

## Module – 5 : Python DB and Framework

### 1. HTML in Python

**Q.1** Introduction to embedding HTML within Python using web frameworks like Django or Flask.

**Ans.** Web frameworks like **Django** and **Flask** are essential tools for building modern web applications. These frameworks facilitate the creation of dynamic web pages by embedding HTML within Python. This process is achieved through the use of template engines, which bridge the gap between Python code and HTML.

#### 1. Purpose of Embedding HTML in Python:

- HTML forms the structure of web pages, while Python handles the logic and data processing.
- Embedding HTML within Python ensures dynamic content delivery, where the content of a webpage can change based on user interaction or backend data updates.

#### 2. Web Frameworks and Template Engines:

- Django and Flask are popular Python web frameworks that provide tools for embedding HTML in a structured and efficient way.
- Both frameworks utilize template engines (Django's built-in engine or Flask's Jinja2) to separate backend logic from the front-end layout.
- Templates allow placeholders for dynamic data, control structures like loops and conditionals, and inclusion of reusable components.

#### 3. Advantages of Using Frameworks:

- Modularity: Templates help in separating Python code from HTML promoting better organization and maintainability.
- Reusability: Templates can inherit from a base layout, allowing consistent design across multiple pages.
- Dynamic Content: Data fetched or computed in Python can be passed to templates, enabling dynamic updates to web pages.

#### 4. How It Works:

- The backend logic in Python processes user requests, fetches data, and determines the content to be displayed.
- The framework passes this data to the template engine, which embeds it into predefined HTML structures.
- The final rendered HTML is sent to the user's browser.

### Django and Flask: A Comparison

#### Django:

- Django follows the Model-View-Template (MVT) architectural pattern.
- It includes a built-in template engine for rendering HTML files.
- Key features:
  - Built-in ORM (Object-Relational Mapping) to interact with databases.
  - Tight integration between views and templates for ease of use.

### **Flask:**

- Flask is more lightweight and flexible, following a Model-View-Controller (MVC)-like structure.
  - It uses the Jinja2 template engine, which offers advanced features like macros and template inheritance.
  - Flask is minimalistic, allowing more customization.
- 

## **2. CSS in Python**

### **Q.2 How to serve static files (like CSS, JavaScript) in Django**

**Ans.** Static files are files that don't change dynamically, such as CSS, JavaScript, and image files. Django provides built-in support to manage static files efficiently using the `django.contrib.staticfiles` app.

#### **➤ Configuring STATIC\_URL and STATICFILES\_DIRS**

In the `settings.py` file, configure the following settings to define how Django should locate and serve static files:

- **STATIC\_URL:** This defines the URL prefix for accessing static files.

Example:

```
STATIC_URL = '/static/'
```

- **STATICFILES\_DIRS:** This setting tells Django where to look for additional static files in your project directory structure (other than the default `app_name/static` directories).

```
STATICFILES_DIRS = [
    BASE_DIR / "static",
]
```

#### **➤ Organizing Static Files**

Django uses a specific directory structure to locate static files:

- Place app-specific static files in a static directory inside each app.
- For example:

```
myapp/  
  static/  
    myapp/  
      style.css
```

- For project-wide static files, use a global static directory (as configured in `STATICFILES_DIRS`).

### ➤ Using the `collectstatic` Command

In a production environment, Django collects all static files into a single location specified by the `STATIC_ROOT` setting. This is achieved using the `collectstatic` management command.

- Configure `STATIC_ROOT` in `settings.py`:

```
STATIC_ROOT = BASE_DIR / "staticfiles"
```

- Run the command:

```
python manage.py collectstatic
```

This command consolidates all static files into the `STATIC_ROOT` directory for efficient serving by a web server.

### ➤ Serving Static Files

- Development: During development, Django automatically serves static files when `DEBUG` is set to `True`. The `runserver` command takes care of this, so you don't need additional configuration.
- Production: In production, Django doesn't serve static files. Use a dedicated web server like Nginx or Apache to serve files from the `STATIC_ROOT` directory. These servers are optimized for handling static file requests.

### ➤ Referencing Static Files in Templates

Use the `{% static %}` template tag to reference static files in templates.

Example:

```
<link rel="stylesheet" href="{% static 'myapp/style.css' %}">  
<script src="{% static 'myapp/script.js' %}"></script>
```

### ➤ **Enabling the Static Files App**

Ensure that `django.contrib.staticfiles` is included in the `INSTALLED_APPS` list in `settings.py`. This app provides tools for managing static files.

#### Summary

- Configure `STATIC_URL`, `STATICFILES_DIRS`, and `STATIC_ROOT` in `settings.py`.
- Organize files in static directories within apps or globally.
- Use `{% static %}` in templates to load static files.
- Use the `collectstatic` command to gather files for production.
- Serve static files with Django during development and with a web server in production.

By following this approach, you can effectively manage and serve static files in a Django project.

