FULLSTACK DEVELOPMENT (21CS62)

Module 4

Generic views -

- powerful way of simplify the creation of common views for tasks such as displaying DB objects, handling form submission, CRUD oprn etc.
- Generic views provide builtin functionality for common patterns, reducing amount of code we need to write.

i) import generic views in views.py -

```
from django.views.generic import DeleteView,
ListView, DetailView, CreateView, UpdateView.
```

ii) defining URL patterns - urls.py :

```
from django.urls import path
from .views import MyListView, MyDetailView, My
CreateView, MyUpdateView, MyDeleteView
urlpatterns = [ path ('mylist/', mylistView.asview(),
                name = 'my-list'),
              path ('mydetail/', myDetailview.asview(),
                name = 'my-detail'),
            { create, update, delete same }
```

iii) defining views by subclassing the appropriate generic views & specifying the model & template_name attributes.
Customize the view by overriding methods like
get_queryset(), form_valid()

```
from django.views.generic import ListView
from .models import MyModel
class MyListView(ListView):
    model = MyModel
    template_name = 'myapp/mymodellist.html'
```

iii) creating the template

v) Access URLs - access URL associated with your generic views in browser (or) link them to template using tags {% url %}

Generic views of objects

- pre built functionality to display list of objects, show details of single object, create new object, update, delete.

model.py

```
class Publisher(models.Model)
    name = models.CharField(maxlength=30)
    address = models.CharField( " )
    website = models.URLField()
    def __str__(self):
        return self.name
    class Meta:
        ordering = ['name']
```

urls.py

```
from django.config.urls import *
from django.views.generic import list_detail
from mysite.books.model import publisher
```

publisher_info = {'queryset': Publisher.objects.all()}
urlpatterns = patterns('', (r'^publishers/$', list_detail.
object_list, publisher_info))

publisher_info → dictionary → contain conf. for generic views
queryset ⇒ specifies set of 'Publisher' objects to be used in view

specify a template,

from django.conf.urls.defaults import *
from django.views.generic import list_detail
from mysite.books.models import Publisher
publisher_info = {'queryset': Publisher.objects.all(),
'template_name': 'publisher_list_page.html',}
urlpatterns = patterns('', (r'^publishers/$', list_detail.
object_list, publisher_info))

template specification ⇒ adds 'template_name' to 'publisher_info' to explicitly set the template used for rendering.

template_name: specifies the custom template 'publisher_list_page.html' to be used instead of default.

Template ~~template~~ Example

{% extends "base.html" %}
{% block content %} <h2>Publisher</h2>
 {% for publisher in object_list %}
 {{ publisher.name }}
{% endfor %}
{% endblock %}

Customizing Generic Views

- **info dict**: dict passed to generic view can be customized to include additional options.
- **template context**: additional context variables can be passed to template by modifying the dict

Extending Generic Views (IOM)

- Extending generic views in Django allows us to customize & add functionality to existing generic views to better suit our appln's req.
- We can extend generic views by subclassing them, overriding them (methods / attributes) or by combining them with mixins (reusable components that provide additional functionality to views)

mixins usage

Eg:- views.py,

```
from django.views.generic import ListView
from .models import myModel
class CustomListView(ListView):
    model = myModel
    templatename = 'myapp/mycustomlist.html'
    def get_queryset(self):
        queryset = super().get_queryset()
        return queryset.filter(condition=True)
```

urls.py

```
import path, .views import CustomListView
urlpatterns = [path('custom-list/', CustomListView.as_view(),
                    name = CustomList'),]
```

3) create template for custom view { mycustomlist.html }
4) access URL associated with custom view in browser or link it to template using template tag.

