Unit-1

Framework: like a frame that provides setup for developers to easily configure and get their job done quickly.

→ frameworks offer a structured approach, promoting consistency and organization in web app development.

→ dev can focus on specific functionality without worrying about infrastructure details.

→ fw streamline development
promote code reusability
facilitate maintainance and scalability of application
provide stdized s/w design structure

famous web fws:

Django

Flask - python based web dev - uses WSGI toolkit and Jinja2 template engine

Ruby on Rails - WA fw based on ruby - faster dev than Java

Angular - by Google, JS fw for powerful WA

ASP.NET - by Microsoft, .NET robust WA

Spring - Java dev fw enterprise high performance robustness

Play - Java and Scala - MVC archi - WA

Virtual environment

libraries, pkgs, fw addons, plugins } pre-written code in s/w or web development

→ fw - offers - prewritten code to implement various features in applications

→ Installation of req lib, plugs - necessary to use prewritten code

→ diff applin req specific fw, libs, plugins for diff features

→ Virtual environments ensure seperate setup for each project, avoiding conflicts and enabling simultaneous running of multiple projects with seperate

interpreters.

Create and use a virtual fw

1. Install python (visit Official python website > downloads
   > select OS > check mark "Add python to the PATH"
   > download > running installable file > choose loc
   > select ☐ to configure OS to use Py installation as default if no other installations are specified.
2. Open terminal
   > pip install virtual env
   (if pip not found error – repeat previous steps)
3. after 2, create a virtual env to
   > virtualenv myenv
4. Activate virtual env
   > myenv\Scripts\activate
5. Once activated,
   > pip install numpy
   Check: lib/site-packages → numpy installation
6. exit virtual envi
   > deactivate

Install Django

> pip install django
if specific version you need, use
> pip install Django == x.x.x
↑ version number

Create a Django project

Music player – project
playlists, popular songs – applications
↓ integrated & combined

1 Project ≙ 1/more applications together

> python -m django --version
> django-admin startproject mydjproject

Django project files

Once you create project using

> django-admin startproject djproject

It creates a folder inyour computer with this content

```
djproject
    manage.py
    djproject/
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
```

To run it and what it looks like in a browser,

1. navigate to ../djproject
2. > py manage.py runserver
3. Open a new browser and type
   127.0.0.1:8000 in address bar

~~Other~~ files:

1. db.sqlite3 - default SQLite db used by Django containing all db entries
2. manage.py - responsible for running Django-admin scrips
3. In internal Django project folder
   - settings.py → project configuration
   - __init__.py → indicates folders can be used as pkg; along with pycache folder
   - wsgi.py → for creating a web server gateway if.
   - urls.py → for URL mappings and corresponding actions.

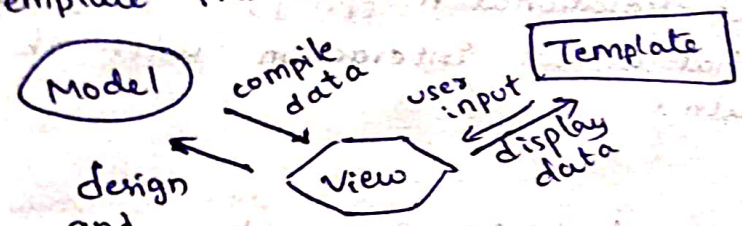
Architecture - proper setup or design
to connect components of a project
ensure optimal performance.

db, backend, frontend, server, n/w connections

Django's MVT → architecture fulfills requirement of
connecting db to backend
rendering data on the frontend
maintaining communication flow

MVT architecture (s/w design pattern)

Variation of MVC
→ Dj itself does work, with the help of templates, done done by controller in MVC
→ Template file combines ~~temp~~ HTML with Dj Template L



