12. Building an API

34. Nested Serializers & Serializer Method Fields

when we don't have a direct relationship with the other model and if we need to add that model's object inside our json:

create a new serializer class for the other model and then do the following inside the parent **model's serializer class:**

```
{variable name 1} = serializers.SerializerMethodField()
def get_{any name}(self, obj):
    {varible name 1 } = obj.{any name which should not be a field name of the other model}.all()
    {variable name 2} = {other model's serializer class name}({variable name 1}, many ={True or False})
    return {variable name 2}.data
```

2:47

12. Building an API

34. Nested Serializers & Serializer Method Fields

to access the child models objects:
we should seperate serializer class for that child model and then call that serializer class like this inside parent model's serializer class:

```
{variable name}= {child model serializer class name}(many = False)
```

many is set to false because in this example 1 project only has 1 owner/

12:28

12. Building an API

33. Django REST Framework & Serializers

to access serializer inside view:

```
from .serializers import {serializer class name}
from {app name}.models import {model name}
@api_view(['GET'])
def {any name for function}(request):
        {variable name1}= {model name}.objects.all()
        {variable name2}= {serializer class name}({variable
name1}, many=True)
        return Response({variable name2}.data)
```

9:44

12. Building an API

33. Django REST Framework & Serializers

serilazer class is similar to modelforms

```
from rest_framework import serializers
from {appname}.models import {model name}
class {any name}(serilizers.ModelSerializer):
        class Meta:
                model = {model name}
                fields = '__all__' or [field1, field2 ......]
```

note that we can use serializers for any datatype not only models for example: dictinory,list etc

6:55

12. Building an API

33. Django REST Framework & Serializers

serialization is a process to convert our data into json format in this case convert python object to json format.

3:48

12. Building an API

33. Django REST Framework & Serializers

for function based view we use **@api_view** as decorator

if we use class we use **APIView** as class name

to import api_view decorator:
from rest_framework.decorators import api_view
to import response:
from rest_framework.response import Response

1:20

12. Building an API

33. Django REST Framework & Serializers

to install django rest framewrork:
pip install djangorestframework
inside settings.py in installed apps:
add **'rest_framework'**

16:13

8. User Actions

22. User Edit Profile

in this timestamp **profile =request.user.profile** is to store the instance of which user is logged in into profile variable and that profile variable is passed into **profileform** as instance so that the form is saved into logged in user in the profile

10:02

8. User Actions

22. User Edit Profile

this timestamp is to update the values of a user when that users profile is edited.

18:19

7. Authentication

19. User Registration

this timestamp is to create a custom UserCreationForm: it is similar to creating normal form we should just pass UserCreationForm into our custom form class.

10:13

7. Authentication

19. User Registration

this time stamp shows how to save the values entered in the registration form into user model:
in this **user = form.save(commit =False)** will save a instance of the entered value before saving it into our user model and here that instance is used to manipulate the entered value to our convenience in this case **user.username = user.username.lower()** is used to change the entered username into lowercase and then using **user.save()** we actually store the form into our user model

5:36

7. Authentication

19. User Registration

to use this form in our template store it in a variable and pass inside context dictionary
{variable name} = UserCreationForm()

5:11

7. Authentication

19. User Registration

to import user creation forms:
from django.contrib.auth.forms import UserCreationForm
this will automatically create a registration form which we use it store the values filled in the form to store in our user model(user database)

3:48

7. Authentication

19. User Registration

this timestamp shows how to have different contents on the same page based on a which view is called.

29:34

7. Authentication

18. User Login, Logout and Flash Messages

for flash messages just check out django website it's easy.

23:01

7. Authentication

18. User Login, Logout and Flash Messages

to restrict not logged in users from accessing specific pages

from django.contrib.auth.decorators import login_required

and add the following decorater on whatever page's function

@login_required(login_url = {whatever page we want to redirect the unauthorized users})

21:12

7. Authentication

18. User Login, Logout and Flash Messages

timestamp for redirecting logged in users to a page this will restrict already logged users to access login page.

18:53

7. Authentication

18. User Login, Logout and Flash Messages

time stamp for logout function

17:19

7. Authentication

18. User Login, Logout and Flash Messages

in this timestamp he the changes the login button to logout button in the navbar if the user is logged in

15:13

7. Authentication

18. User Login, Logout and Flash Messages

time stamp for login function

to import login ,authentication and logout:
from django.contrib.auth import login,authenticate

8:47

6. Add More Apps

17. Signals

to connect signal function with the model:
{action}.connect({function name}, sender ={model name})

20:57

6. Add More Apps

17. Signals

when we move the signals into different python file we need to type the following in to the apps.py of that app:

```
def ready(self):
    import {app name}.signals
```

18:16

6. Add More Apps

17. Signals

this timestamp shows creating an action when a user is deleted from profile model it deletes that user from user model

14:43

6. Add More Apps

17. Signals

this timestamp shows creating an action when a user is created it stores those field values of user model into profile model by creating a new object.

