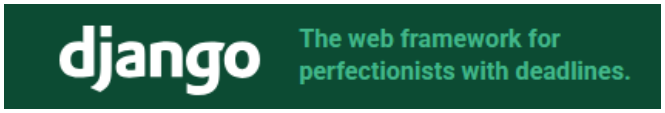
Django Notes



- **Django makes it easier to build better web apps more quickly and with less code.**

Introduction to Django

- Django is a free, open-source, Python-based web framework used for developing web applications.

Key Points:

- **Free:** No cost is required to develop commercial applications using Django.
- **Open Source:** The source code is publicly available, allowing customization and modification according to our requirement and creating new frameworks (e.g., Chitkarago).

Examples of Open-Source Software:

- **Operating Systems:** Linux/Unix and its various distributions like Ubuntu, Fedora, HP-Linux, CentOS.
- **Python Variants:**
    - IronPython (for C#)
    - Jython (for Java)
    - Anaconda (for Data Science)

Django Framework Overview

- Django is written in Python.
- It follows the **MVT (Model-View-Template)** architecture.
    - **Model** → Handles database interactions.

    - **View** → Contains business logic and processes user requests.
    - **Template** → Controls the presentation layer (HTML).

- It is maintained by the **Django Software Foundation (DSF),** a non-profit organization.
- Comparable to other technologies:
    - **Java → Maintained by Oracle**
    - **C# → Maintained by Microsoft**

Popular Websites Built with Django:

- YouTube
- Google
- Dropbox
- Yahoo

History of Django

- Created in 2003 as an internal project for the *Lawrence Journal-World* newspaper.
- Developed by Adrian Holovaty and Simon Willison.
- Released for public use on July 21, 2005.
- Named after the famous guitarist Django Reinhardt.

Official Website:

[Django Project]

Django's Tagline: ***"The web framework for perfectionists with deadlines."***

Current Version:

Django 5.1 (Check the official site for updates).

Top 5 Features of Django

1. Fast Development

- Django automates most of the development process, reducing coding efforts.
- Out of 100% work, minimum 95% work will be done by Django, only for 5% developers are responsible.
- Example:
    - A task that requires 20+ lines in **Servlet(Java)** can be done in just 2-3 lines in Django.
    - A basic project can be developed within a single day.

2. Fully Loaded

Django provides various built-in libraries for common functionalities, such as:

- User Authentication
- Email Handling
- Database Management

3. Security

Django protects against common security threats, including:

- SQL Injection
- Cross-Site Request Forgery (CSRF)

4. Scalability

Django supports high traffic loads(multiple client requests simultaneously), making it scalable.

Example:

- Initially handling 100 requests, later scaling up to 500,000 requests without performance issues.

5. Versatile

Django is used in various domains, such as:

- YouTube → Video streaming
- NASA → Scientific applications
- Universities → Educational platforms

For more details, visit:

Django Project Overview

Installing Django:

Prerequisites:

- Ensure that **Python** is installed on your system.

Installation Command:

```
pip install django
```

Installing a Specific Version:

- To install a particular version of Django, use:

```
pip install django==3.2.4
```

Checking the Installed Django Version:

```
python -m django --version
```

Django Project vs Django Application

Django Project:

- A Django project is a collection of multiple applications along with configuration files, forming a complete web application.

Django Application:

- A Django application is a modular component of a project that can be reused across different projects.

Django Project = django applications + configuration informations

Example:

A Banking Project may contain multiple applications:

- **Loan App**
- **Insurance App**
- **Customer Management App**
- **Configuration Files (Database settings, Middleware, etc.)**

Key Points:

- A Django project consists of multiple Django applications + configuration files.
- Individual applications can be reused in different projects.
- Multiple developers can work on different applications simultaneously.
- A Django application cannot exist without a Django project.
- First, we create a Django project, and then we create applications inside it.

Creating a Django Project:

Using django-admin Command:

- Django provides the django-admin command-line tool to create projects.

```
django-admin startproject project1
```

Project Directory Structure:

- After running the command, the following structure will be created:

```
project1/
|-- manage.py
|-- project1/
   |-- __init__.py
   |-- asgi.py
   |-- settings.py
   |-- urls.py
   |-- wsgi.py
```

- Outer project1/ → The project folder.
- Inner project1/ → Stores project settings and configurations.

