Module – 4 : Python DB and Framework

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

- Django is a Python web framework used to build dynamic web applications.

- Instead of showing output in the terminal, Django sends data from views.py to HTML template files.

- HTML is kept inside the templates folder and Django renders it through the render() function.

  Ex :

  # views.py

```
from django.shortcuts import render

def home(request):
        return render(request, "index.html", {"name": "User"})
```

  #index.html

```
<h1> Hello {{ name }} </h1>
```

- {{ name }} displays the value passed from the view to the HTML template.
- Templates allow loops, conditions, and dynamic content.

- **Conclusion**:
  Embedding HTML in Django means connecting Python logic in views.py with HTML templates to create dynamic web pages.

2) Generating dynamic HTML content using Django templates.

- Django templates are used to display dynamic data in HTML pages.

- Data is sent from views.py to the template using the render() function.

- {{ }} is used to show variable values and {% %} is used for loops and conditions.

- **views.py**

```
def student_list(request):
   students = [“Amit”, “Riya”, “Vivek”]
   return render(request, “students.html”, {“students”: students})
```

- **students.html**

```
<ul>
{% for s in students %}
  <li>{{ s }}</li>
{% endfor %}
</ul>
```

- **Conclusion**:
  Django templates help create dynamic web pages by combining Python data with HTML.

3) Integrating CSS with Django templates.

- CSS is used to style the HTML pages in Django.

- CSS files are stored inside the static directory in a Django app.

- To use CSS in templates, we load static files using {% load static %} and link the CSS file path.

- **Folder Structure**

```
app/
  static/
    style.css
  templates/
    index.html
```

- **index.html**

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

```
<link rel="stylesheet" href="/static/style.css">

<h1 class="title">Welcome</h1>
```

- **style.css**

```
.title {

   color: blue;

   font-size: 30px;

}
```

- Static files must be configured in settings.py for Django to locate them.

- **Conclusion:**
  CSS can be easily connected to Django templates using the static folder and {% static %} tag to style web pages.

4) How to serve static files (like CSS, JavaScript) in Django.

- Static files include CSS, JavaScript, and images used for styling and client-side functionality.

- In Django, static files are stored inside a folder named static within the app.

- To use static files in templates, we load the static tag and reference the file using {% static %}.

- **Folder Structure**

```
app/
  static/
    style.css
    script.js
  templates/
    index.html
```

- **settings.py**

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

- **<u>index.html</u>**

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

- Django collects and serves these files automatically during development.

- **Conclusion:**
  Static files are placed inside the static directory and accessed in templates using the {% static %} tag, enabling Django pages to include CSS, JavaScript, and images.

5) Using JavaScript for client-side interactivity in Django templates.

- JavaScript runs in the browser and makes pages interactive without reloading.
- Place .js files in the app's static folder and load them with {% load static %} and {% static %}.

- **<u>index.html</u>**

```
{% load static %}

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


<button id="btn">Click me</button>

<p id="msg"></p>
```

- **<u>script.js</u>**

```
document.getElementById('btn').addEventListener('click', function() {

document.getElementById('msg').textContent = 'Hello from JavaScript!';

});
```

- For dynamic data, use fetch() to call a Django view that returns JSON, then update the DOM.

- JavaScript improves user experience by handling input validation, UI updates, and asynchronous requests on the client side.

- **Conclusion:**
  Add JavaScript files in static, include them in templates, and use DOM methods or fetch() to make Django pages interactive.

6) Linking external or internal JavaScript files in Django.

- JavaScript can be added inside Django templates or linked as separate external files.

- **Internal JavaScript (written directly in template)**

```
<script>
  alert("Hello from internal JS");
</script>
```

- **External JavaScript file**
  1. Save the file in your app's static directory, example:

     myapp/static/js/main.js
  2. Load static in the template and link the file:

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

- Cleaner code

- Reusable across multiple templates

- Easier to maintain and update

7) Overview of Django: Web development framework.

- Django is a high-level Python web framework used to build secure and scalable web applications.

- It follows the MVC-based architecture known as MVT (Model View Template).

- Django includes built-in features such as an ORM for database handling, an admin panel, authentication, and URL routing, which reduces development time.

- Django encourages rapid development and clean, reusable code.

- It is widely used for applications like e-commerce sites, social networks, dashboards, and CMS platforms. Because of its security features like protection against SQL injection, XSS, and CSRF attacks, it is considered a reliable framework for production.

- **Key Advantages**

  • Fast development

  • Built-in security features

  • Powerful ORM system

  • Automatic admin interface

  • Scalable and flexible architecture

8) Advantages of Django (ex: scalability, security).