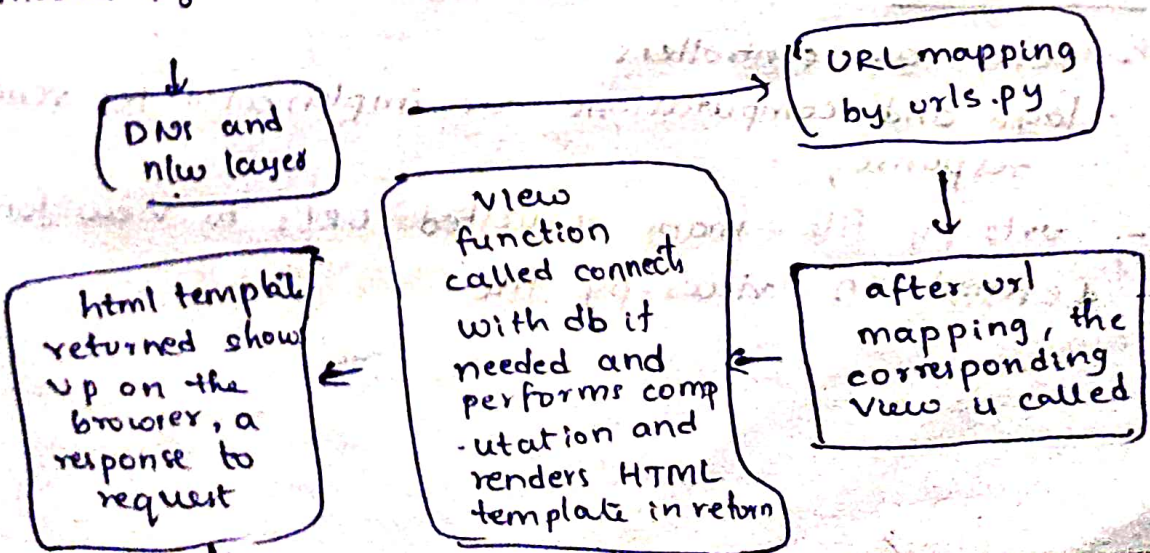
Model: responsible for establishing connection with db
each model → single table
handles db queries within the Django project.

View - does logical part of the application
interacts with the model to get data modifies
template accordingly

Template - handles UI part of an application.

→ handles req from client (browser) to server for DJ built website.
→ Server 1) receives client request
2) looks for requested URL in urls.py file
3) maps corresponding view func in views.py

models.py → to create tables in db and handle queries



Models in MVT

- serve as representations of database tables
- application describe type of data to be stored
- each model class - table, objects - rows
  attributes - columns
- provides easy way to interact with db tables, simplifies db operations
- creating an obj → making an entry into db table
- Django.db.models.Model (your model is subclass of)
- facilitate handling db operations in code regardless of prog. lang user, making it convenient to store and retrieve data.
- ORM (Object-Relational Mapping) programming paradigm that allows interaction with dbs using OOP concepts
- no SQL required

Ex: SQL Query

```
create table users (id INT primary key, name varchar(50),
    age int, email varchar(100));
```

model class

```
from django.db import models
class User(models.Model):
    id = models.IntegerField(primary_key = True)
    name = models.CharField(max_length = 50)
    age = models.IntegerField()
    email = models.EmailField(max_length = 100)
```

Views in MVT:

- serve as controllers
- logic and computations are implemented for rendering response,
- urls.py file maps requested URLs to view functions defined in views.py file

→ In urls.py, urlpatterns list contains path function calls that map expected URLs to view functions in views.py file.

→ Regular expressions (regex) used to define expected URLs in the first argument of path function calls

→ when a URL matches the expected URL, corresponding view function is called to handle the request.

View funs - loc in views.py
↑ execute the logic and return the template as a response to req

→ If no match, Django renders built-in 404 error template

Ex: urls.py.

