BACKEND

Command Prompt →

① if want to know subfolders in directory
'dir'

.android ⟶ . signifies that folder is hidden

② changing directory to go back
cd name → name of folder
cd ..

③ clear
cls

④ printing current directory → cd

Python

① Number < int / float

② # comment

③ print (my_string[2:]) → grab all letters starting from index 2
my_string[:3] → prints upto 3rd index excluding 3rd index
[2:5] → all letters from 2 to 5 excluding 5
[::2] → step size 2 prints every 2nd letter
[::1] → step size

④ string is immutable
str[0] = 'x' ⟶ X

⑤ my_string.lower()

⑥ my_string.split() ⟶ ['Hello', 'World']
split('e') → ['H', 'llo World']

- Print formating →

x = "Item One": {} Item two : {} ". format("dog", "cat")

O/P Item One : Dog Item two : Cat

- List →

mylist = [1,2,3] ⟶ mylist[0] ⟶ 1
[-1] ⟶ 3

mylist = ['Hey', 1, 2, 3, 1.2, True]

print (len(mylist) ⟶ length

Add an item → mylist.append ("New") ↳ add to end of list

mylist = [1,2,3]
listtwo = [4,5,6]

mylist.extend(listtwo) ⟶ 1,2,3,4,5,6

Remove

item = mylist.pop() ⟶ 1 2 3
print (item) ⟶ 1
print (mylist) ⟶ 2 3
item = mylist.pop(1) ⟶ 3

Reverse →

mylist.sreverse () ⟶ [1,2,3]
print (mylist) ⟶ [3,2,1]

Sort

mylist.sort()

Matrix →

matrix = [[1,2,3], [4,5,6], [7,8,9]]

[1,4,7] ← first_col = [row[0] for row in matrix]

→ Dictio
my
pr
pr
→ Boolea
→ Tupl
→ Set
→ for
to tuple

→ Dictionary

my = {"key1": "value", "key2": "value2"}

print (my['key2'] → value 2

print (my)

→ Boolean

True 1 False 0

→ Tuples → lists but immutable

tup t = (1, 2, 3) t = (1, 1.2, "Hey")

print (t[0])

→ Set → unordered collection

x = set()

x.add(1)

x.add(2)

x.add(0.1)

x.add(1)

x.add(1)

print (x) → {1, 2, 0.1}

↳ any order

↳ no duplicate allowed

→ if else →

→ and or used

→ {} x → intendation

→ if 1<2:

print ('yes!')

elif (2<3):

print ('NO!')

else

print ('ya')

→ for

for item in seq

i array or anything

print (item)

to tuple → for (tup1, tup2) in mypair

print (tup1)

print (tup2)

→ while →

* comment → """ """ ''' '''

```
while i<5:
    print ("i : {}".format(i))
    i= i+1
```

for each num do num*2 and give in out

★ out=[num**2 for num in seq]

→ for item in range(10) → 0-9
    print (item)

→ Function →

syn → def my-func (param1 = 'default'):
        print (param1) → default

    my-func()

eg → def add (num1, num2)
        return num1 + num2

    result = add (1, 2)
    print ('result')

★ check type
print (type(result))

→ filter → filters needed input

```
def even (num)
    return num%2==0
evens = filter (even, mylist)
print (list(evens))
```

→ SCOPE →

Local
Enclosing fn ~~goto~~ local
Global
Built in

```
def func()
    global(x) global x → effects to global x
    x=1000
```

OOP →

```
class Dog():
    def __init__(self):          # refers to itself necessary
        print("Animal created")

    def eat(self):
        print("Eat")

class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog created")
```

Initialize ① `def __init__(self):`

② `def __init__(self, breed):`
`self.breed = breed`

```
mydog = Dog(breed = "Lab")
print(mydog.breed)        → Lab
```

```
class Dog():
    species = mammel
    def __init__(self, breed, name):
        self.name = name
        self.breed = name
    def area(self):          # method of class
        return self.name * self.name * Dog.species
```

```
mydog = Dog(name = "Jash", breed = "Lab")
mydog.name
mydog.breed
mydog.species
```

Calling another class →

```
mya = Animal()        → Animal created
mya.eat()             → Eat
mydog = Dog()
dog.eat()
```

→ parent init
Animal created
→ class init
Dog created
Eat → inherited methods

Printing class →

```
class Book:
    page = 100
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return "Title: {}, Author: {}".format(self.title, self.author)

    def __len__(self):
        return self.book.page

b = Book("Python", 99)
print(b)
print(len(b))
```

Error & Exception →

```
try:
    f = open('simple.text', 'w')
    f.write("Test write to")
except IOError:
    print("could not open file")
else:
    print("success")
    f.close()
finally:
    print("I always work")
```

open fn
open('file_name', 'r')
open('file_name', 'w')

checks for input output error
don't need to specify

else → if error

finally: → will work even if exception called

Regular Expressions →

```
import re ——→ importing regular exp.

patterns = ['term1', 'term2']
text = 'this is a string with term1 but not other'
for pattern in patterns:
    print ("I am searching for " + pattern")
    if re.search(pattern, text):
        print("match")
    else no mat
```

match object ← re.search(pattern, text) ← attribute; key to search; where to search