- Extending using mixins,

```
from django.view.generic import ListView
from django.contrib.auth.mixins import LoginRequiredMixin
from .model import myModel
class CustomListView (LoginRequiredMixin, ListView):
    model = MyModel
    templatename = 'myapp/mycustomlist.html'
```

extra content

```
publisher_info = { 'queryset': Publisher.objects.all(),
template_object_name : 'publisher',
'extra_content': { 'booklist': Book.object.all() }.
# adds booklist containing all books to template's context
```

Call Back Fn

```
# handling dynamic queryset with callbacks.
def getbooks(): return Book.objects.all()
publisher_info = { 'queryset': Publisher.object.all(),
'template_object_name': 'publisher',
'extra-content': { 'book_list': getbooks } }
```

- Filtering querysets, wrapping fns
- Perform Extra work

MIME Types (Multipurpose Internet Mail Extension)

- Standard way of indicating the type of data that a file contains on Internet.
- They are used by webservers & browsers to interpret & handle different types of files accordingly.

- Plan Text – apple.txt file – simple text
- HTML – apple.html file – web page
- CSS – apple.css file – styling web page.
- Image – JPEG – apple.jpeg – complex color detail photos
  - PNG – apple.png – transparent background, simple
  - GIF – apple.gif – simple animation
- Audio – MP3 – apple.mp3 – music file
  - WAV – apple.wav – sound file
- Video – MP4 – video file
  - WebM – webm video file
- JSON – Structured data file (JavaScript Object Notation)
- PDF – Portable Document Format
- XML – eXtensible MarkUp Lang – Structured data file

Generating CSV & PDF files using Django (10)

```
import csv
from django.http import HttpResponse
UNRULY_PASSENGERS = [146, 147, 148, 149, 150, 200, 211,
                     200, 234, 207, 266]
```

↓

List of no representing count of unruly airline passengers from each year from 1995 – 2007.

```
def unruly_passengers(request):
response = HttpResponse(content_type='text/csv')
response['Content-Disposition'] = 'attachment;
    filename="unruly.csv"'
writer = csv.writer(response)
writer.writerow(['Year', 'Unruly Airline Passengers'])
for year, num in zip(range(1995, 2008), UNRULY_PASSENGERS):
writer.writerow([year, num])
return response.
```

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def unruly_passengers(request):
    response = HttpResponse(content_type='appln/pdf')
    response['Content-Disposition'] = 'attachment;
    filename = "unruly.pdf"'
    p = canvas.Canvas(response)
    p.drawString(100, 800, "Year Unruly Airline Passengers")
    y = 700
    for year, num in zip(range(1995, 2008), UNRULY_PASSENGERS):
        p.drawString(100, y, f"{year} {num}")
        y -= 20
    p.showPage()
    p.save()
    return response
```

1

2

Syndication Feed Framework -

- high level framework for generating RSS & Atom feeds.
- create multiple feed using simple Python classes.
- These feeds can be consumed by users to stay update with latest content from our site.

Eg:- 1) define feeds - each feed class represent a diff type of feed & content to include.

```
from django.contrib.syndication.views import Feed
from .models import Post
Class LatestPostFeed(feed):
    title = "My Blog"
    link = "/blog/"
    description = "Latest Post"
    def items(self):
        return Post.objects.orderby('publisheddate')[:5]
    def item_title(self, item):
        return item.title
    def item_description(self, item):
        return item.content
```

2) configure URL pattern: map URLs to feed views

```
from myapp.feeds import LatestPostFeed
urlpattern = [ path('latest/feed/',
    LatestPostFeed(), name='latestPostfeed'), ]
```

3) Include feed url in settings

```
urlpattern = [path('feeds/', include('myapp.feeds'))]
```

4) access by visiting corresponding URLs in web browser

Site Maps

- generate XML sitemap files for our website.
- sitemap is XML file that helps search engine index our site.
- Tells search engine how frequently pages change & their importance.

S1) define app/sitemaps.py - each sitemap class represents a different section / type of content on our site & specifies URLs to include in sitemap.