7:28

6. Add More Apps

17. Signals

instance argument from receiver will store the returned value from the model.
created argument will store whether the receiver is triggered upon creating a new object to the model this contain boolean value.

this is shown with example in this timestamp.

5:52

6. Add More Apps

17. Signals

to create reciever for model with example of post save action:

```
def {function name}(sender, instance, created, **kwargs):
        {i here we can do any actions we can to do}
        ex: print("profile saved!")
post_save.connect({function name}, sender= {model name})
```

4:01

6. Add More Apps

17. Signals

to import signals for models:
from django.db.models.signals import {actions}
example for actions: **post_save**.

0:06

6. Add More Apps

17. Signals

explained what are signals

37:43

6. Add More Apps

16. Add & Render Profiles

to pluralize a word:
example: vote{{project.vote_total | pluralize: "s"}}

34:40

6. Add More Apps

16. Add & Render Profiles

for loop to parse through child model's objects:
{% for i in {model name}.{child model name}_set.all %}

24:03

6. Add More Apps

16. Add & Render Profiles

for if condition in html:
{% if {condition}%}

7:49

6. Add More Apps

16. Add & Render Profiles

to limit the character to show in a page
{{{model name}.{field name}|slice: "150" }}
note: 150 is number of charaters that should be displayed

14:42

6. Add More Apps

15. Users App

To import user model from django:
from django.contrib.auth.models import User

for one to one relationship between the models:
{varibale name} = models.OneToOneField({other model name})

31:22

5. Static Files & Theme Installation

13. Static Files

To let the django know how serve the new static folder do the following:
pip install whitenoise
inside settings.py:
inside middleware
add **'whitenoise.middleware.WhiteNoiseMiddleware'**

by doing this we can see our static files applied to our website but the user uploaded files will not be applied in our website (whitenoise will not do that)we should use some other package which he will teach later in the course.

25:48

5. Static Files & Theme Installation

13. Static Files

when we setup our website to a server we should do the following for our static files:
inside settings.py:

set **DEBUG =FALSE**
ALLOWED_HOSTS = [' {url you are working on} ']
next to **MEDIA_ROOT** add **STATIC_ROOT =os.path.join(BASE_DIR, ' {name of new static folder} ')**

to create new static folder run **python manage.py collectstatic** in the terminal.

inside urls.py in the base app:
urlpatterns += static(settings.STATIC_URL, document_root= settings.STATIC_ROOT)

22:29

5. Static Files & Theme Installation

13. Static Files

to save the uploaded image from the forms:
inside the template: **<form action="" method="POST" enctype ="multipart/form-data'>**
inside the views : **{variable name} = {model form name}(request.POST, request.FILES)**

refer chapter: 11 for more.

20:16

5. Static Files & Theme Installation

13. Static Files

adding MEDIA_URL and MEDIA_ROOT to urls.py of base app:

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

17:45

5. Static Files & Theme Installation

13. Static Files

to make our image work in the template we should do the following next to STATIC_URL in settings.py:

MEDIA_URL ='/{image folder name}/'

16:59

5. Static Files & Theme Installation

13. Static Files

to add image from the model to the template:
<img src="{{ {model name}.{image field name}.url }}">

15:17

5. Static Files & Theme Installation

13. Static Files

to let the server know where should the uploaded image should store we should do the following in the settings.py next to **STATICFILES_DIRS**

MEDIA_ROOT = os.path.join(BASE_DIR, 'static/{images folder}')

11:24

5. Static Files & Theme Installation

13. Static Files

for imagefield to work we should install pillow library.
python -m pip install pillow

7:37

5. Static Files & Theme Installation

13. Static Files

we should add **{% load static%}** to every template where we use static folder.

4:55

5. Static Files & Theme Installation

13. Static Files

load {%load static%} into our main template
in the stylesheet section of our main template do the following:
href='{% static "{styles folder}/{css file name}"%}'
note: remember in href if you gave single quotes outside give double quotes to the path(next to static)

3:46

5. Static Files & Theme Installation

13. Static Files

inside settings.py next to STATIC_URL write the following.
STATICFILES_DIRS=[BASE_DIR/'static']. inorder let our server know about the static file in the root directory

1:30

5. Static Files & Theme Installation

13. Static Files

in the root directory add a folder named as static.
inside the static folder you can create multiple folders for images, styles(for css), javascript etc..