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.home, name='home'),
    path('about/', views.about, name='about'),]
```

views.py

```
from django.shortcuts import render
from django.http import HttpResponse
def home(request):
    return HttpResponse("Welcome home!")
def about(request):
    return render(request, 'about.html').
```

Templates in MVT

→ allow you write HTML code and manipulate the content using Python code.

→ written in special way using tags.

→ provides default template engine but can also use other " " : Jinja2, Mako

→ Templates empower you to dynamically generate HTML content
implement loops, condi , reuse common HTML sec

→ enables you to handle static frontend dynamically through Python code.

Ex:

```
<!DOCTYPE HTML>
<html>
<head>
    <title>                </title>
</head>
<body>
    <h1> Welcome to {{ website_name }} </h1>
    <ul> {% for item in items %}
        <li> {{ item }} </i>
        {% end for %}
    </ul>
    {% if user_logged_in %}
        <p> welcome, {{ username }}!</p>
    {% else %}
        <p> Please log in to continue. </p>
    {% endif %}
</body>
</html>
```

Django Admin Utility

provides an admin portal that allows administrators to control the application without need for additional development.

- included with Dj two
- grows along with the application
- can be customized and configured as needed
- provides a working ORM, allows to interact with db and manage data through user friendly interface.

Admin page of your Django project:

Create an app

multiple apps → project
contributing to overall functionality
solve smaller dedicated problems

to create app : 2 ways

1. Django admin-tool
2. manage.py script

1. Go to terminal and navigate to project

> python manage.py startapp app1

creates an application - app1 inside your project

- Project root → Project root folder : contains envi and pro folder
  - mypro → project folder
    - mypro
      - __init__.py
      - settings.py
      - urls.py
      - wsgi.py
    - app1 → APP folder
      - migrations
        - __init__.py
      - admin.py
      - apps.py
      - models.py
      - tests.py
      - views.py
    - manage.py
    - db.sqlite3

admin.py : here you register model class of your app → so that they appear in admin app

apps.py : dj has default config for apps, while it's possible to make changes in the settings, it is recommended to proceed with caution

models.py : to define your model class

tests.py : to define test cases for testing

views.py : where your logic is imple and response is generated

migrations : to capture changes made to model classes and store them in migration files : tracking modifi and rollbacks

Editing models.py, settings.py and admin.py

1. Open settings.py
   add your appname.apps in installed_apps list
   → to tell Django that you have created an app

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes'
    'sessions'
    'messages'
    'staticfiles',

    'appl',
]
```

2. In models.py define model classes - appl.student with some attributes
   should inherit base models.Model class

open models.py

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

class StudentDetails(model.Model):
    gen_choices = [('M','Male'),('F','Female')]
    gen = models.CharField( choices = gen_choices,
          max_length=1, default = None, null = True)
    f_name = models.CharField( max_length=250)
    l_name =        , id =      .SlugField( "  ,unique=True)
    fathers_name =
    dob = models.DateTimeField()
    objects = models.Manager
    std_choices = [('1','I')...      ('10','X')]
    std = models.CharField( choices = std_choices,
          max_length=2, default = None, null=True)

    class Meta:
        verbose_name_plural = "Stu Details -
```

```
def get_absolute_url(self):
    return reverse('student:viewStudentDetails',
        args=[self.id])
```

admin.py

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

admin.site.register(StudentDetail)
```

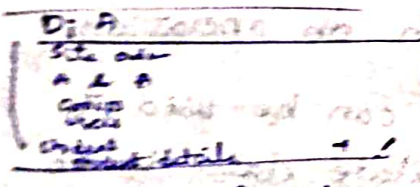
To tell ORM to create a table in db

- python manage.py makemigrations
- python manage.py migrate

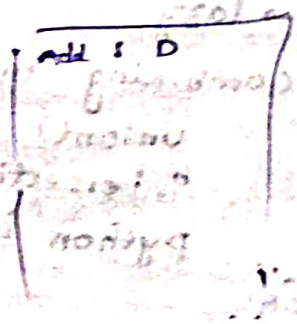
Adding data to your db

login to admin page by starting the server

observe student tab



Click student details
Click on add student detail to fill details



Save

Now you can observe a list of objects.