## **3. JavaScript with Python**

### **Q.3** Using JavaScript for client-side interactivity in Django templates.

**Ans.** Using JavaScript for client-side interactivity in Django templates is a common practice to enhance the user experience by adding dynamic behavior to web pages.

#### **1. Embedding JavaScript in Django Templates**

- You can include JavaScript directly in your Django templates using `<script>` tags.
- Ensure the JavaScript code interacts with HTML elements rendered by the Django template.

#### **2. Using Static Files for JavaScript**

For better organization, it's recommended to store JavaScript code in separate files and serve them using Django's static file system.

- Create a `static/js` directory in your app and add your JavaScript files there.
- Load the static file in the template using the `{% load static %}` tag.

#### **3. Interacting with Django Context**

You can pass data from Django views to JavaScript via template context variables. Use Django's template tags and filters to render variables directly into your JavaScript.

#### 4. Using AJAX for Client-Server Communication

You can use JavaScript libraries like Fetch API or jQuery to send asynchronous requests to Django views (e.g., for fetching or submitting data).

### Q.4 Linking external or internal JavaScript files in Django.

**Ans.** Django, linking JavaScript files—whether external or internal—requires understanding how Django handles static files and template rendering.

#### 1. Set Up Static Files

Ensure your Django project is configured to serve static files:

- In your settings.py file:

```
STATIC_URL = '/static/'  
STATICFILES_DIRS = [BASE_DIR / "static"]
```

- Create a static directory in your project root or app directories. For example:

```
project_root/  
  static/  
    js/  
      script.js
```

#### 2. Linking Internal JavaScript Files

To link JavaScript files stored in your static directory:

##### 1. Load Static Files in Templates

At the top of your template, load the static template tag:

```
{% load static %}
```

##### 2. Include the JavaScript File

Use the static tag to reference your file:

```
<script src="{% static 'js/script.js' %}"></script>
```

### 3. Linking External JavaScript Files

To include an external JavaScript file, directly add the `<script>` tag with the file's URL:

```
<script src="https://cdn.example.com/library.js"></script>
```

### 4. Using JavaScript in a Django Template

Here's an example of a full Django template:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Django JavaScript Example</title>
  {% load static %}
  <script src="{% static 'js/script.js' %}"></script>
</head>
<body>
  <h1>Hello, Django!</h1>
  <button id="myButton">Click Me</button>
</body>
</html>
```

### 5. Collect Static Files for Deployment

When deploying, use Django's `collectstatic` management command to gather all static files into the directory defined by `STATIC_ROOT`.