---

→ read

→ write

```
text = "gt term1"
match = re.search('term1', text)
print (match.start())
```

→ 3

```
email = username@gmail.com
print (re.split('@', email)
```

'@' ↓ where to split; ↳ what to split

['match', 'match'] ← `print (re.findall ('match', 'test phrase match to match'))`

Django

free & opensource web framework

used because packages get updated

virtual environment → allows to have virtual installation of python and other packages on your computer

to make virtual env with conda by
conda create --name myEnv Django
(name) (Package)

activate by
activate myEnv

deactivate by
deactivate Env

How to start →

① make a directory for your project
mkdir file_name → make a new folder
cd file_name → go to folder

② Activate your env
activate myDjangoEnv
(name)

③ create a project
django-admin startproject name

- you will get various files

① __init__.py → blank, name tells python that it can be used as package

② settings.py → to store your project settings

③ urls.py → store all URL patterns for your project

④ wsgi.py → Python script that acts as the web server gateway interface. help to deploy web app to production

⑤ manage.py → associates with many commands as we build our web apps

→ Managing & using Manage.py →

① python manage.py runserver
copy the last url on chrome — will lead to first project

② Migration → reversible, allows to move database from one design to another.

→ Terminologies →

① Django Project → collectn of applicatn & config that when combined together will make up the full web application.

② Django Application → created to perform a particular functionality for your entire web application.
Eg → registration app, polling app etc
They are pluggable. others can use apps from your project & vice versa.

→ To create app →

① creating a app folder
python manage.py startapp name

② contains file →
① __init__.py → ''
② admin.py → can register your model here which django will then use with ~~other~~ Django's admin interface
③ apps.py → place applicatn specific config
④ models.py → store applicatn's data model
⑤ test.py → store test fn to test your code
⑥ Migration folder → stores database specific info as it relates to the model

③ Need to tell Django that we created an application

→ setting.py > Installed_APPS

Search in file

→ Add a string 'file_name' to the installed_apps array

name of app file

④ Creating views →

Eg → sending Hello World →

first_app > views.py →

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello World")
```

1st view ← variable

HttpResponse object

⑤ To see view → linking views to urls.py

→ first_project > urls.py

→ search for urlpattern = [

Add → from APP_name import views

urlpattern=[

```
    path('', views.index, name='index'),
    path('admin/', admin.site.urls),
]
```

If run & web see Hello World

So created application, create views & map it urls file to work through url.

→ Mapping Urls → Another way [include()]

impot from django.conf.urls import include() → allows to look for a match for regular expressions and link back to our application's own urls.py file.

We have manually add in urls.py file

Method → add in urls.py of project

```
from django.conf.urls import include
url patterns = [
    path('app_name/', include('app_name.urls'))
]
```

this will be the extension name in the url

steps

① Go to urls.py of project
add ie fso

```
>{
```

② Make a new file in app folder as urls.py

add

```
from django.conf.urls import url
from first_app import views
                    ↳ name of app
urlpatterns = [
    path('', views.index, name='index')
```

Templates

contains static parts of Html page

template tags → injects dynamic content that your Django app views will produce, effecting the final Html

template variables → inject content into the Html directly from Django

Creating template →

① under project (first_project) create a new folder named template.

② In the subfolder of first_project > settings.py
(i) add below BASE_DIR → to make it run on any operating system

```
TEMPLATE_DIR = os.path.join(BASE_DIR, "template")
```

(ii) In TEMPLATES [

'DIRS' : [ ],
↓ Add
TEMPLATE_DIRS

Adding a template →

① In template folder, add file Index.html

② Adding a template tag in body →

{{ insert_me }}
↓
tag variablename

③ Connecting variable (insert_me) to our project and application

in first_project > views.py >

```
def index(request):
    my_dict = {'insert_me': "Hello I am"}
    return render(request, 'index.html', context=my_dict)
```

key value pair with key as the tag variable name

html file to add

what to link

④ To make modular →

in templates folder add a folder named on application and put index.html in it

change * to first_app/index.html

name of app

Static Files

{{ }} Simple text injections

% % complex injections and logics

① Creating Folder

→ in project folder add a new folder named static and create a

→ folder in it named images

→ first_project > static > images

→ add image required in the image folder

② first_projects > setting.py

(i) Below base_DIR & Template_DIR add →

STATIC_DIR = os.path.join(BASE_DIR, "static")

joins base dir to static

(ii) at the bottom of page.

below STATIC_URL = '/static/'

↳ through this url we can access

add → STATICFILES_DIRS = [

STATIC_DIR,

refers to loc on comp where static files are stored

]

★ Put static & template in app folder.

③ For loading image through html

→ <!DOCTYPE.html>

{% load static %}

→ <body>

<img src = "{% static "first_app/images/image.jpg" %}

alt = "Uh not loaded" >

④ For adding CSS

① make a folder in static name css write full name.css

② link html with CSS as

<link rel = "stylesheet" href = "{% static "css/name.css" %}

create a table

Models

used to incorporate a database into django/projec

① go to model.py in project folder

② → Code