- Django provides several strong advantages that make it a popular choice for modern web development.

- **Scalability**

  Django can handle high traffic and large-scale applications. It is used by companies like Instagram and Pinterest because it supports load balancing, caching, and distributed systems.

- **Security**

  Django has built-in protection against common attacks such as SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), and clickjacking. It also provides a secure user authentication system.

- **Rapid Development**

  Django includes many ready-to-use components such as the admin panel, form handling, and authentication, which allows developers to build applications faster.

- **ORM Support**

  The Object Relational Mapper helps manage databases using Python instead of SQL, making database operations easier and less error-prone.

- **Community and Documentation**

  Django has a large and active community with extensive documentation, making learning and troubleshooting easier.

9) Django vs. Flask comparison: Which to choose and why.

- Both Django and Flask are popular Python frameworks for web development, but they serve different purposes.

- **Django**

  - Full-stack framework with many built-in features (ORM, authentication, admin panel).
  - Follows the MVT architecture.
  - Best for large projects and rapid development.
  - Suitable for complex apps like e-commerce, social networks, dashboards.

- **Flask**

  - Lightweight and minimal.
  - Provides only basic tools; extra features require extensions.
  - Developer has full control and flexibility over structure.
  - Ideal for small applications, microservices, APIs, or highly customized systems.

- **Which to choose?**
  Choose Django when you need fast development, built-in security, and support for a complex application. Choose Flask when you want flexibility, simplicity, and full control over design for smaller or experimental projects.

10) Understanding the importance of a virtual environment in Python projects.

- A virtual environment is an isolated workspace that allows a project to have its own independent Python packages and dependencies. It prevents conflicts between different project requirements on the same system.

- For example, one project may need Django 4.2 while another requires Django 5.2.7 Without a virtual environment, installing both versions together would cause errors. A virtual environment keeps each project separate and well organized.

- **<u>Benefits</u>**

  • Avoids package version conflicts

  • Keeps projects clean and manageable

  • Allows easier deployment and testing

  • Improves teamwork consistency

- **Creating a Virtual Environment**

```
python -m venv env
```

- **Activate it**

```
env\Scripts\activate       # Windows
source env/bin/activate    # Linux / macOS
```

- **Deactivate**

```
deactivate
```

11) Using venv or virtualenv to create isolated environments.

- venv and virtualenv are tools used to create separate Python environments for different projects. They allow each project to have its own dependencies without affecting system-wide installations.

- **Installation:**
  pip install virtualenv

- **Create environment:**
  virtualenv myenv

- **Activate:**
  myenv\Scripts\activate   # Windows
  source myenv/bin/activate # Linux / macOS

- Keeps dependencies separate
- Prevents version conflicts
- Makes deployment and teamwork easier

12) Steps to create a Django project and individual apps within the project.

- **Step 1: Install Django**

  pip install django

- **Step 2: Create a Django Project**

  django-admin startproject myproject

- **Step 3: Move into the Project Folder**

  cd myproject

- **Step 4: Run the Development Server**

  python manage.py runserver

- **<u>Creating an App inside the Project</u>**

- **Step 1: Create an App**

  python manage.py startapp myapp

- **Step 2: Add App to settings.py**

```
INSTALLED_APPS = [
    ...,
    'myapp',
]
```

- **Step 3: Create Views in views.py**

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello Django")
```

- **Step 4: Add URL in myapp/urls.py**

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),
]
```

- **Step 5: Include App URL in myproject/urls.py**

```
from django.urls import path, include

urlpatterns = [
    path('', include('myapp.urls')),
]
```

13) Understanding the role of manage.py, urls.py, and views.py.

- **manage.py**

  • A command-line utility created automatically when a Django project starts.

  • Used to run commands like starting the server, creating apps, applying migrations, creating users, and managing the database.

- **Example:**

```
python manage.py runserver

python manage.py makemigrations

python manage.py migrate
```

- **urls.py**

  • Controls the URL routing of the project.

  • Maps URLs to views, telling Django which function or page should respond to a request.

- **Example:**

```
urlpatterns = [
   path('', views.home),
]
```

- **views.py**

  • Contains Python functions or classes that process requests and return responses (HTML pages, JSON, etc.).

  • Acts as a link between models, templates, and logic.

- **Example:**

```
def home(request):
   return HttpResponse("Hello from Django view")
```

14) **Django’s MVT (Model-View-Template) architecture and how it handles request-response cycles.**

- **Model**

  Handles database operations. Stores and manages data using Django ORM.

- **View**

  Contains the logic that processes requests, interacts with models, and returns responses.

- **Template**

  The front-end HTML files that display data to the user.