```
from django.contrib.sitemaps import Sitemap
from .models import Post
class PostSitemap(Sitemap):
    changefreq = 'weekly'
    priority = 0.9
    def items(self): → return Post.objects.all()
    def lastmod(self, obj) → return obj.modified_date
```

S2) configure URL pattern in project's urls.py:

```
    import path
from d.contrib.sitemaps.views import sitemap
from myapp.sitemaps import PostSitemap
sitemaps = {'posts': PostSitemap',}
```

S4) access sitemap :- http://abc.com/sitemap.xml/

Customize sitemaps → items(), lastmod()

Cookies & Sessions

Cookies :- small piece of code, stored in client browser
- they are sent to server with every HTTP request, allowing server to track & identify individual users
- key : value pair storage
- can expire after certain period.
- used for user authentication, personalization
- vulnerable to security risk (CSRF, XSS) if not handled safely.

Sessions - server side data storage mechanisms
- maintain stateful info about user's interaction with webpage. Each session has identifier (session ID) that is sent to client as cookie
- session - stored in server, ID - client browser
- used for authentication, tracking user activity, storing temp data across multiple HTTP Request
- more secured by session data stored on server & not accessible to client.
- can store large amount of data

Eg:- Cookies & session handling.

myproject / myapp / → views.py

```
from django.shortcuts import render
from django.http import HttpResponse

def set_cookie(request):
    response = HttpResponse("Set Successfully")
    response.set_cookie('my_cookie', cookie_value)
```

return response

```
def get_cookie(request):
mycookie = request.COOKIES.get('my_cookie')
return HttpResponse(f"Value of my cookie {mycookie}")
```

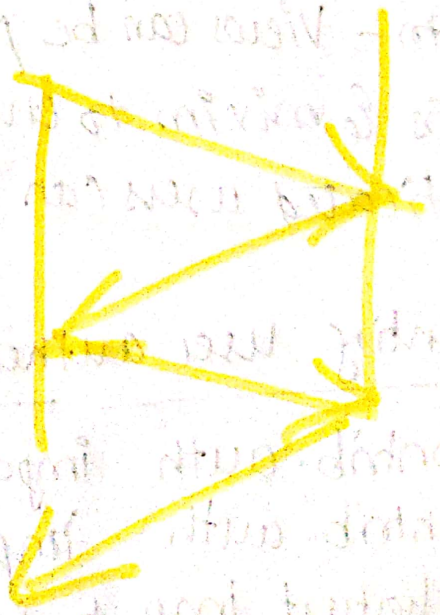
urls.py

```
from django.urls import path
from .views import set_cookie, get_cookie
urlpattern = [ path('set-cookie/', set_cookie,
                    name='set-cookie'),
               path('get_cookie/', get-cookie,
                    name=`get-cookie'),]
```

settings.py

```
MIDDLEWARE =
'django.contrib.sessions.middleware.sessionMiddleware',]
```

```
python manage.py migrate
python manage.py runserver
```



User & Authentication -

Users :-

- User Model - Django provides inbuilt user model (django.contrib.auth.models.User)
- User registration - Username, Email, Password
- User Profile - additional info
- User manag - admin can manage users, creating, updating & deleting accounts of user.

Authentication -

- Login, Logout, Session Management
- Password Management

Permission & Authorization -

- permission - permission system to control access to views & functionality within app.
- Groups - users can be organised into groups & permission can be assigned to groups.
- Authorizato - Views can be protected using decorators & mixin to ensure that only authenticated users can access them.

Eg :- Implementing user authentication

```
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.models import User
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
```

```
def user_login(request):
if request.method == "POST":
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(request, username=uname, password=passwd)
    if user is not None:
        login(request, user)
        return redirect('HOME')
    else:
        return render(request, login.html {Error})
else:
    return render(request, 'login.html')
```

← logout →

```
def user_logout(request):
    logout(request)
return redirect('login')
```

```
def home(request):
return render(request, 'home.html')
```

login

Generic Views Example

```
from django.conf.urls.default import *
from django.views.generic.simple import
direct_to_template
urlpatterns = patterns('', (r'^about/$', direct_to_template,
    { 'template': 'about.html' }) )
```

- direct_to_template → simple generic view,
  - provide render a template
  - allows us to render a specific template directly in the response to URL request
- More advanced versions of generic views which work with model data & offer more functionality are ListView, DetailView