Understanding Django Project Files:

1. **__init__.py (Package Indicator):**
    - if any folder contains __init__.py file then that folder is treated as a python package. But this rule is applicable only up to python 3.3 version, it is an empty file.
2. **settings.py (Project Configuration):**
    - Stores project settings, similar to mobile phone settings.

- All the project related settings we have to configure here
- Contains configurations for:
    - Installed apps
    - Database settings
    - Middleware
    - Static files handling

3. **urls.py (URL Routing):**

- Here we have to store all the urls of our projects, basically mappings to the view functions.

- A Django project contains the django apps and each app may contain the various **view functions** which are responsible to trap the request and generate the response.
- For every view we have to define a separate url pattern inside this file.
- These views can be a class based view**(CBV)** or a function based view**(FBV)**.

4. **wsgi.py (Web Server Gateway Interface)**
    - Required for deploying the application to **production servers**.
    - Helps Django interact with web servers like **Gunicorn, Apache, or Nginx**.
    - The required configuration is required in this file

5. **asgi.py (Asynchronous Server Gateway Interface)**

- Handles **asynchronous** operations in Django.
- Enables real-time applications like chat, notifications, and WebSockets.

Difference Between Synchronous & Asynchronous:

- **Synchronous (WSGI):** A phone call – the next action happens only after the call ends.
- **Asynchronous (ASGI):** A text message – you send it and continue with other tasks.

- 🔹 Modern web applications prefer asynchronous processing for better performance.
- 🔹 Python implements async using coroutines, async, and await keywords.

6. **manage.py (Django Management Utility)**

- This file contains the command-line utilities for administrative tasks.
- It contains various scripts to manage our application like:
  - To start the django server
  - Database migrations
  - Creating applications

Example:

- python manage.py runserver # Start the Django server
- python manage.py migrate # Apply database migrations
- python manage.py startapp app_name # Create a new app

Note: Django by default uses its own integrated web server and sqlite database, but we can configure our own server or any other RDBMS s/w also.

Note: Inside the above generated files, only 3 files is required as a beginner:

1. manage.py
2. settings.py
3. urls.py

Django's Pre-Installed Apps:

- Inside settings.py, Django includes built-in applications:

```
INSTALLED_APPS = [
    'django.contrib.admin',      # Admin panel
```

```
    'django.contrib.auth',      # Authentication system
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

- These are the already provided applications by Django with a startup project.

Adding a Custom Application

- If you create an app named **testapp**, add it to INSTALLED_APPS:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'testapp',  # Custom app added manually
]
```

Running the Django Development Server:

- Django itself provides an integrated webserver.
- Move inside the **project1** main folder where "**manage.py**" file is available

```
cd project1
```

- Run the following command to start the server

```
python manage.py runserver
```

- Access the project at:
    - http://127.0.0.1:8000/

- To run the server at different port:

```
python manage.py runserver 7777
```

- Now, access it at:
    - http://127.0.0.1:7777/

Creating our first application inside the project project1:

Syntax:

```
python manage.py startapp <app_name>
```

Example:

```
python manage.py startapp testapp
```

- It will create another folder inside the main folder **project1** with some files:

```
testapp/
|-- migrations/
|    |-- __init__.py
|-- __init__.py
|-- admin.py
|-- apps.py
|-- models.py
|-- tests.py
|-- views.py
```

- **Purpose of Each File**

1. admin.py:
    - To customize the admin page
    - We can register our models in this file, to use those models in django admin interface
2. models.py:
    - Application level configurations(application specific) we need to define inside this file.
3. apps.py:
    - Application specific models we have to define inside this file.
4. test.py:
    - To define test cases to test our app functionality
5. views.py: (most important file)
    - In this file we have to define a view function to handle requests and responses.
6. migrations folder:
    - This folder is used to hold the Database migrations related information.

Activities required for the application:

1. After creating an app, register it inside INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'testapp',  # Add this line
]
```

2. Create views for our application to provide the required functionalities inside the **views.py** file present inside our application "**testapp**".

   Django supports two types of views:

   1. **Function-Based Views (FBVs)**
   2. **Class-Based Views (CBVs)**

   Example of Function-Based View

   Edit views.py inside testapp:

   - Here we have to define a view function to handle the request and generate the response to the end user.