```
python manage.py collectstatic
```

Ensure your web server (e.g., Nginx) is configured to serve files from the `STATIC_ROOT` directory.

- ☐ Use the `{% block %}` and `{% extends %}` tags to manage JavaScript in a base template for consistency across pages.
- ☐ For dynamically loaded JavaScript, consider using Django's `JsonResponse` and `AJAX`.

## 4. Django Introduction

### Q.5 Overview of Django: Web development framework.

**Ans.** Django is a high-level Python web framework designed for rapid development and clean, pragmatic design. It emphasizes reusability, modularity, and scalability, allowing developers to build robust and efficient web applications quickly.

#### ➤ Features of Django

##### 1. Ease of Use

- Django simplifies web development by providing a clear and concise framework for developers.
- Built-in tools like an admin interface and ORM reduce the need for manual coding.

##### 2. Batteries Included

- Comes with many built-in features such as authentication, URL routing, templates, and form handling.
- Provides libraries for common web development tasks, such as content management and user authentication.

##### 3. MVC/MVT Architecture

- Django follows the **Model-View-Template (MVT)** architectural pattern:
  - **Model:** Handles data and database structure.
  - **View:** Manages business logic and interacts with models.
  - **Template:** Deals with the presentation layer.

##### 4. ORM (Object-Relational Mapping)

- Allows developers to interact with databases using Python objects instead of writing raw SQL.
- Supports multiple databases like PostgreSQL, MySQL, SQLite, and Oracle.

##### 5. Security

- Offers robust protection against common vulnerabilities, including SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), and clickjacking.
- Includes built-in mechanisms for managing user authentication securely.

##### 6. Scalability

- Designed to handle traffic-heavy websites and complex data requirements.
- Powers several high-profile, large-scale sites like Instagram and Pinterest.

##### 7. Rapid Development

- Django automates repetitive tasks, enabling developers to focus on app logic.
- A rich ecosystem of reusable apps speeds up development.

##### 8. Extensibility

- Supports adding third-party packages for additional functionality.
- Modular design allows for easy customization and extension.

##### 9. Community and Documentation

- Django has a large, active community and excellent documentation.
- Regular updates ensure compatibility with modern web technologies.

## Core Components

### 1. Django Apps

- Django projects are composed of modular apps, each handling a specific functionality.
- Apps can be reused across different projects.

### 2. Middleware

- Components that process requests and responses globally, such as session management and authentication.

### 3. Admin Interface

- Automatically generated admin panel for managing application data.

### 4. Django Templates

- A simple yet powerful templating engine for dynamic HTML generation.

### 5. Django REST Framework (DRF)

- Extends Django for building RESTful APIs.
- 

## Pros of Django

- Quick development cycles.
- Highly secure.
- Encourages clean and maintainable code.
- Scales well for growing applications.

## Cons of Django

- Can be monolithic for smaller projects.
  - May require some effort to optimize for very high-performance needs.
- 

## Common Use Cases

- Content management systems (CMS)
- E-commerce websites
- Social networking sites
- RESTful APIs and backend services
- Data-driven dashboards and web apps

## Q.6 Advantages of Django (e.g., scalability, security).

Ans.

### 1. Scalability

- **Efficient Handling of Traffic:** Django is designed to handle high traffic volumes and supports features like caching, database optimization, and load balancing.



- **Modular Architecture:** The framework's modularity allows developers to scale applications horizontally (by adding more servers) or vertically (by upgrading resources).
- **Proven at Scale:** Used by high-traffic websites such as Instagram, Spotify, and YouTube.

## 2. Security

- **Built-in Protections:** Django includes tools to protect against common vulnerabilities like:
  - **SQL Injection**
  - **Cross-Site Scripting (XSS)**
  - **Cross-Site Request Forgery (CSRF)**
  - **Clickjacking**
- **Secure Authentication:** Provides a robust user authentication and session management system.
- **Regular Updates:** Actively maintained with frequent security patches.

## 3. Rapid Development

- **Out-of-the-Box Features:** Prebuilt components like the admin interface, authentication system, and ORM reduce development time.
- **Reusable Apps:** Modular structure allows developers to reuse apps across multiple projects, speeding up the process.

## 4. Versatility

- Suitable for a wide range of projects, including:
  - Content management systems (CMS)
  - E-commerce platforms
  - Social networking sites
  - RESTful APIs
  - Enterprise web applications
- **Multi-Platform Support:** Compatible with different operating systems and databases.

## 5. Ease of Use

- **Python-Based:** Leverages Python's simplicity and readability.
- **Comprehensive Documentation:** Detailed documentation and tutorials make it easy for developers of all levels.
- **Admin Interface:** Auto-generated admin panel for quick data management.

## 6. Community and Ecosystem

- **Active Community:** Large and engaged developer community offers support and contributes plugins.
- **Third-Party Libraries:** A rich ecosystem of packages extends Django's functionality, from payment processing to social authentication.

## 7. Performance

- **ORM Optimization:** Manages database queries efficiently, minimizing performance bottlenecks.
- **Built-In Caching:** Supports caching frameworks like Memcached and Redis for faster response times.

## 8. Flexibility

- **Customizability:** Highly customizable architecture to fit unique business requirements.
- **Middleware:** Offers middleware support to modify request/response cycles globally.

## 9. Maintainability

- **Clean Code Structure:** Encourages the use of reusable code and design patterns.
- **Version Compatibility:** Ensures smooth upgrades through its backward-compatible design philosophy.

## 10. Internationalization and Localization

- **Multi-Language Support:** Built-in features to create applications that support multiple languages and time zones.

## 11. Cost-Effectiveness

- **Open Source:** Free to use, with no licensing costs.
- **Time Savings:** Rapid development features save significant time and reduce overall project costs.

## 12. SEO-Friendly

- **Human-Readable URLs:** Encourages the use of SEO-friendly URL patterns.
- **Dynamic Content:** Enables easy optimization of dynamic content for search engines.

Django's combination of scalability, security, rapid development, and a vibrant ecosystem makes it one of the most robust frameworks for modern web development.

## Q.7 Django vs. Flask comparison: Which to choose and why.

**Ans.** Django and Flask are two of the most popular Python web frameworks. While both are powerful and widely used, they cater to different needs and development philosophies.

Attributes	Django	Flask
Type of Framework	Django is a full-stack web framework that enables ready to use solutions with its batteries-included approach.	Flask is a lightweight framework that gives abundant features without features without external libraries and minimalist features.
Architecture	It follows an MVT architecture, which is split up into three parts, model, view, and template.	No strict architecture; developers choose their structure.
Project Layout	Django is suitable for multiple page applications.	Flask is suitable for only single-page applications.
Maturity	Launched in 2005, it is a very mature framework with extensive community support.	Launched in 2010, Flask is a younger framework but has a large community.
Database Support	Django supports the most popular relational database management systems like MySQL, Oracle etc.  Includes ORM for database handling.	Flask does not support the basic database management system and uses SQL Alchemy for database requirements.

### ➤ When to Choose Django

- You need a **quick start** for a large, complex application.
- You require **built-in tools** like ORM, admin panels, or authentication.
- Your project has a defined **architecture and structure** from the beginning.
- Scalability and **traffic-heavy applications** are a priority.

**Examples:** E-commerce platforms, social media sites, content management systems (CMS).

### ➤ When to Choose Flask

- You need a **lightweight framework** for a small-to-medium project.
- Flexibility is more important than built-in features.
- You want to customize every component of your application.
- **Micro services** or APIs are the primary focus.

**Examples:** REST APIs, lightweight web apps, quick prototypes, IoT apps.

- Choosing between Django and Flask depends on your project requirements, team expertise, and development philosophy.

## 5. Virtual Environment

**Q.8** Understanding the importance of a virtual environment in Python projects.

**Ans.** A virtual environment in Python is a self-contained directory that contains all the necessary files to run a specific Python project.

### Importance of Virtual Environments:

1. **Dependency Management:**
  - Different projects may require different versions of the same library. A virtual environment allows each project to maintain its own dependencies without conflicts.
2. **Avoiding System-wide Changes:**
  - Installing packages globally can lead to version clashes and unintended updates, potentially breaking other projects. Virtual environments prevent this by keeping dependencies isolated.
3. **Reproducibility:**
  - Using virtual environments ensures that the project behaves consistently across different machines or setups, as the same versions of dependencies can be installed.
4. **Ease of Collaboration:**
  - By sharing a `requirements.txt` file (a list of dependencies and their versions), collaborators can replicate the exact environment needed to run the project.
5. **Environment Isolation:**
  - Different projects can use different Python versions or configurations, which is especially useful when working with legacy systems or newer Python features.

## 6. Cleaner Development Workflow:

- Keeps the global Python environment clean and uncluttered, making it easier to manage and debug individual projects.

## Q.9 Using venv or virtualenv to create isolated environments.

**Ans.** Using venv or virtualenv creates isolated Python environments, preventing dependency conflicts between projects.

- To create a virtual environment with venv:  
`python -m venv myenv`

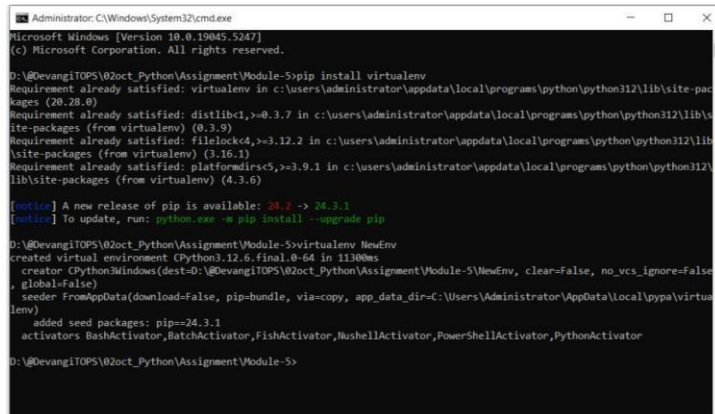
Activate it:

Windows: `myenv\Scripts\activate`

- virtualenv works similarly but supports older Python versions and offers extra features.
- Virtual environments improve project management, enhance security, and ensure consistent dependencies.

## Lab:

- Set up a virtual environment for a Django project.