```
from django.db import models

class Topic(models.Model):
    top_name = models.CharField(max_length=264, unique=True)

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

built in class → inherited parent class

① Table

each attribute refers to a field or a column name

CharField → field type

unique=True → no duplicates allowed

return self.topic_name → if print topic → returns name

② Table

```
class Webpage(models.Model):
    topic = models.ForeignKey(Topic)
    name = models.CharField(max_length=264, unique=True)
    url = models.URLField(unique=True)

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

columns

* Similarly for date → DateField etc

Primary key → unique identifier for each row in a table.

Foreign key → denotes that column connects with a primary key of another table

Migrations

③ Apply following command →

→ python manage.py migrate → migrate database corresponding to models

→ " " makemigrations first_app → register changes to app

→ " " migrate → migrate database one more time

→ " " shell → Changes add through cmd

1st Method to add contents

Method ①

→ from first_app.models import Topic

→ t = Topic(top_name = "Hy")

To save a entry → t.save()

to print → print(Topic.objects.all())

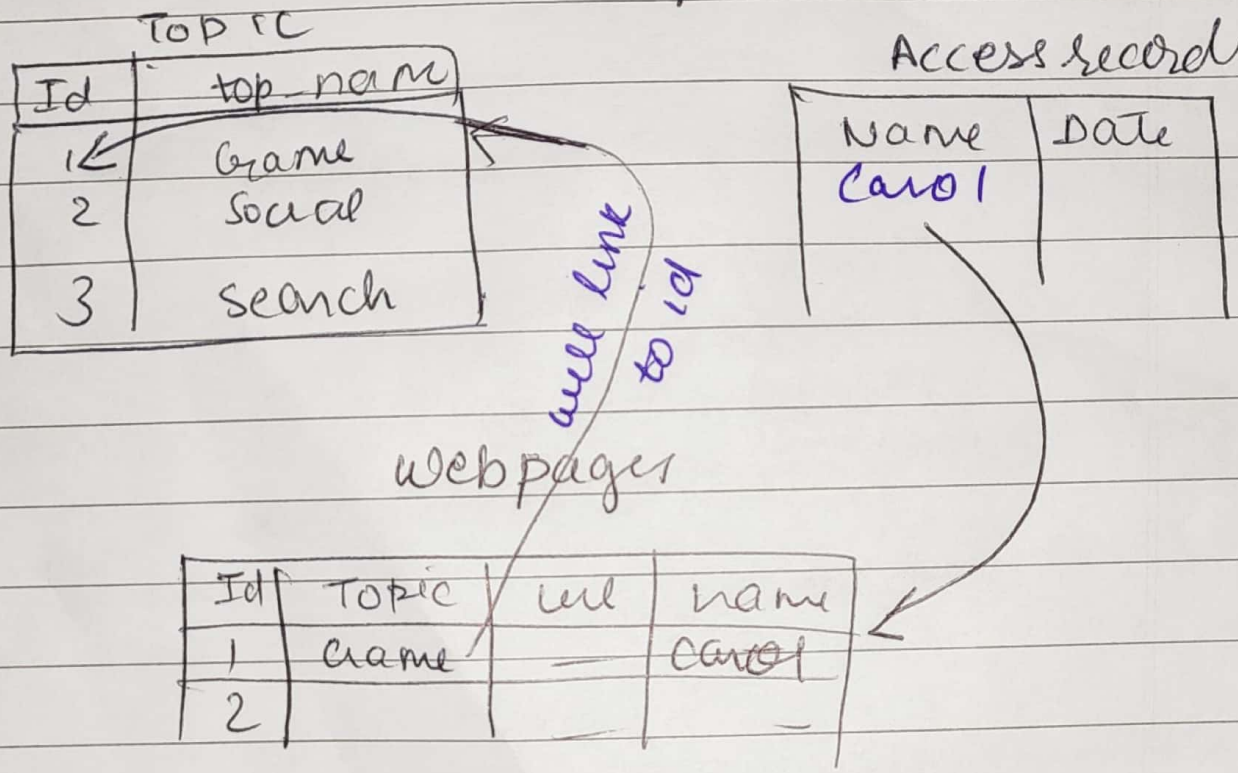
* Foreign key links with the id of parameter

We have 3 classes (3 tables)

1) Topic → top_name

2) Webpag → Topic (foreign keys), url, name

3) Accessrecord → name (fore), date



AccessRecords

Name = models.ForeignKey(Webpage)

↳ means name links to the webpage table

In name we can input parameters of webpage table and get info

against sound design principles, so we instead create a relationship (the 'R' in ORM) between the `yourapp_Course` table and the `yourapp_UserProfile` table.

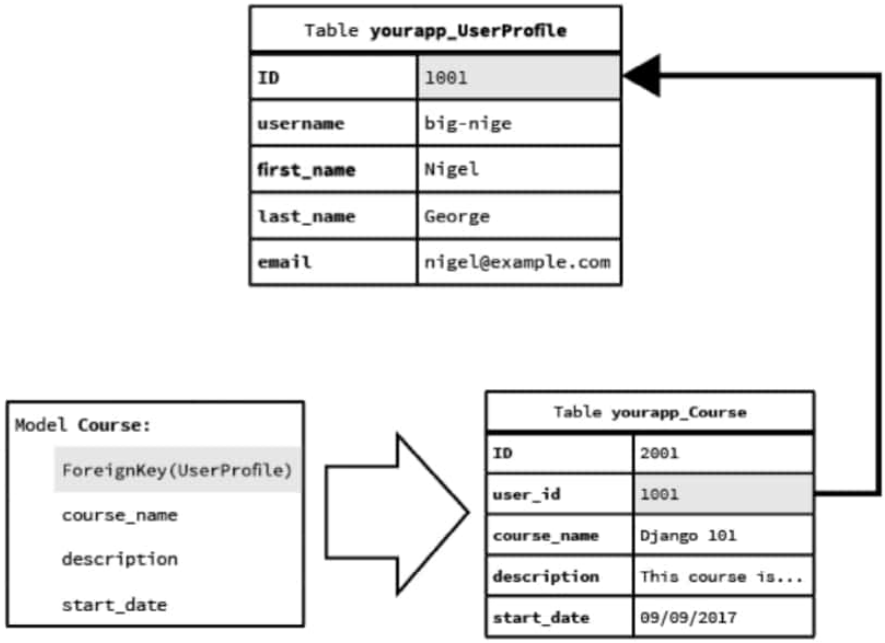


Figure 4-3: Foreign key links in Django models create relationships between tables.

This relationship is created by linking the models with a foreign key—i.e., the `user_id` field in the `yourapp_Course` table is a key field linked to the `id` field in the foreign table `yourapp_UserProfile`.

to quit → quit()

② Method to add contents | Admin | Method ②

① go to first-app > admin.py