Editting data in your database

Click on any of the objects → form will pop up on screen → allows you to edit fields

If you edit id = existing id: → error coz unique = True attribute

Unit-2

Querysets

→ used in views to manipulate data in db programmatically
→ allows fetching and saving data from/to the db.

ORM → enables you to interact with dbs in a convinient manner.
→ Provides a built-in interface for admin functionalities, allowing easy mgmt of content.
→ compatible with MySQL, PostgreSQL, SQLite, Oracle
→ define db in settings.py file
→ supports working with multiple dbs simultaneously and allows customization through db routers.

1. Abstraction of db operations:
ORM provides powerful db abstraction API
2. Efficient db management:
manage db sche

Queryset

→ set of cmds that need to be evaluated to interact with db and perform db operations.
→ rep a SQL query and can be used to filter, retrieve and manipulate data in db.
→ lazy

Combining multiple Querysets
union()
intersection()
python list operations.

Eg:

```
from django.db.models import Queryset
from django.shortcuts import render
from .models import MyModel

def combine_querysets(request):
    qs1 = MyModel.objects.filter(attr1="value1")
    qs2 = MyModel.objects.filter(attr2="value2")
```

```
combined_qs = qs1.union(qs2)
return render(request, 'template.html',
              {'result': combined_queryset})
```

Operations on a Queryset:

Iteration	- Iterate over Queryset to access each record
Slicing	- slice over Qs using indices
repr()	- displays results in py interactive interpreter
len()	- get len of Queryset
list()	- save Qs results in a list.
bool()	check if Qs has any results.

methods

filter	- retrieve results matching spec condi
exclude	- retrieve " excluding " "
latest	- most recent obj that meets the c
Update	- updates rec, return no.of rows affect
Delete	- delet
Get	- retrieve single obj matching condi" (raises exception if multiple objs found)

Adding elements to your db:

ere following commands

① Navigate to project folder

```
> python manage.py shell
>>> from student.models import StudentDetail
>>> from datetime import datetime
>>> data = StudentDetail(.objects.create(
                    dob=datetime.strptime('2009-09-10',
                    "%Y-%m-%d").date(), std=6)
>>> data.save()
```

to check whether inserted

```
>all_students_list = StudentDetail.objects.all()
>all_students_list
[<StudentDetail: StudentDetail object (1)>, <
]>
>allstudents_list[4].first_name
```

Manipulating elements of your database

fetch

```
>> data = StudentDetail.objects.filter( student_id = 101)
>> data
 < QuerySet [<StudentDetail : StudentDetail object (5) >] >
>>> data[0].mothers_name
'Subadra'
```

modify particular field

```
>>> StudentDetail.objects.filter( student_id = 1090).update
        ( mothers_name = 'Subadra Khanna')
1
>>> data[0].mothers_name
'Subadra Khanna'
```

to iterate over queryset

```
>>> for obj in data:
        obj.fathers_name = 'Ramakant Khanna'
        obj.save()
>>> data[0].fathers_name
'Pankanth Khanna'
```

deleting elements of your database:

```
>>> data= StudentDetail.objects.get( student_id = 101)
>>> data.delete()
(1, {'student.Studentdail':1})
```

Introduction to Models

Models - for storing & retrieving data in your app

doc - " extensive, covering various aspects of model class, fields and their options

- rs understanding
- play central role in db schema design interaction in Dj

Model Fields = type of cols / attributes of a db table

AutoField : auto gen pk field
BooleanField : T/F field
CharField : for storing text
DateField : for storing date
DateTimeField : date & time
DecimalField
FloatField
EmailField
FileField - upl files
ImageField - imgs
IntegerField
SlugField - for gen URL friendly strings
Relationship fields : for est. rs between models

Meta Options — provide metadata about the model class

app_label : used when using a model class outside the app
default_manager_name : allows specifying a custom model manager
db_table -> sets name of table for model
get_latest_by - determines field used to retrieve most recent objs
managed - controls if model is translated into a table in db
ordering - sets default ordering of obj in retrieval objs
unique_together - spe fields that must be uniq together
verbose_name - sets a readable name for model
verbose_name_plural - define plural name for model on UI