20:08

4. Create Update Delete (CRUD)

12. Create Read Update Delete (CRUD)

to delete an object to our model from the form:

```
def {function name}(request):
    {variable name} = {model name}.objects.get({attribute in the object} ={variable in order to match the id of the object})
    if requests.method == 'POST':
```

```
        {variable name}.delete()
         return redirect('{whatever page we want}')
    context ={ '{whatever name}': {variable name}}
    return render(request, "{template name}" , context)
```

10:01

4. Create Update Delete (CRUD)

12. Create Read Update Delete (CRUD)

url path for update:

```
path('{any name}/<str:{variable in order to match the id of the object}>' , views.{functions name to update} , name={any name you like to call the url path})
```

9:11

4. Create Update Delete (CRUD)

12. Create Read Update Delete (CRUD)

to update an object to our model from the form:

```
def {function name}(request, {variable in order to match the id of the object}):
    {variable name1} = {model name}.objects.get({attribute in the object} ={variable in order  to match the id of the object})
   {variable name2} = {model form name}( instance ={variable1} )
    if requests.method == 'POST':
      {variable name2} = {model form name}(request.POST, instance = {variable1})
       if {variable name2}.is_valid():
          {variable name2}.save()
          return redirect('{whatever page we want}')
    context ={ '{whatever name}': {variable name}}
```

5:05

4. Create Update Delete (CRUD)

12. Create Read Update Delete (CRUD)

to create a new object to our model from the form:

```
def {function name}(request):
    {variable name} = {model form name}()
    if requests.method == 'POST':
      {variable name} = {model form
name}(request.POST)
       if {variable name}.is_valid():
          form.save()
          return redirect('{whatever page we want}')
    context ={ '{whatever name}': {variable name}}
    return render(request, "{template name}" , context)
```

to import redirect and render:

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

18:25

3. Building our Database

10. Database Queries

the below function is to add the child model:

```
def {function name}(request, pk):
    {variable name1} = { model name }.objects.get({any
field in the model} = pk)
    {variable name 2} = {variable name1}.{child model
name}.all()
    context = { ' {any name1}' : {variable name1} , '{any
name2}' : {variable name2} }
    return render(request, '{ template name }', context)
```

15:18

3. Building our Database

10. Database Queries

in order to make our models reflected in the templates we should import it into our views:

```
from .models import {model name}
```

the below functions is to display all the fields in the model :

```
def {function name}(request):
    {variable name} = { model name }.objects.all()
    context = { ' {any name}' : {variable name} }
    return render(request, '{ template name }', context)
```

and now we can use our model using the variable inside the template.

8:30

4. Create Update Delete (CRUD)

11. Model Forms

to add a form in a template :

```
<form method ="POST">
{% csrf_token%}
{{name inside the context for the forms}} or {{name inside the context for the forms.as_p}}
<input type = "submit>
</form>
```

.as_p is to wrap every field in the model to wrap inside a paragraph which makes it look neat and clean

7:55

4. Create Update Delete (CRUD)

11. Model Forms

to import our model form inside the views:

```
from .{name of the python file you have created for model forms} import{name of the model form}
def {function name}(request):
    {variable name} = {model form class name}()
    context ={ ' {name you want}' : {variable name used to call the model form class} }
    return render(request, " {folder where the template stored}/ { template name } ", context )
```

6:06

4. Create Update Delete (CRUD)

11. Model Forms

to create a model form:

```
from django.forms import ModelForm
from .models import {name of the table for which we need to create a form}

class {model form name}(ModelForm):
        class Meta:
                model ={name of the table for which we need to create a form}
                fields = '__all__' or [ ' field1 ', ' field2 ' etc ]
                labels = {
                        'field1': '{whatever name we want to display in the form before textbox}' }
```

12:42

3. Building our Database

10. Database Queries

to see the all the objects of the child model:

```
{variable} = {parent model}.objects.get({attribute})="{value}")
{variable}.{child model name}_set.all()
```

note that the child model name should be always in smallcase while giving inside the above code even if the child model name you have given have caps in it.