```
from django.contrib import admin
from first_app.models import Topic, Webpage, AccessRecord

admin.site.register(AccessRecord)
admin.site.register(Topic)
admin.site.register(Webpage)
```

class name

} registered classes

② In cmd → → to use database and admin, we create a superuser

```
python manage.py createsuperuser
```

→ returns password + username to login to admin

will ask for username password gmail
remember to login to admin

```
python manage.py runserver
```

③ /admin in site leads to → login
leads to a django admin site.

→ Faker library to create script

① Install →
cmd → pip install faker → in first project (code all)

② create populate_first_app.py

③ cmd → "..." .py

④ cmd runserver

Models Template Views

① Connecting views to database

steps →

① Go to views.py.

② Add → from first_app.models import Topic, Webpage, AccessRecord
models.py classes
connecting views to database
def index(request): database info → views → index page as table
webpages_list = AccessRecord.objects.order_by('date') ← method to order by date
date_dict = {'access_record': webpages_list}
return render(request, 'first_app/index.html', context = date_dict)

③ index.html → code
style as per need → css
↓
imp →
→ <!DOCTYPE
→ {% load static %}
→ <link ---- href = "{% static "css/style.css" %}"
→ {% if access_record %} → to check if there are access records
→ {% else %} → else tag
→ {% endif %} → necessary to end if
→ {% for acc in access_records %} → for loop
{% endfor %} → nece end tag

④ runserver

Django forms

Making Django Project from scratch

Level II till module

Creating Project Folder, APP folder

(1) cmd
- → activate myDjangoEnv
- → django-admin startproject Protwo (names → Making Project)
- → Protwo > python manage.py startapp appTwo (Making app)

(2) Creating Template →
Right click appTwo folder > add folder Template
create → Template > appTwo > index.html, .html } needed html files

(3) Setting.Py changes →
① # See template section 7 creating template 6 page before
- → TEMPLATE_DIR (Linking & Registering template to Project)
- → 'DIRS' = [ ]

② INSTALLED APPS = [
'appTwo'] (APPName → Registering & Linking app to project)

(4) In Models.py # Refer to Models section 7 code 3 page before
- → Create required class

(5) In views.py
code →

from Apptwo (appname) import render, HttpResponse, class_name (class name)
(Import)

- def index (request) (main webpage display index.html)
  return render (request, Apptwo/index.html)

(Methods)

can similarly insert other methods linked to respective html files

⑥ urls.py →

① create files urls.py in Apptwo folder

② In urls.py code →

import { from django.urls import url
from apptwo import views

urlpatterns [

path('', views.users, name="users")

App name

main page method called from views.py

]

③ Go to Project > urls.py

import { from django.contrib import admin
from django.urls import path, include
from apptwo import views

urlpatterns [

mainpage ← path('', views.index, name="index")
admin page ← path('admin/', admin.site.urls)
users page ← path('users/', include('apptwo.urls'))

]

⑦ Migration → (of Models to Database)

Protwo > python manage.py migrate → (to register all the changes to the app)
python manage.py makemigrations Apptwo
python manage.py migrate

⑧ In admin.py →

Check admin section

⑨ ★ Regularly check for succesful changes by python manage.py runserver

⑩ create index.html

⑪ Populate → create file in project
Refer laptop ← → code
→ cmd protwo > python file-name

⑫ create user table in user.html →

⑬ Run server mainpage → index.html
/admin → admin page
/users → user page

If want to create user table at user page

Scanned with CamScanner

Django Forms

HTTP → enable communecoon between a client and a server
request / respond

For ← Request/Response
METHODS

Get
request data from a resource

POST
submits data to be processed to a resource

Creating basic form →

① Create project basicform, app basicapp

② Create template folder / basicapp folder / index.html / form.html

③ do changes in settings.py — Template_DIR
DIR = [ ]
INSTALLED APPS

④ create file forms.py in basicapps.

code →