```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.5247]
(c) Microsoft Corporation. All rights reserved.

D:\@DevangiTOPS\02oct_Python\Assignment\Module-5>pip install virtualenv
Requirement already satisfied: virtualenv in c:\users\administrator\appdata\local\programs\python\python312\lib\site-packages (20.28.0)
Requirement already satisfied: distlib<1, >=0.3.7 in c:\users\administrator\appdata\local\programs\python\python312\lib\site-packages (from virtualenv) (0.3.9)
Requirement already satisfied: filelock4, >=3.12.2 in c:\users\administrator\appdata\local\programs\python\python312\lib\site-packages (from virtualenv) (3.16.1)
Requirement already satisfied: platformdirs<5, >=3.9.1 in c:\users\administrator\appdata\local\programs\python\python312\lib\site-packages (from virtualenv) (4.3.6)

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip

D:\@DevangiTOPS\02oct_Python\Assignment\Module-5>virtualenv NewEnv
created virtual environment CPython3.12.6.final.0-64 in 11300ms
  creator CPython3Windows(dest=D:\@DevangiTOPS\02oct_Python\Assignment\Module-5\NewEnv, clear=False, no_vcs_ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, via=copy, app_data_dir=C:\Users\Administrator\AppData\Local\pypa\virtualenv)
    added seed packages: pip==24.3.1
    activators BashActivator, BatchActivator, FishActivator, NushellActivator, PowerShellActivator, PythonActivator

D:\@DevangiTOPS\02oct_Python\Assignment\Module-5>
```

## Practical Example:

5) Write a Python program to create and activate a virtual environment, then install Django in it.

```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.5247]
(c) Microsoft Corporation. All rights reserved.