```
from django.shortcuts import render

from django.http import HttpResponse

def display(request):

    return HttpResponse("<h1>Welcome to Django!</h1>")
```

   - Each view function should take at least one argument: **request**, which represents the **HttpRequest** object, this view function will be called by Django internally by passing the request object.
   - The view function **must return** an HttpResponse object.

3. Define the url-pattern for our view function inside the **urls.py** file(project level).

```
from django.contrib import admin
```

```
from django.urls import path

from testapp import views  # Import the view

urlpatterns = [

    path('admin/', admin.site.urls),

    path('hello/', views.display),  # Map the view to a URL

]
```

Access the View

Visit:

http://localhost:8000/hello

Note: to mention the root url we need to use as follows:

```
path('', views.display), #django internally manages '/' to start the url
```

Assignment:

Develop an application in the above Django project to display the current date and time on the webpage.

Solution:

Step1: create another app called **dateapp** inside the above project.

```
python manage.py startapp dateapp
```

Step2: register this app inside the settings.py file at project level

```
INSTALLED_APPS = [
```

```
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'testapp',
    'dateapp'
]
```

Step3: Define the view function inside the views.py file inside the dateapp folder

```
from django.shortcuts import render
from datetime import datetime
from django.http import HttpResponse
# Create your views here.
def get_date_time(request):
    time = datetime.now()
    return HttpResponse(f"<h1>The Current date time is: {str(time)} </h1>")
```

Step4: Define the url mapping for the above view function inside the **urls.py** file of **project1** folder**(at the project level)**

```
from testapp import views as v1

from dateapp import views as v2


urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello/', v1.display),
    path('date/', v2.get_date_time),
]
```

Step5: run the server

```
python manage.py runserver
```

Step6: access the app using following url:

http://127.0.0.1:8000/date/

Note: inside the single Django project we can define multiple applications and inside a single application we can have multiple views also.(multiple view function inside the views.py file).

we can create multiple view functions inside the views.py file and then need to provide url mapping information to each of the view functions separately.

Defining the url-patterns at the application level instead of project level:

Until now, we have been defining URL patterns at the **project level** inside the urls.py file. However, this approach is **not recommended** for large projects.

Why Should We Define URLs at the Application Level?

1. **Better Modularity**:
    - A Django project may contain multiple applications. If we define **all URL patterns in the project-level urls.py**, the file can become **large and unorganized**, making it harder to manage.
2. **Avoid Mixing URLs from Different Applications**:
    - If each application has **100+ views**, managing all URL patterns in a **single urls.py** will cause clutter and reduce readability.
3. **Improved Reusability**:
    - Defining URLs inside an application-level urls.py allows us to **reuse** the entire application in another project **without modifying URLs**.
    - We can simply **copy and paste** the application folder into another Django project and include its URLs easily.

Recommended Approach: Application-Level URL Configuration

Step 1: Create an urls.py File Inside each Application

- Inside each application (e.g., testapp) and (dateapp), create a file named urls.py (if not already present).
    - urls.py inside the **testapp** folder:

```
from django.urls import path

from . import views

urlpatterns = [

   path('hello/', views.display)

]
```

    - urls.py inside the **dateapp** folder:

```
from django.urls import path

from . import views

urlpatterns = [
```

```
    path("date/", views.get_date_time)
]
```

Step2: Include Application-Level URLs inside the Project's level urls.py file.

- urls.py inside the **project1** subfolder.

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('testapp/', include('testapp.urls')),
    path('dateapp/', include('dateapp.urls'))
]
```

Step3: Give the request as follows:

http://127.0.0.1:8000/dateapp/date

http://127.0.0.1:8000/testapp/hello

Advantages of Using Application-Level URLs:

✅ **Keeps the project-level urls.py clean and organized**
✅ **Improves maintainability and readability**
✅ **Enhances reusability – applications can be used in multiple projects**

Steps to develop template based application:

- In Django, templates help separate the **presentation logic (HTML)** from the **business logic (views.py)**, making the application more modular and maintainable.