```
from django import forms
~~from django.core import validators~~

class FormName(forms.Form):
    name = forms.CharField()          # name char
    email = forms.EmailField()        # email field
    ~~verify email = forms.EmailField(label = 'Enter your mail again')~~
    text = forms.CharField(widget = forms.Textarea)   # same as Html input = textarea; text area
```

CLEAN DATA → All validated checked data

⑤ Go to views.py →

- from django.shortcuts import render
- from . import forms ← Or → from forms import FormName

forms.py ↑ class name ↑

import from same directory as current.py file

```
def index(request):
    return render(request, 'basicapp/index.html')
```

will be called from urls.py

```
def form_name_view(request):
    form = forms.FormName()

    if request.method == 'POST':
        form = forms.FormName(request.POST)

        if form.is_valid():
            print("Name"+form.cleaned_data['name'])
            print("email"+form.cleaned_data['email'])
            print("text"+form.cleaned_data['text'])

    return render(request, 'basicapp/form_page.html', {'form':form})
```

If we hit submit with method POST

Passing the POST request

If we get POST back, check if data is valid and so grab data

Method to check if each input field has correct datatypes

dictionary to access valid data

input field as key

⑥ Go to urls.py of project

```
from basicapp import views

urlpatterns [
    path('admin/', admin.site.urls),
    path('formpage/', views.form_name_view, name='form_name'),
]
```

⑦ Go to form_page.html

```
<div>
<form method = "POST">
{{ form.as_p }}
{% csrf_token %}
<input type="submit" value="Submit">
</form>
</div>
```

key from context dictionary

data to be submitted to be processed

form to be inside para tag

MOST

will accept info and secure it and check if inputs on form submitted is not going on some other websites and saves from getting fake data

⑧ runserver

Form Validation

① Creating botcatcher → Hidden field remains in Html but hidden from user. This field is used to catch bots. can be seen in INSPECT.

In formName in forms.py add →

*• botcatcher = forms.CharField(required=False, widget=forms.HiddenInput)

creating own Validator ↳

OWN VALIDATOR

①• def clean_botcatcher(self):
(keyword to check for item; ↗ checked item)

botcatcher = self.cleaned_data['botcatcher']
(dictionary attribute; valid grab data what botcatcher returns in forms html)

Means a robot scraped the page and filled form and html field ← if length(botcatcher) > 0:

raise forms.ValidationError("caught..")
(raise ↓ to raise errors)

* If bot is catched, i.e. if its changes something in html, will show on page [(Hidden field botcatcher) caught!

check by ←

Instead ① → ② built in method.

BUILT-IN

from django.core import validators

botcatcher = form.CharField(_ _ _ ,
validators=[validators.MaxLengthValidator(0)])

(validators → method; MaxLengthValidator → built-in validator; checks that it should be 0)

OWN VALIDATOR

More validators creating

• def check_z(value): → must to identify as validator if making own validator
(creating a function to check for z)

if value[0].lower() != 'z':

raise forms.ValidationError("Name needs to start with z")
(error)

add validator → name = forms.CharField(validator=[check_z])

→ verify email double check → Method →

add a variable in class ← verify_email = forms.EmailField(label = "Enter email again")

def clean(self): ← keyword to check entire form at once

dictionary ← all_clean_data = super().clean() → grab all clean data at once

email = all_clean_data['email']

vmail = all_clean_data['verify_email']

check for match ← if (vmail != email)

raise forms.ValidationError('Make sure emails match')

Connecting forms to Model — accepting form input and passing it to a model as table

① FrontEnd

edit index and user + insert form tag in users.html

② Backend

i) forms.py in app Folder

from django import forms

from app.models import User → class

class NewUserForm(forms.ModelForm) → to connect to model class to create a form from a preexisting model

If you want validate write form fields here ← class Meta():

model = User → link to class

field = '__all__' → for info see page

provides info connecting the model to the form

3) views.py ★ a form would directly be generated from User class / model

import Render

from appTwo.forms import NewUserForm

only 2 needed

def index(__):

def users(request):

form = NewUserForm()

if (request.method == "POST")

form = NewUserForm(request.POST)

if form.is_valid():

saved in admin database ← Save to form ← form.save(commit=True) to database

~~four~~ Fields

① field = "__all__"

grabs all fields from model

② exclude = ["name", "email"]
field 1 field 2

exclude those fields and include everything else.

③ field = ("field1", "field2")
→ list of included fields

include these fields.

return index(request) → will take to homepage

else:
print('Error')

return render(request, 'Apptwo/users.html', {'form':form})

→ Relative URLS with Templates →

i.e. hardcoded urls ⟶ href with URL template

steps →

① create project learning_templates, app basicapp, templates and templates/ basic_app / index.html
folder folder other.html
base.html
~~other~~ relative_url_template.html

② change settings

③ views.py

def index ——
return ——

def other ——
return ——

def relative ——
return ——

④ project > urls.py

path('' ——
path('admin/ ——
path('basicapp/', include('basicapp.urls'))

⑤ basic app > urls.py → for template tagging

app_name = 'basicapp'

urlpatterns = [
path('relative/', view ——),
path('other/' —— name='other'),
]

⑥ in relative_url_template.html

add → Other Page

same name

* no space or django will look for basicapp

Now it is relative to actual application only

For admin link → already in installed apps

 admin

For index link →

—— "{% url 'index' %} " ——

Template Inheritance

→ allows to create base template we can inherit for
→ saves repeatative work
→ template extending : extending base.html to other html files

Steps →

① In base.html

<links to JS, CSS, ——>
<html navbar>
<body>

{% block body_block %}

{% endblock %}

Everything outside this will be extended to all pages

In other.html file

<!DOCTYPE html> → MUST

{% extends "basicapp/base.html" %}

{% block body_block %}

{% endblock %}

everything inside will show

Template Filters

General form of template filter →

{{ value | filter : "parameter" }}

* search django + templates for documentation

I) Built in template filter →

① Adding a tag in index →

in views.py