D:\DevangiTOPS\02oct_Python\Assignment\Module-5>pip install virtualenv
Requirement already satisfied: virtualenv in c:\users\administrator\appdata\local\programs\python\python312\lib\site-packages (20.28.0)
Requirement already satisfied: distlib<1,>=0.3.7 in c:\users\administrator\appdata\local\programs\python\python312\lib\site-packages (from virtualenv) (0.3.9)
Requirement already satisfied: filelock4,>=3.12.2 in c:\users\administrator\appdata\local\programs\python\python312\lib\site-packages (from virtualenv) (3.16.1)
Requirement already satisfied: platformdirs<5,>=3.9.1 in c:\users\administrator\appdata\local\programs\python\python312\lib\site-packages (from virtualenv) (4.3.6)

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip

D:\DevangiTOPS\02oct_Python\Assignment\Module-5>virtualenv NewEnv
created virtual environment CPython3.12.6.final.0-64 in 1130ms
creator CPythonWindows(dest=D:\DevangiTOPS\02oct_Python\Assignment\Module-5\NewEnv, clear=False, no_vcs_ignore=False, global=False)
seeder FromAppData(download=False, pip-bundle, via=copy, app_data_dir=C:\Users\Administrator\AppData\Local\pip\Cache\virtualenv)
added seed packages: pip==24.3.1
activators BashActivator,BatchActivator,FishActivator,NushellActivator,PowerShellActivator,PythonActivator

D:\DevangiTOPS\02oct_Python\Assignment\Module-5>
```

```
Administrator: C:\Windows\System32\cmd.exe

D:\DevangiTOPS\02oct_Python\Assignment\Module-5>NewEnv\Scripts\activate
(NewEnv) D:\DevangiTOPS\02oct_Python\Assignment\Module-5>pip install django==4.1
Collecting django==4.1
  Using cached Django-4.1-py3-none-any.whl.metadata (4.0 kB)
Collecting asgiref<4,>=3.5.2 (from django==4.1)
  Using cached asgiref-3.8.1-py3-none-any.whl.metadata (9.3 kB)
Collecting sqlparse<0.2.2 (from django==4.1)
  Using cached sqlparse-0.5.3-py3-none-any.whl.metadata (3.9 kB)
Collecting tzdata (from django==4.1)
  Using cached tzdata-2024.2-py2.py3-none-any.whl.metadata (1.4 kB)
Using cached Django-4.1-py3-none-any.whl (8.1 MB)
Using cached asgiref-3.8.1-py3-none-any.whl (23 kB)
Using cached sqlparse-0.5.3-py3-none-any.whl (44 kB)
Using cached tzdata-2024.2-py2.py3-none-any.whl (346 kB)
Installing collected packages: tzdata, sqlparse, asgiref, django
Successfully installed asgiref-3.8.1 django-4.1 sqlparse-0.5.3 tzdata-2024.2