Step 1: Create a Django Project

- Use the following command to create a Django project:

```
django-admin startproject UniversityProject
```

- Navigate into the project directory:

```
cd UniversityProject
```

Step 2: Create a Django Application:

- Inside the project directory, create a Django application named `StudentApp`:

```
python manage.py startapp StudentApp
```

Step 3: Register the Application in settings.py:

- To make Django aware of the new application, add **`StudentApp`** to the **`INSTALLED_APPS`** list inside **`settings.py`**:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'StudentApp', # Registering the application
```

]

Step 4: Create the templates/ Folder

- Inside the **StudentApp folder**, create a **templates/** directory to store templates specific to the StudentApp..

UniversityProject/

| -- manage.py

| -- **UniversityProject/**

| -- **StudentApp/**

| -- **templates/** # All templates for StudentApp application

Step 5: Create a Template (index.html)

- Inside the **StudentApp/templates** folder, create an HTML file named index.html:

StudentApp/

| -- **templates/**

├── index.html

- Add the following content to index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>Wish Page</title>
</head>
<body bgcolor="cyan">
    <h1>Welcome to the Student App!</h1>
    <p>This is the Student Home Page</p>
</body>
</html>
```

Step 6: Define a View in views.py of **StudentApp/** folder

- Inside the **StudentApp/views.py**, define a view function to render index.html:

```
from django.shortcuts import render
def home_view(request):
    return render(request, "index.html")
```

Step 7: Define a URL Pattern

- **Define URLs at the App Level (StudentApp/urls.py)**
    - **Inside the StudentApp folder, create a urls.py file:**

```
from django.urls import path
from . import views
urlpatterns = [
```

```
    path("", views.home_view),
]
```

Include this StudentApp's urls.py file in the main urls.py file at project level (UniversityProject/urls.py file):

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('student/', include('StudentApp.urls')),  # Include URLs from StudentApp
]
```

Step 8: Start the Development Server

- Run the following command to start the server:

```
python manage.py runserver
```

Step 9: Send a Request and Test the Application

- Open your browser and visit:

  http://127.0.0.1:8000/student/

Django Template Tags

What are Template Tags?

- Template tags allow us to inject dynamic content (data) from Python's views.py into an HTML template file.
- Template tags are also known as template variables.
- Django template tags follow the Jinja2-style syntax (similar to Flask).

Syntax for Inserting Dynamic Data:

- inside the html template
    - {{variable_name}}

Example: Using Template Tags in Django

Step 1: Define a View in views.py

- **Modify the views.py file to pass data to the template.**

```
from django.shortcuts import render
from datetime import datetime
# Create your views here.
def home_view(request):
    date = datetime.now()
    my_dict = {"msg": date}
    return render(request, "index.html", context=my_dict)
```

Step 2: Use Template Tags in index.html

- Inside the index.html file, use {{ msg }} to display the dynamic data.

```
<body>
    <h1>Welcome to the Student Home Page!</h1>
    <p>The Current Date and Time is :  {{msg}}</p>
</body>
```

Alternative Approach: Passing Data Directly

- Instead of defining a dictionary separately, we can pass data directly in the render() function

```
def home_view(request):
    date = datetime.now()
    my_dict = {"msg": date}
    return render(request, "index.html", my_dict)
```

- Or we can pass the dictionary directly:

```
def home_view(request):
        date = datetime.now()
        return render(request, "index.html", {"msg": date})
```

Note: Unlike Flask, where we can pass variables directly to templates, **Django requires passing a dictionary** (as context) when sending data to templates.

Passing Multiple Data from views.py to Template:

- In Django, we can pass multiple pieces of data from views.py to an HTML template using a dictionary as the context.
- **Example:** Passing Multiple Data Items

Modify views.py in StudentApp:

```
from django.shortcuts import render
from datetime import datetime
def home_view(request):
    date = datetime.now()
    my_dict = {"date": date, "name": "Chitkara University", "batch": "CSE(AI)", "year": 2024}
    return render(request, "index.html", my_dict)
```

Modify index.html Template:

```
<body>
    <h1>Welcome to Student Home Page</h1>
    <p>Have a great day!</p>
    <h3>The Date and Time is: {{ date }}</h3>
    <h3>University Name: {{ name }}</h3>
    <h3>Batch: {{ batch }}</h3>
    <h3>Year: {{ year }}</h3>
</body>
```

Random city selection example:

- To generate a dynamic city name and pass it to the template:

Modify views.py:

```
import random

def home_view(request):
    cities = ["Chandigarh", "Mohali", "Amritsar", "Jalandhar"]
    selected_city = random.choice(cities)
    return render(request, "index.html", {"city": selected_city})
```

Modify index.html:

```
<body>
    <h1>Welcome to the Wish Page!</h1>
    <h3>The Selected City is: {{ city }}</h3>
</body>
```

Assignment:

- **Pass the list of 5 city names from the view function to the template and display them inside the template (HTML) using <li> tag.**

Working with Static Files (CSS, Images, JavaScript) in Django:

- Static files like CSS, images, and JavaScript need to be placed inside the static folder and referenced correctly in templates.

Steps to include the static files inside the html templates:

Step1: Create a folder named with "**static**" inside the application folder. same as templates folder.

Step2: Inside the **StudentApp/static** folder create the "**images**" folder to keep all the images, and the "**css**" folder to keep all the external css files.

```
StudentApp/
        |-- static/
        |      |-- images/
        |      |-- css/
```

Step3: Using Static Files in Templates

(a) Load Static Files in Templates

- At the beginning of your HTML file, load the `static` tag:

  <!DOCTYPE html>

  {% load static %}

(b) Referencing an Image File

```
<img src="{% static 'images/sh1.jpg' %}" alt="oops image not found" width="200px" height="200px">
```

(c) Linking an External CSS File

```
<link rel="stylesheet" href="{% static 'css/a1.css' %}">
```

index.html:

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Wish Page</title>
    <link rel="stylesheet" href="{% static 'css/a1.css' %}">
</head>
<body>
    <h1>Welcome to Student Home Page!</h1>
    <h3>The City is: {{citydata}}</h3>
    <img src="{% static 'images/sh1.jpg' %}" alt="oops image not found"
width="200px" height="200px">
</body>
</html>
```

a1.css:

```
body{
    background-color: aquamarine;
}
```

Now access the application using:

http://127.0.0.1:8000/student/

Inter-Template Communication in Django:

- Django templates **do not communicate with each other directly**. Instead, templates interact through **views and URL routing**.

Example: Navigating Between Templates

1. Create a New `dashboard.html` File

- Place this inside `StudentApp/templates` folder

```
<body bgcolor="cyan">
    <h1>Welcome to Dashboard</h1>
</body>
```

2. Modify the View Function in views.py in **StudentApp** folder

```
from django.shortcuts import render
from datetime import datetime
import random
def home_view(request):
    cities = ["Chandigarh", "Mohali", "Amritsar", "Jalandhar"]
    city = random.choice(cities)
    return render(request, "index.html", {"citydata": city})
def dashboard_view(request):
    return render(request, "dashboard.html")
```

3. Configure URLs in `urls.py (at application level)`

```
from django.urls import path
from . import views
urlpatterns = [
```

```
        path("", views.home_view),
        path("dashboard/", views.dashboard_view),
    ]
```

4. Add Navigation Links in **index.html**

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Wish Page</title>
    <link rel="stylesheet" href="{% static 'css/a1.css' %}">
</head>
<body>
    <h1>Welcome to Student Home Page</h1>
    <h3>The City is: {{citydata}}</h3>
    <img src="{% static 'images/sh1.jpg' %}" alt="oops image not found"
width="200px" height="200px">
    <a href="./dashboard">Go To Dashboard</a>
</body>
</html>
```

Assignment:

Inside the **index.html** file of the StudentApp create a link called **Get All Student Details**

when clicking on this link it should display the following Students list inside the bootstrap table:

```
students = [
    {"roll": 101, "name": "Raj", "address": "amritsar", "marks": 700},
    {"roll": 102, "name": "Simran", "address": "mohali", "marks": 720},
    {"roll": 103, "name": "Rahul", "address": "", "chandigarh": 800},
    {"roll": 104, "name": "Ankit", "address": "jalandhar", "marks": 750}
]
```

Solution:

Step1: Define the following view function inside the views.py file of **StudentApp**