```
def index ——
    context_dict = {'text': 'hello', 'number': 100}
    return render(——, context_dict)
```

② adding tag in index.html & adding filter to tag

{{ text|upper }} ⟶ hello ⟶ HELLO

{{ number|add:"99" }} ⟶ 100 ⟶ 199

(filter, value, parameter)

③ For creating own filters →

① → create template tags folder > __init__.py file → to act as model

my_extsas.py file

② → in my_extsas.py →

```
from django import template
register = template.Library()
```

→ a library where tags and filters are registered

(template way)

```
@register.filter → if no arg used will take fn name as name
def cutout(value, args)
    return value.replace(arg, '')

@register.filter(name='lower')
def lower(value)
    return value.lower
```

```
def cutout( ⟶
    return ——

register.filter(
'cutout' 'cut')
```

fn name → python function

③ In index.html

```
<!DO ——
{% extend ——
{% load my_extsas %}  MUST
{% body ——
<h1> index </h1>
{{ text|cutout:'rl' }} ⟶ Hello World → Heo Wod
```

① @register.filter

② register.filter('name', python fn)

Django Passwords →

① create learning_user > basic_app > app
folder

② settings.py

→ make sure django.contrib.admin
" " .auth
" " .contenttype be there
in installed APPS

→ installed APPS
add basic_app

③ migrations

④ Installations →
learning_users > pip install bcrypt
> pip install django[argon2]

⑤ settings.py → above AUTH_PASSWORD_

add →
• → PASSWORD_HASHERS = [
'django.contrib.auth.hashers. PBKDF2 PASSWORDHASH

password validators
5 layers of Password hashing security
]

• → AUTH_PASSWOR VALIDATORS [
{'NAME' = ____ . USERATTR ____
Password is not same as user name
{'NAME' : '____ ____ MinimumLength ____
Add → 'OPTIONS': {'min_length': 9}
},
{'NAME': 'dja____ commonPa____
};

⑥ Create basic_app/templates/ folder & basic_app/ folder

content that you provide ← basic_app/static/ folder

content that user provide basic_app/media/ folder

⑦ In settings.py → at bottom after STATIC_URL

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

⑧ In models.py →

```
from django.db import models
from django.contrib.auth.models import User

class UserProfileInfo(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    portfolio_site = models.URLField(blank=True)
    profile_pic = models.ImageField(upload_to='basic_app/profile_pics', blank=True)

    def __str__(self):
        return self.user.username

* Remember to pip install pillow
```

to use this → pip install pillow

additional attributes

when ref deleted object is deleted it also deleted (CASCADE)

not nece to feed

built-in attribute of user

⑨ In forms.py →

```
from django import forms
from django.contrib.auth.models import User
from .models import UserProfileInfo

class UserForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput())
    class Meta():
        model = User
        fields = ('username', 'email', 'password')

class UserProfileInfoForm(forms.ModelForm):
    class Meta():
        model = UserProfileInfo
        fields = ('portfolio_site', 'profile_pic')
```

editing field

(10) In admin.py register the model.

```
from django.contrib import admin
from .models import UserProfileInfo, User

admin.site.register(UserProfileInfo)
```

* Migrate when →
  → change admin.py
  → create forms.py
  → changes in app

(11) create template > basic_app > base.html
index.html
registration.html
login.html

ONLY MAIN THINGS →

(12) → Creating base.html

urls.py (name) → views.py

If user is authenticated which is in views.py, will give an option to logout

```
{% if user.is_authenticated %}
<a class="nav-link" href="{% url 'logout' %}">Logout</a>
{% else %}
<a class="nav-link" href="{% url 'basic_app:user_login' %}">LOGIN</a>
{% endif %}
```

if not authenticated, then key is in urls.py → views.py (name)

(13) Creating index.html

If user is authenticated i.e in views.py then will show this

```
{% if user.is_authenticated %}
<h2> Welcome {{ user.username }}! </h2>
{% else %}
<h2> Welcome to site </h2>
{% endif %}
```

(14) Creating registration.html