(NewEnv) D:\DevangiTOPS\02oct_Python\Assignment\Module-5>
```

## 6. Project and App Creation

**Q.10** Steps to create a Django project and individual apps within the project.

Ans.

1. Install Django: Ensure you have Python installed, then install Django using pip:  
- pip install django
2. Create a Django project: Run the following command to start a new Django project:  
- django-admin startproject project\_name

3. Navigate into the project directory:  
`cd project_name`
4. Create a Django app: Run the following command inside the project directory to create an app:  
`python manage.py startapp app_name`
5. Register the app in settings: Open `settings.py` and add the app name to the `INSTALLED_APPS` list.
6. Define models: In `models.py` of the app, create your database models.
7. Apply migrations: Run  
`python manage.py makemigrations app_name`  
`python manage.py migrate`
8. Create a superuser (optional, for admin access):  
`python manage.py createsuperuser`
9. Configure URLs:
  - In `project_name/urls.py`, include the app's URLs.
  - In `app_name`, create a `urls.py` file and define URL patterns.
10. Run the development server:  
`python manage.py runserver`
11. Start development by defining views, templates, and static files as needed.

## Q.11 Understanding the role of `manage.py`, `urls.py`, and `views.py`.

### Ans.

1. **manage.py:** A command-line tool to manage the Django project, like running the server, applying migrations, and creating superusers.
2. **urls.py:** Maps URLs to views. The project-level file routes to apps, and app-level files handle specific routes for the app.
3. **views.py:** Contains logic to process user requests and return responses, like rendering templates or displaying data.

## 7. MVT Pattern Architecture

**Q.12** Django's MVT (Model-View-Template) architecture and how it handles request-response cycles.

**Ans.**

### **1. MVT Architecture**

- Model: Manages the database and data structure.
- View: Contains logic to handle requests and interact with the model.
- Template: Displays data to the user in a web page format.

### **2. Request-Response Cycle**

- The user sends a request via a URL.
- `urls.py` maps the URL to the corresponding view.
- The view processes the request, interacts with the model, and prepares a response.
- If required, the response is passed to a template for rendering.
- The server sends the final response (HTML, JSON, etc.) back to the user.

This architecture ensures a clean separation of data, logic, and presentation.

## **8. Django Admin Panel**

**Q.13** Introduction to Django's built-in admin panel.

**Ans.**

#### **1. Introduction:**

The admin panel is a web-based interface for managing database records. It allows CRUD operations without writing code.

#### **2. Features:**



- Manage users, roles, and permissions.
- View, add, edit, and delete data.
- Secure and customizable interface.

### 3. Usage:

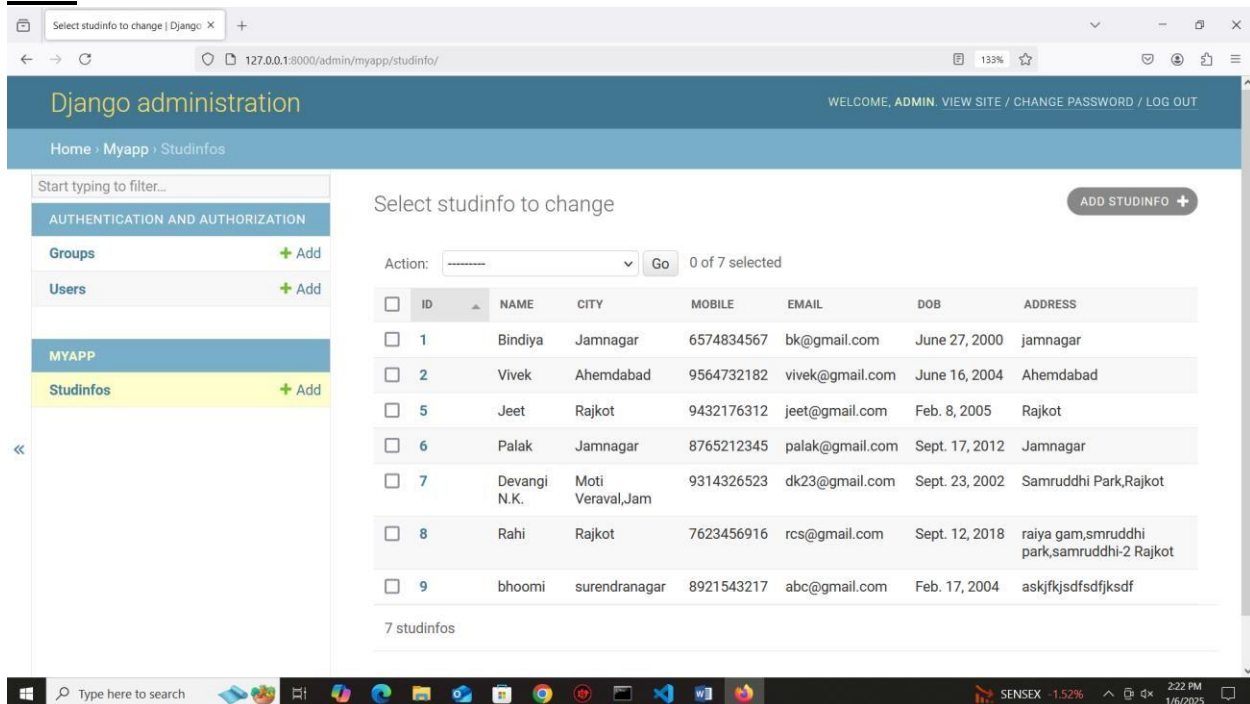
- Create a superuser with `python manage.py createsuperuser`.
- Register models in `admin.py` using `admin.site.register(ModelName)`.
- Access the admin panel at `http://127.0.0.1:8000/admin/`.

### 4. Benefits:

Saves time, simplifies database management, and is extensible for advanced use cases.

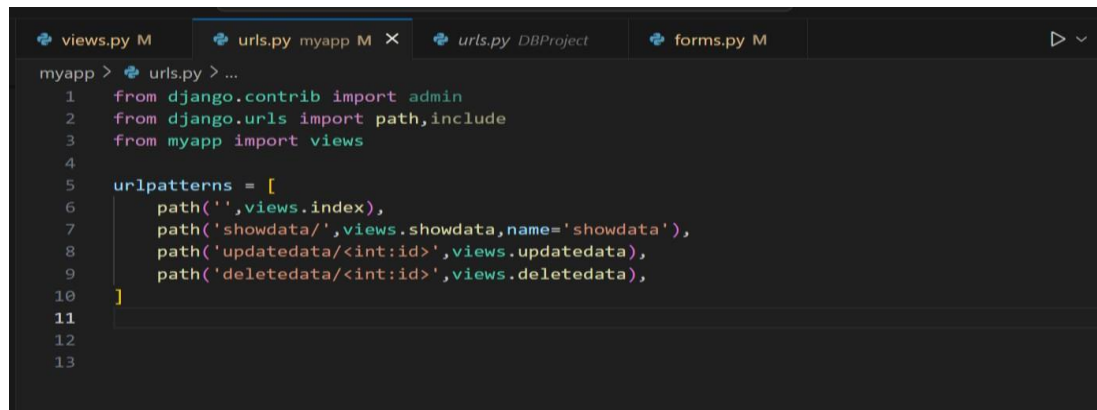
## Q.14 Customizing the Django admin interface to manage database records.

**Ans.**



## 9. URL Patterns and Template Integration

**Q.15** Setting up URL patterns in urls.py for routing requests to views.