Ex:
```
from django.db import models
class ExampleModel(models.Model):
    field1 = models.CharField(max_length=50)
    field2 = IntegerField()
    class Meta:
        app_label = 'myapp'
        d_m_name = 'objects'
        db_table = 'my_table'
        get_latest_by = 'field1'
        managed = True
        ordering = ['field1', '-field2']
        unique_together = ( : )
        verbose_name = 'Example'
        verbose_name_plural = 'Examples'
```

Model methods & classes

Model managers - implemented at table level
methods - row level

→ functions defined within scope of the model class used to perform various op" on individual model instances
→ invoked using instances
→ used for basic tasks 1. obj creation
adv purposes 2. generating absolute URLs
→ can be used in views, T or any other file where instance is available

Ex:
```
from django.db import models
class Book(models.Model):
    title =
    author=
    publication_yr=
    def is_published(self):
        curr_year = datetime.date.today().year
        return self.publication_yr <= curr_year
book = Book(title='A', author='B', pub_yr=2020)
print(book.is_published())
```

RS btw models : to implement r db concept and avoid repetitive storage of data

3 types

1. Many-to-one : imp using FK field one obj to many obj
2. One-to-One : OnetoOne field
3. Many-to-Many : ManyToMany field

FK field : has options like on_delete to specify behavior where the source object is deleted
limit_choices : limit choices available for field value

ManytoMany : Creates an intermediate model to represent rs between 2 models

OnetoOne : to connect 2 tables with info about similar things that cannot be stored in single table

→ to link tables
manage connected data efficiently.

refer to prog in pdf (5)

Connecting models

can connect models from diff apps / external files by importing model class using

→ from app-name.models import ModelName

✓ app containing model in INSTALLED-APPS in settings.py
✓ once imported, use it like any other model
✓ enable reuse model class across diff apps
✓ maintain proper rs and dependencies between models when connecting from diff sources

Views – Intro mundhu chusko peh

Types: Function-Based Views : V def as functions
Class-Based Views : classes
reusability & struct use

1. Built-in CBV
2. Base View : base cls for other CBVs
3. View : base cls for views handling HTTP req & res
4. Template View : renders a template along with context data
5. Redirect View : redirects to specified URL
6. Generic Views : predefined CBVs for common use cases
7. List View : displays list of objs from model
8. Detail View : details of spec objs from model

program - front

Configuration done in settings.py under

TEMPLATES tag.

prac. prog

BACKEND : specifies path to the template engine class for Django - T Lang : DjT

DIRS : indi where engine should find templates typically structured by app name.

APP_DIRS : set to True allows engine to search for templates within individual app directories

OPTIONS : can hold advanced settings for the template backend, relevant for complex proj

Template inheritance

child T from parent T - extends tag

✓ redundant HTML code ✗

✓ if prop - both parent & child → child's prop precedence

✓ overriding ✓

✓ inheritence creates a structure where child templates fill in data using block tags.

✓ Base T — serve as skeletons
child T - populate with content

Ex:

base.html

```
<!DOCTYPE HTML>
<html> <head> <title> </title>
    </head>
    <body> <header>
        <h1> Welcome! </h1> </header>
        <main>
            {% block content %} {% endblock %}
        </main>
        <footer>
            <p> &copy; 2023 My website </p>
        </footer>
    </body>
</html>
```

child.html

```
{% extends "base.html" %}
{% block content %}
    <h1> H1 </h1>
{% endblock %}
```

DJ Template Language

→ for creating templates that incorporate dynamic data into web pages

→ combines python & HTML to generate dynamic HTML responses

→ variables - {{ var_name }} return values from context passed by views

→ tags - {% tag_name %} allow exe of py code within templates

→ filters - modify values for display such as |date : "Y-m-d" for date formatting

default, default_if_none, time → available filters

→ comments - {# comments #}

not rendered in HTML output

Forms in Django

HTMC forms require manual HTML coding & validation

→ used for user input.