12:44

3. Building our Database

9. Database Relationships

to create a many to many relationship between the models:

```
models.ManytoManyField({modelname})
```

in both the case "one to many" or "many to many" if the other model is below the model where you are typing then add a single quote to the model name. like below
models.ManytoManyField('{modelname}')

11:05

3. Building our Database

9. Database Relationships

to make a one to many relationship between the models:
{variable name } = models.ForeignKey({the other model name})

18:17

3. Building our Database

8. Models & Admin Panel

inorder to choose what should we show after we add values to the model we should do the following:
add a string method and choose what variable should be shown from the model
ex:

```
def __str__(self)
    return self.{variable name}
```

17:05

3. Building our Database

8. Models & Admin Panel

in order to add our model to the admin panel:
go to the admin.py of our app and import our class from models an register it with the admin

```
from .models import {class name}
admin.site.register({class name})
```

15:47

3. Building our Database

8. Models & Admin Panel

python manage.py migrate
this will execute what the migrations told inside the database

15:01

3. Building our Database

8. Models & Admin Panel

to make migration of our model:
python manage.py makemigrations
this will create migrations

14:24

3. Building our Database

8. Models & Admin Panel

models is represented as a class in django.
shown in this timestamp of the video.

4:19

3. Building our Database

8. Models & Admin Panel

to create admin:
python manage.py createsuperuser

2:32

3. Building our Database

8. Models & Admin Panel

to turn on default modules in the django :
python manage.py migrate
for example to access admin panel.

19:39

2. The Basics

7. Rendering Data to Templates

accessing urls using url names
{% url '{url name}' {dynamic value}%}

12:10

2. The Basics

7. Rendering Data to Templates

for loop in template:
ex:
{% for project in projects %}
content
{% endfor %}
where projects is a dictionary

7:25

2. The Basics

7. Rendering Data to Templates

if statement inside template:
{ % if x >10% }
content
{% endif %}
same for elif and else

5:59

2. The Basics

7. Rendering Data to Templates

inorder to send a variable to the template add the variable name inside {{ }} in the template.
for ex: **{{ message}}**
and in the function of views.py we should initialize the variable here and send it with a name inside render function
for ex:
def projects(request):

```
    page = 'projects'
    return render(request, 'projects/projects.html,{ 'page' : page})
```

we can also wrap multiple variables inside a single dictionary and pass it inside the render function

16:53

2. The Basics

6. Templates & Template Inheritance

in order to organize the templates based on the apps create templates folder inside the specific app and create another folder inside the template with the same name as our app. then add our html files inside this folder.
in order to make these newly added templates to work: change the {template}.html inside the function of our views.py to **{appname}/{template}.html**

7:25

2. The Basics

6. Templates & Template Inheritance

Extending a Template:
create a main.html template and write html boiler plate inside the body of the main template
add the following code :

```
{% include '{template name}.html' %}
{% block content %}
{% endblock content%}
```

add **{% extends 'main.html'%}** at the top of every child template.
we should add block and endblock content inside the child template and inbetween those content of the child template should be present.
so between block and end block content all our child templates sits.

6:25

2. The Basics

6. Templates & Template Inheritance

Template inheritance:
{% include '{template name}.html' %}

2:33

2. The Basics

6. Templates & Template Inheritance

to let the django know where our templates are we should do the following:
import os inside the settings.py
and the following code inside the TEMPLATES variable
'DIRS' : [os.path.join(BASE_DIR, ' {template folder name} ']

0:52

2. The Basics

6. Templates & Template Inheritance

create a template folder inside the outermost folder
and create html pages inside the templates folder

10:15

2. The Basics

5. Views & URL's

we are also changing the urlpatterns from rootapp to the newapp which will easy for us to use and understand as we keep increasing the number of apps

create urls.py inside the new app and include the following things in it:
from django.urls import path # for adding paths

from . import views # to access the functions inside the views (. represents the current app)
urlpatterns =[path('{url extension}/<str:{variablename}>',views.{function name} , name= "{what we like to call this path}" etc]

9:27

2. The Basics

5. Views & URL's

all the functions are written inside the views file

12:37

2. The Basics

5. Views & URL's

if we have a different app and have to include it to our main app we should do the following:
from django.urls import include

path({url route}, include('{appname}.urls')) add this path to the url patters of the main app

8:31

2. The Basics

5. Views & URL's

dynamic route url:
path('{url extension}/<str:{variablename}>',{function name} , name= "{what we like to call this path}"
variable name should be passed as a parameter to function where we can use it inside the functions

4:15

2. The Basics

5. Views & URL's

adding url paths:

urlpatterns = [path('{url extension}/',{function name} , name= "{what we like to call this path}"]

16:34

1. Introduction

4. Installation & Setup

add newly created app inside installed apps in settings.py if we have to give the class inside the app then ({project name}.apps.{class name})

8:07

1. Introduction

4. Installation & Setup

Commands needed for Django:
virtualenv commands:
virtualenv {env name}
source {env name}/bin/activate

django commands:
pip install django
django-admin startproject {project name}
cd {project name}
python manage.py runserver
python manage.py startapp projects