```
views.py M  urls.py myapp M X  urls.py DBProject  forms.py M  ▶ ▼
myapp > urls.py > ...
1  from django.contrib import admin
2  from django.urls import path,include
3  from myapp import views
4
5  urlpatterns = [
6      path('',views.index),
7      path('showdata/',views.showdata,name='showdata'),
8      path('updatedata/<int:id>',views.updatedata),
9      path('deletedata/<int:id>',views.deletedata),
10 ]
11
12
13
```

**Q.16** Integrating templates with views to render dynamic HTML content.

```
html U <> appointment.html U <> contact.html U <> feature.html U urls.py drpro U <> about.ht
drapp > views.py > about
1 from django.shortcuts import render
2
3 # Create your views here.
4 def index(request):
5     return render(request, 'index.html')
6
7 def about(request):
8     return render(request, 'about.html')
9
10 def appointment(request):
11     return render(request, 'appointment.html')
12
13 def contact(request):
14     return render(request, 'contact.html')
15
16 def feature(request):
17     return render(request, 'feature.html')
18
19 def service(request):
20     return render(request, 'service.html')
21
22 def team(request):
23     return render(request, 'team.html')
24
25 def testimonial(request):
26     return render(request, 'testimonial.html')
27
28
```

## 10. Form Validation using JavaScript

**Q.17** Using JavaScript for front-end form validation.

Ans.

Front-end form validation is an essential part of web development to ensure users submit valid and correctly formatted data before it is sent to a server. JavaScript provides several methods to validate forms on the client side, improving user experience by providing instant feedback and reducing unnecessary server requests.

### ➤ Why Use JavaScript for Form Validation?

1. **Improved User Experience:** Immediate feedback helps users correct errors before submission.
2. **Reduced Server Load:** Errors are caught before sending data to the server.
3. **Prevention of Basic Security Issues:** Helps prevent malformed input but should not be relied on for security.
4. **Customization & Flexibility:** Can implement complex validation rules dynamically.

### ➤ Common Validation Types in JavaScript

1. **Required Fields:** Ensure a field is not empty.
2. **Email Validation:** Check if input matches an email pattern.
3. **Password Strength Validation:** Ensure passwords meet criteria (length, special characters, numbers).
4. **Numeric Validation:** Ensure input contains only numbers.
5. **Custom Regex Validation:** Validate formats like phone numbers or postal codes.

#### **Example:** Email Validation

```
function validateEmail(email) {  
    let emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;  
    return emailPattern.test(email);  
}  
  
console.log(validateEmail("test@example.com")); // true  
console.log(validateEmail("invalid-email")); // false
```

### ➤ JavaScript-Based Validation

JavaScript allows real-time validation using event listeners.

#### **Basic Example:** Validating Required Fields

```
<form id="myForm">  
    <input type="text" id="name" placeholder="Enter your name">  
    <span id="nameError" style="color: red;"></span>  
    <button type="submit">Submit</button>  
</form>  
  
<script>  
    document.getElementById("myForm").addEventListener("submit", function(event) {  
        let name = document.getElementById("name").value;  
        let nameError = document.getElementById("nameError");
```

```

    if (name.trim() === "") {
        nameError.textContent = "Name is required!";
        event.preventDefault(); // Prevent form submission
    } else {
        nameError.textContent = "";
    }
});
</script>

```

### ➤ Real-Time Validation

JavaScript can validate user input as they type using the `input` event.

#### Example:

```

document.getElementById("email").addEventListener("input", function() {
    let email = this.value;
    let errorSpan = document.getElementById("emailError");

    if (!validateEmail(email)) {
        errorSpan.textContent = "Invalid email format";
    } else {
        errorSpan.textContent = "";
    }
});

```

- Using JavaScript for front-end form validation helps improve user experience, reduce server load, and ensure data accuracy. However, front-end validation should always be complemented by secure back-end validation to prevent security vulnerabilities.

## 11. Django Database Connectivity (MySQL or SQLite)

**Q.18** Connecting Django to a database (SQLite or MySQL).

**Ans.** Connecting Django to a database like SQLite or MySQL involves configuring Django's settings and installing necessary dependencies. Below are the steps for both databases.

```
# Database
# https://docs.djangoproject.com/en/4.1/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'dr_db',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': '127.0.0.1',
        'PORT': '3306',
    }
}
```

**Q.19** Using the Django ORM for database queries.