```
def student_list_view(request):
    students = [
     {"roll": 101, "name": "Raj", "address": "Amritsar", "marks": 700},
     {"roll": 102, "name": "Simran", "address": "Mohali", "marks": 720},
    {"roll": 103, "name": "Rahul", "address": "Chandigarh", "marks": 800},
     {"roll": 104, "name": "Ankit", "address": "Jalandhar", "marks": 750}
    ]
    return render(request, "students.html", {"students": students})
```

Step2: Create a **students.html** file inside the **StudentApp/templates** folder.

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.
css" rel="stylesheet"
integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6h
W+ALEwIH" crossorigin="anonymous">
</head>
<body>
    <h1 class="text-center">All Student Details</h1>
    <table class="table">
        <tr>
            <th>Roll</th>
            <th>Name</th>
            <th>Address</th>
            <th>Marks</th>
        </tr>
        {% for student in students %}
        <tr>
            <td>{{ student.roll }}</td>
            <td>{{ student.name }}</td>
            <td>{{ student.address }}</td>
            <td>{{ student.marks }}</td>
        </tr>
```

```
        {% endfor %}
    </table>
</body>
</html>
```

Step3: Map the **student_list_view** function inside the **StudentApp/ursl.py** file:

```
path('allstudents/', views.student_list_view)
```

Step4: Create the following link inside the index.html file:

```
<a href="./allstudents">Get All Student Details </a>
```

Working with Dynamic URL:

Note: to generate the dynamic url instead of hardcoding it inside the html file, (similar to url_for in flask) here we need to use the following approach:

1. specify a name to the url mapping. inside the urls.py file.

```
path("dashboard/", views.dashboard_view, name="dash"),
```

2. Use this name inside the **index.html** file inside the template tag.

```
<a href="{% url 'dash' %}" class="btn btn-primary">Go To Dashboard</a>
```

Example passing the dynamic value using the url:

1. Define a view function inside the views.py file:

```
from django.http import HttpResponse

def student_detail_view(request, roll):
```

```
    return HttpResponse(f"Student Roll is: {roll}")
```

2. Specify the url mapping inside the urls.py at application level with a name:

```
from django.urls import path

from . import views

urlpatterns = [

    path("", views.home_view),

    path("dashboard/", views.dashboard_view, name="dash"),

    path('getstudent/<int:roll>/', views.student_detail_view,
name='student_detail')

]
```

3. Specify the link of the above **student_detail** inside the **index.html** file.

```
<a href="{% url 'student_detail' roll=10 %}" class="btn
btn-success">View Student</a>
```

Optional: Managing the templates for all the applications at Project level:

- We can also manage the templates and static files for all the applications of a project at project level. In this case we need to create the **templates** folder and **static** folder at project level and specify their locations inside the settings.py file explicitly.

Example:

```
# Define TEMPLATE_DIR dynamically

TEMPLATE_DIR = BASE_DIR / "templates"

TEMPLATES = [
```

```
...
'DIRS': [TEMPLATE_DIR], # Use the dynamically generated path
...
]
```

And to keep all the app specific html files by creating app specific folders inside the **templates** folder at project level.

UniversityProject2/

|--manage.py

|--**templates/**

| |--**StudentApp/**

| |--index.html

|--**StudentApp/**

- Inside the **StudentApp/views.py**, define a view function to render index.html:

```
from django.shortcuts import render
def home_view(request):
    return render(request, "StudentApp/index.html")
```

Optional: Managing the static files for all the applications at Project level:

- Similar to the templates, we can manage the static files for all the applications at Project level also.

Steps to include the static files inside the html templates:

Step1: Create a folder named with "**static**" inside the main project folder(**UniversityProject2**) folder. same as templates folder at project level.

Step2: Create the folder with the application name(here **StudentApp**) inside the **static** folder to keep all the application specific static contents.

Step3: Inside the **static/StudentApp/** folder create the "**images**" folder to keep all the images, and the "**css**" folder to keep all the external css files.