```
{% if registered %}
<h1> thankyou </h1>
{% else %}
<form method="post" enctype="multipart/form-data">
{% csrf_token %}
{{ user_form.as_p }}
{{ profile_form.as_p }}
<input type="submit" name="" value="Register">
</form>
```

enctype="multipart/form-data" → to upload image (MUST)

} from views.py

(14) creating login.html

<form method = "post" action = "{% url 'basic_app: user_login' %}" > ↓ from views

{% csrf token %}

<input username >

<input password >

<input submit >

</form>

(15) a) Project > url.py

import admin
path, include
views

urlpatterns = [
path('', views.index, name = 'index'),
path('admin/', admin.site.urls),
special page ← path('special/', views.special, name='special'),
path('basic-app/', include('basic_app.urls')),
logout page ← path('logout/', views.user_logout, name='logout')
]

b) app > urls.py

import path
views

app_name = 'basic-app' ⟶ template tag urls

urlpatterns=[
path('register/', views.register
path('user_login/', views.user_login
]

(16) In views.py → write all imports. (see doc)

def index(request)
set

@ login_required
def special(request):
return HttpResponse("you are logged in")

Date Aug 2nd, 2020

Part 3 : Registration

Part 1 : Defining our views.py

```
from    django.shortcuts import render
from    basic_app.forms import UserForm, UserProfileForm

# extra imports for login & logout capabilities
from    django.contrib.auth import login, logout, authenticate
from    django.http import HttpResponse, HttpResponseRedirect
from    django.urls import reverse
from    django.contrib.auth.decorators import login_required

def index(request):
    return    render(request, "basic_app/index.html")

@login_required
def user_logout(request):
    logout(request)

    return    HttpResponseRedirect(reverse('index'))


@login_required
def special(request):
    return    HttpResponse("Logged In!")
```

Views.py

↳ whenever you want to check if a user is logged in or not

the decorator has to be directly above the fn

this means that the user must be logged in inorder to execute this function.

PTO

MATRIKAS

Scanned with CamScanner

Date Aug 2nd, 2020

```
def register (request):
    registered = False
    if request.method == 'POST':
        registered = False          --> tells if the User have registred or not!

        user_form = UserForm (data = request.POST)
        profile_form = UserProfileForm ( data = r " )

        if user_form.is_valid() and profile_form.is_valid() :

            user = user_form.save()

            user.set_password ( user.password)  <-- hash the password

            user.save()  <-- Update with saved hashed password

            # Now we'll deal with extra info

            profile = profile_form.save ( commit = False )

            profile.user = user  <-- Establish one to one relat b/w UserForm & UserProfileForm

            if 'profile_pic' in request.FILES
                print ('found it!')

            profile.profile_pic = request.FILES['profile_pic']

            profile.save()  <-- Now save model
            registered = True  <-- registration was succesful

        else:
            print ('user_form.errors, profile_form.errors)
```

Margin notes:

- views.py
- we'll send these as in one context dict & grab info from userform
- checks if they are valid. ie are not null, meets all conditions etc!
- saves User form to db
- this goes through settings.py file & it sets it as hash
- can't commit yet as we still have to manipulate data
- check if a user has provided a pic or not!
- this is how we'll extract pics, csvs etc
- if yes, then grab it from POST form reply
- if this get's called then one of the forms was invalid

MATRIKAS

Scanned with CamScanner

* Register your UserProfileInfo Model in basic_app admin.py file

Date Aug 2nd, 2020

```
else:
    user_form = UserForm()
    profile_form = UserProfileForm()

return render(request, 'basic_app/register.html',
                {'user_form': user_form,
                 'profile_form': profile_form,
                 'registered': registered})
```

Note! → Authenticate: prove or show to be valid, genuine or real.

Aug 2nd, 2020

Step3: Navigate to views.py to edit it!

After "register function":

```
def user_login(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')

        user = authenticate(username=username, password=password)

        if user:
            if user.is_active:
                login(request, user)
                return HttpResponseRedirect(reverse('index'))
            else:
                return HttpResponse('Your account isn't ACTIVE!')
        else:
            return HttpResponse('Invalid login details supplied.')
    else:
        return render(request, 'basic_app/login.html', {})
```

views.py

If this is true then the user has filled in the info.

this matches with the name we entered in our login.html when we created input for username

Checks if the user is authenticated or not!

returns the user to some page. In this case the home pg.

If we haven't submitted anything. Only the page have loaded then we return to 'login.html', fill the form there, & when we press SUBMIT then 'method == POST'.

* Now just adjust the URL's. Refer to the notes for better understd^n.

MATRIKAS

Scanned with CamScanner

* Never write fn name as import files

* username = request.POST.get('username')

grap name="username" from login.html

so get username from this POST

* user = authenticate(username = username, password = password)

this format is nece.

built in fn → will authenticate user.

* login(request, user)

built in fn

↓ object returned by authenticate

@login_required → filter decorator that check that only logged in users can log out.

logout(request)

↓ built in fn

Class Based Views

Normal →

① Set settings, template, base.html
index.html.

② In views.py →