```
urls.py dbproject  models.py ×  forms.py  <> index.html
dbapp > models.py > ...
1  from django.db import models
2
3  # Create your models here.
4  class stud_info(models.Model):
5      name=models.CharField(max_length=30)
6      city=models.CharField(max_length=30)
7      email=models.EmailField()
8      dob=models.DateField()
9      mobile=models.BigIntegerField()
10     address=models.TextField()
11
```

**Ans.**

## 12. ORM and QuerySets

**Q.20** Understanding Django's ORM and how QuerySets are used to interact with the database.

**Ans**

### **Fetch all data:**

```
products = Product.objects.all() # Get all products
for product in products:
    print(product.name) # Print the name of each product
```

### **Filter data:**

```
expensive_products = Product.objects.filter(price__gt=50000)
for product in expensive_products:
    print(product.name, product.price)
```

### **Get a single row:**

```
product = Product.objects.get(id=1)
print(product.name, product.price)
```

### **Add data:**

```
Product.objects.create(name="Smartphone", price=62000, description="A great phone.")
```

### **Update data:**

```
product = Product.objects.get(id=1)
product.price = 44000 # Change the price
product.save() # Save the changes
```

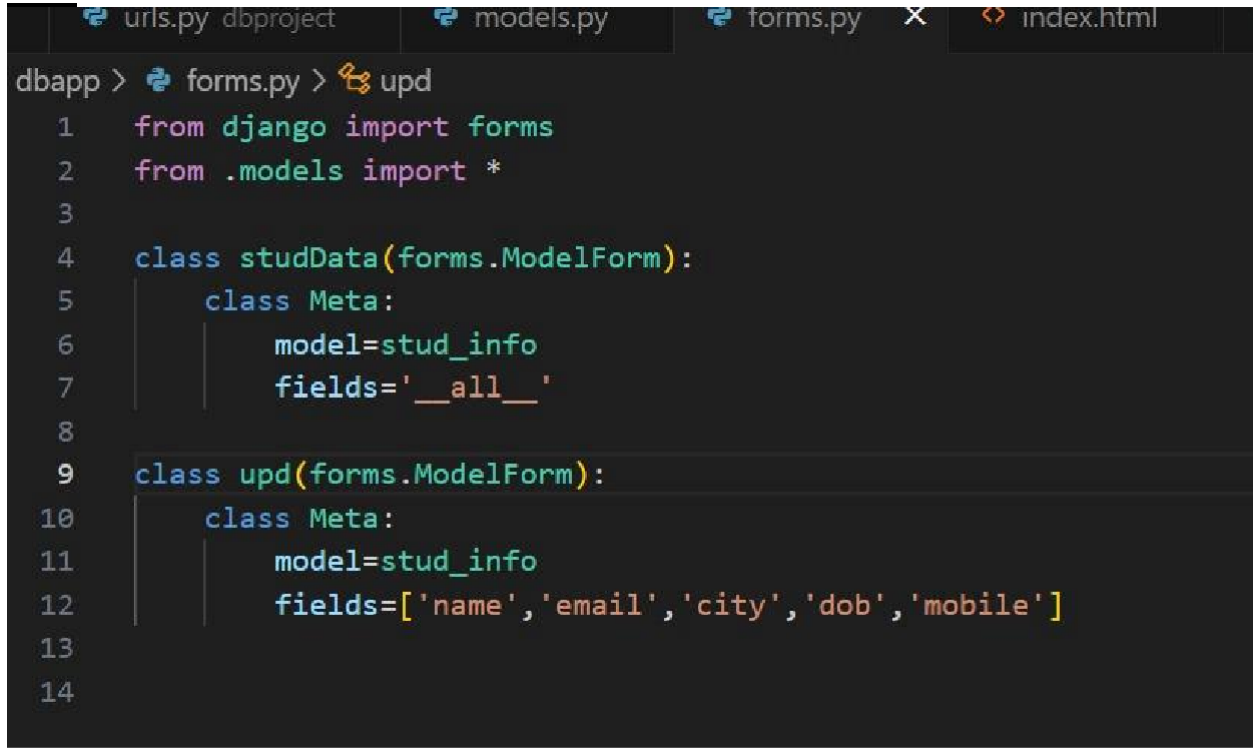
### **Delete data:**

```
product = Product.objects.get(id=1)
product.delete()
```

### 13. Django Forms and Authentication

**Q.21** Using Django's built-in form handling.

**Ans.**

A screenshot of a VS Code editor window with a dark theme. The top of the window shows several open files: 'urls.py', 'dbproject', 'models.py', 'forms.py' (which is the active file), and 'index.html'. The 'forms.py' file contains Python code for Django forms. The code starts with imports for 'forms' from 'django' and '\*' from '.models'. It then defines two classes: 'studData' and 'upd', both inheriting from 'forms.ModelForm'. Each class has a nested 'Meta' class. 'studData.Meta' sets 'model=stud\_info' and 'fields="\_\_all\_\_"'. 'upd.Meta' sets 'model=stud\_info' and 'fields' to a list of field names: ['name', 'email', 'city', 'dob', 'mobile']. Line numbers 1 through 14 are visible on the left side of the code editor.

```
dbapp > forms.py > upd
1  from django import forms
2  from .models import *
3
4  class studData(forms.ModelForm):
5      class Meta:
6          model=stud_info
7          fields='__all__'
8
9  class upd(forms.ModelForm):
10     class Meta:
11         model=stud_info
12         fields=['name', 'email', 'city', 'dob', 'mobile']
13
14
```

**Q.22** Implementing Django's authentication system (sign up, login, logout, password management).

**Ans.** Performed In Vscode editor.