Step4: Add the **static** folder path to the **settings.py** file so that Django can be aware of our static contents.

```
STATIC_DIR = BASE_DIR / 'static'

STATICFILES_DIRS = [

    STATIC_DIR,

]
```

OR we can use:

```
import os

STATICFILES_DIRS = [

    os.path.join(BASE_DIR, 'static')

]
```

Step5: Using Static Files in Templates

(a) Load Static Files in Templates

- At the beginning of your HTML file, load the `static` tag:

  <!DOCTYPE html>

  {% load static %}

(b) Referencing an Image File

```
        <img src="{% static 'StudentApp/images/sh1.jpg' %}" alt="oops
image not found" width="200px" height="200px">
```

(c) Linking an External CSS File

```
<link rel="stylesheet" href="{% static 'StudentApp/css/a1.css' %}">
```

index.html:

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Wish Page</title>
    <link rel="stylesheet" href="{% static 'StudentApp/css/a1.css' %}">
</head>
<body>
    <h1>Welcome to Student Home Page</h1>
    <h3>The City is: {{citydata}}</h3>
    <img src="{% static 'StudentApp/images/sh1.jpg' %}" alt="oops image
not found" width="200px" height="200px">


</body>
</html>
```

a1.css:

```
body{
    background-color: aquamarine;
}
```

Now access the application using:

http://127.0.0.1:8000/student/

Limitation for the managing templates and static files at project level:

- App Independence is Lost – If multiple apps rely on the same template, it may cause conflicts.
- Harder to Maintain – Large projects require careful organization.
- Namespace Conflicts – If multiple apps have files with the same name, Django may get confused.

Template Inheritance in Django:

- Template inheritance allows the creation of a **base template** that can be extended by other templates.
- It is similar to the Template inheritance concept in Flask.

Step 1: Create a Base Template (base.html)

base.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title_block %} {% endblock %}</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.
css" rel="stylesheet"
integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6h
W+ALEwIH" crossorigin="anonymous">

    <style>
        body {
            margin: 0;
            padding: 0;
        }
        header {
            background-color: aquamarine;
            height: 10vh;
        }
        main {
            background-color: beige;
            height: 80vh;
        }
```

```
        footer {
            background-color: aqua;
            height: 10vh;
        }
    </style>
</head>
<body>
    <header>
   <h1 class="text-center">Welcome to the Chitkara University</h1>
    </header>
    <main>
        {% block main_block %}
        {% endblock %}
    </main>
    <footer>
        <p class="text-center">&copy; This is the Footer Section</p>
    </footer>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle
.min.js"
integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN
7N6jIeHz"
        crossorigin="anonymous"></script>
</body>
</html>
```

Step 2: Extend the Base Template

Modify `index.html`:

```
{% extends "base.html" %}

{% load static %}

{% block title_block %} Home Page {% endblock %}

{% block main_block %}

<h2 class="text-center">Welcome to Student Home Page</h2>

<h4>The Date and Time is: {{ date }}</h4>

<h4>University Name: {{ name }}</h4>

<h4>Batch: {{ batch }}</h4>

<h4>Year: {{ year }}</h4>

<img src="{% static 'images/sh1.jpg' %}" alt="oops image not found"
width="200px" height="200px">

<ul>

    <li><a href="{% url 'dash' %}">Go To Dashboard</a></li>

    <li><a href="./allstudents">Get All Student Details </a></li>

    <li><a href="{% url 'student_detail' roll=10 %}">View
Student</a></li>

</ul>

{% endblock %}
```

Modify `dashboard.html`:

```
{% extends "base.html" %}

{% block title_block %} Dashboard {% endblock %}
```

```
    {% block main_block %}
        <h1 class="text-center">Welcome to Dashboard</h1>
    {% endblock %}/
```

Modify the students.html:

```
{% extends "base.html" %}
{% block title_block %} All Student Page {% endblock %}
{% block main_block %}
<h1 class="text-center">All Student Details</h1>
<table class="table">
    <tr>
        <th>Roll</th>
        <th>Name</th>
        <th>Address</th>
        <th>Marks</th>
    </tr>
    {% for student in students %}
    <tr>
        <td>{{ student.roll }}</td>
        <td>{{ student.name }}</td>
        <td>{{ student.address }}</td>
        <td>{{ student.marks }}</td>
    </tr>
```

```
    {% endfor %}
</table>
{% endblock %}
```