```
from django.shortcuts import render
from django.urls import reverse_lazy
—— django.http —— HttpResponse
—— django.views.generic —— View
from . import models

class CBView(View):
    def get(self, request):
        return HttpResponse('Hey')
```

③ urls.py →

```
urlpatterns = [
    ~~path('', views.IndexView~~
    path('', views.CBView.as_view())
```

Template Class Based View →

views.py from django.shortcuts import render
django.views.generic View, TemplateView

→ inherited

```
Class IndexView(TemplateView):
    template_name = 'index.html'
```

attribute → loc in template

urls.py path('', views.IndexView.as_view())

Injecting template →

views.py → class IndexView

method in CBV

```
def get_context_data(self, **kwargs):
    context = super().get_context_data(**kwargs)
    context['injectme'] = "Basic Injection"
    return context
```

key to be placed in html file

**kwargs accepts args as dictionary

*arg - pass as tuple, when you dont know how many args to send, we accept any no. of arg.

Models

Models.py →

```
from django.db import models
                      reverse
django.urls
```

```
class School(models.Model):
    name = models.CharField(ma
    principal =
    location =

    def __str__(self):
```

```
class Student(models.Model):
    name =
    age = models.PositiveIntegerField   → +ve no.
    school = models.ForeignKey(School, related_name='students')   → to call it

    def __str__
```

admin.py

admin.site.register(School)
(Student)

~~template/basic_app/~~

views.py

from django.shortcuts import render
django.views.generic import View, TemplateView, ListView, DetailView
.models import models
class SchoolDetailView(~~Create~~DetailView):
context_object_name = 'school_detail'
model = models.School
template_name = 'basic_app/school_detail.html'
↳ loc in template folder

class SchoolListView(ListView):
model = models.School
will create a default object school_list
Or else create context_object_name = 'school'

set base.html, index.html)
(urls)

In school_list.html →
{% for school in school_list %}
(1) <h2>
{{ school.name }}
</h2>
{% endfor %}.

Inject context_object_name allotted to different classes in the html file

In urls.py of basic_app
app_name = 'basic_app'
→ for base.html url linkings
href = {% url 'basic_app:list' %}

url pattern=[
path('', views.SchoolListView.as_view(), name='list')
]

If don't define primary key, Django sets id as unique identifier for each entry.
i.e 1 for first entry, 2 for 2nd

(1) can also be written by linking →

```
<h2><li><a href = "{{ school.id }}" > {{ school.name }}
</a></li>
</li>
```

no. for school name entry

In school_detail.html →

```
{% for student in school_detail.students.all %}
<p> {{ student.name }} who is {{ student.age }}
    years old </p>
```

school_detail → object created
students → related name used in class
related_name relates to the class

* related_name → reverse connection of school to student.
taken in everything matching to key ('student') → class (2)

* for this in urls.py of app →

```
path ('<int:pk>', views.SchoolDetailView.as_view(), name = 'detail'
```

pk → primary key

CRUD
Create Retrieve Update Delete

Creating createView class →

① In views.py →

```
from django.views.generic import (View, TemplateView, CreateView,
    ListView, DetailView, UpdateView, DeleteView)
```

CreateView → create new editions to model
UpdateView → update existing entry
DeleteView → del

```
class SchoolCreateView(CreateView):
    fields = ("name", "principal", "location")
    model = models.School
```

→ expects default template school_form.html
↓ nece for creating new.

② urls.py app →

```
path('create/', views.SchoolCreateView.as_view(), name='create')
```

③ In school_form.html →

```
{% if not form.instance.pk %}
Create School
{% else %}
Update School
{% endif %}
```

→ check if instance of primary key exist not

★ Here when submit is clicked, error shows to do something to direct the url →

④ In models.py from django.~~core~~ import reverse
In School class → method

```
def get_absolute_url(self):
    return reverse("basic_app:detail", kwargs={'pk': self.pk})
```

go reverse to detail url page or whatever primary key is of the School you just made

Updating View →

① In views.py →

```
class SchoolUpdateView(UpdateView)
    fields = ("name", "principal")
    model = models.School
```

② In urls.py

```
path('update/<int:pk>/', views.SchoolUpdateView.as_view(), name='update')
```

(pk: primary key of school to update)

③ In school_detail.py →

```
<p> <a href = "{% url 'basic_app:update' pk=school_detail.pk %}"> update </a>
```

* An update button created by us will lead to a page form with principal & name input box.

delete View →

① views.py →

```
from django.urls import reverse_lazy
class SchoolDeleteView(DeleteView):
    model = models.School
    success_url = reverse_lazy("basic_app:list")
```

(success_url: once successfully deleted school go back to school list page)

(reverse_lazy used because we dont want to evaluate completely when running VI file, we wanted to wait till its actualy called a success)

② urls.py

```
path('delete/<int:pk>/', views.SchoolDeleteView.as_view(),
    name='delete')
```

③ add a school_confirm_delete.html file

<h1> Delete {{school.name}} </h1>

→ default use lower case of class name

<form method = "post">

{% csrf_token %}

<input type = 'sub

~~</form>~~

<a href = "{% url 'basic_app : detail' pk = school.pk %}

"> cancel

this url takes pk as parameter

through pass pk