→ Dj forms 1) simplify process by automatically generating HTML form elements.

2) handle data validation ensuring user input is crct and safe.

3) process user ip to python ds for easy manipulation

4) can be tied to models - efficient to create forms based on db fields

5) simplifies process of collecting & storing user data in db.

Creating basic form

1. create Dj project and app

```
django-admin startproject basic-form-pro
cd basic-form-pro
python manage.py startapp formapp
```

2. create templates directory, and html files

create dpp > templates > app > index.html
form_page.html

3 index.html

```
<title> Home </title>
<body>
    <h1> welcome homes
    <h2> Go to homepage to fill out form </h2>
</body>
```

4 form_page.html

```
<title> forms </title>
<body>
    <h1> Fill out form </h1>
    <div class = "container">
        {{ form }}
</body> </div>
```

5. open settings.py of pro & update

```
TEMPLATE_DIR = os.path.join(BASE_DIR, 'basic_app/templates')
'DIRS': [TEMPLATE_DIR,],
```

P... ... -- info that is being transmitted.

6. In app folder
create forms.py

```
from django import forms
class FormName(forms.Form):
    name = forms.CharField()
    email = forms.EmailField()
    text = forms.CharField(widget=forms.
                           Textarea)
```

7. create views for homepage in views.py

```
from django.shortcuts import render
from . import forms
def index(request):
    return render(request, 'basic_app/index.html')

def form_name_view(request):
    my_form = forms.FormName()
    return render(request, 'basic_app/form_page.
                  html',{'form':
                  my_form})
```

8. updating urls.py file

```
from django.urls import path
    basic_app          views

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

9. open terminal and execute runserver command

Fetching data entered in forms

1. open views.py in basic_app folder

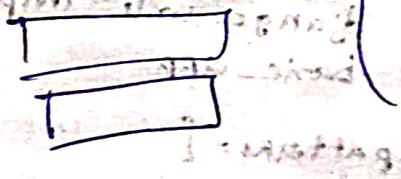
update form_name_view

```
def form_name_view(request):
    if request.method == 'POST':
        my_form_with_data = forms.FormName(request.POST)
        if                .is_valid():
            print('Validated')
            print("Name:" + my_form_with_data.
                                cleaned_data['name'])

            #
            #
    my_form = forms.FormName()

    return render(request, 'basic_app/form.html',
                          {'form': my_form})
```

else
print("form invalid"
return render(request, 'basic_app/form_page.html',
{'form': form_empty})

update index.html and form_page.html

reload page



Form fields and arguments

- CharField
- EmailField
- IntegerField
- BooleanField
- DateField
- DateTimeField
- ChoiceField
- ModelChoiceField
- ModelMultipleChoiceField
- FileField

→ required
label
label_suffix
initial
widget
help_text
error_messages
validators
localize
disabled

← django.core

form validation

form.py

```
django forms
django.core validators
def start-with-a(value):
    if value[0].lower() != 'a':
        raise forms.ValidationError('name should start with "A"')
class FormName(forms.Form):
    name = forms.CharField(validators = [start-with-a])
    email = forms.EmailField()
    text = forms.CharField(widgets = forms.Textarea,
                           max-length=100)
    rob-catcher = forms.CharField(required = False,
                  widget = forms.HiddenInput,
                  validators = [validators.MaxLength
                                Validator(0)])
```

Model Forms

update models.py

```
django.db
                models
class Enquiry(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    text = models.TextField()
```

form.py

```
from basic-app.models import Enquiry
class EnquiryForm(forms.ModelForm):
    class Meta():
        model = Enquiry
        fields = '__all__'
```

views.py

7th step and

urls.py

8th step

18. settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'contenttypes',
    'sessions',
    'messages',
    'staticfiles',
    'basic_app'
]
```

18. Execute makemigrations
migrate
runserver command.