/using-postgresql-with-django/
- [ ] If unable to pip install psycopg2, try pip install psycopg2-binary