- **Request-Response Cycle in Django**

  User sends a request through the browser (URL).

  Django receives it and checks urls.py to determine which view to call.

  The View processes the request, uses Model to get or update data if needed.

  The view passes data to a Template for display.

  The template returns the final HTML response to the user’s browser.

  Browser -> URL -> View -> Model -> Template -> Response to Browser

  This cycle continues for every request in a Django application.

15) **Introduction to Django’s built-in admin panel.**

- Django provides a powerful built-in admin panel used to manage application data through a web interface. It is automatically created when a Django project is set up and is very useful for managing databases without writing SQL manually.

- The admin panel allows you to add, edit, delete, and view records stored in the database. It is widely used by developers and site administrators to control backend data.

- **Steps to enable admin panel**

- **Create a superuser:**

```
python manage.py createsuperuser
```

- **Run the server:**

```
python manage.py runserver
```

- **Login to admin panel:**

  Open browser → http://127.0.0.1:8000/admin

- **Register models in admin**

```
from django.contrib import admin

from .models import Student

admin.site.register(Student)
```

- **Benefits**
  - Easy data management
  - No need to build a custom admin system
  - Secure and authenticated access

16) Customizing the Django admin interface to manage database records.

- Django admin so it’s easier to manage database records by creating a ModelAdmin class in admin.py and adding options that control how data is displayed and edited.

- **Example:**

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

@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
        list_display = ("name", "price", "stock")
        search_fields = ("name",)
        list_filter = ("category",)
        list_editable = ("price", "stock")
```

- **Key features:**
    - list_display shows chosen fields in the list view.
    - search_fields and list_filter help find records faster.
    - list_editable allows editing directly in the table.
    - fieldsets organizes form layout.
    - inlines lets you edit related models on the same screen.
    - Custom actions allow bulk operations.

17) Setting up URL patterns in urls.py for routing requests to views.

- URL patterns in urls.py define how incoming requests are routed to the right views. You create a list called urlpatterns and map paths to view functions or class-based views.

- **Example:**

```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.home, name="home"),
    path("products/", views.product_list, name="product_list"),
    path("products/<int:id>/",                              views.product_detail,
name="product_detail"),
]
```

- The path string matches a URL.
- Django calls the linked view.
- Optional converters like <int:id> capture dynamic values.
- Including URLs from apps
- **In the main project urls.py**:

```
from django.urls import path, include

urlpatterns = [
    path("", include("shop.urls")),
]
```

- This is how Django handles request routing cleanly across multiple apps.

18) Integrating templates with views to render dynamic HTML content.

- Templates are used to generate dynamic HTML by combining Python data from views with HTML files. In a view, you typically call render() and pass a template name along with a context dictionary.

- Django replaces placeholders in the template with real values and returns the final HTML to the browser.

- **Example:**

```
from django.shortcuts import render

def product_list(request):
   products = ["Laptop", "Phone", "Tablet"]
   return render(request, "products.html", {"products": products})
```

- **The template:**

```
{% for item in products %}
<p>{{ item }}</p>
{% endfor %}
```

19) Using JavaScript for front-end form validation.

- JavaScript can validate form inputs in the browser before sending data to the server. It checks things like empty fields, email format, password length, or number ranges.

- If something is wrong, JavaScript shows an error message and prevents form submission. This improves user experience and reduces unnecessary server requests.

- **Example:**

```
function validateForm() {
          const name = document.getElementById("name").value;
           if (name.trim() === "") {
             alert("Name is required");
             return false;
           }
```

```
  return true;
}
```

- Front-end validation works together with server-side validation for better security and usability.

20) Connecting Django to a database (SQLite or MySQL).

- Django to a database by configuring the DATABASES setting in settings.py.

- Django uses SQLite by default, but you can switch to MySQL or others by changing the engine and connection details.

- **Example for SQLite:**

```
DATABASES = {
  "default": {
    "ENGINE": "django.db.backends.sqlite3",
    "NAME": BASE_DIR / "db.sqlite3",
  }
}
```

- **Example for MySQL:**

```
DATABASES = {
  "default": {
    "ENGINE": "django.db.backends.mysql",
    "NAME": "mydb",
    "USER": "root",
    "PASSWORD": "password",
    "HOST": "localhost",
    "PORT": "3306", }
```

- After updating settings, install the driver if needed and run migrations to create tables.

21) Using the Django ORM for database queries.

- The Django ORM lets you work with the database using Python instead of writing SQL.

- Each model represents a table, and you use model methods to create, read, update, and delete records.

- **Example:**

  ```
  from .models import Product
  # create
  Product.objects.create(name="Laptop", price=50000)
  # read
  products = Product.objects.all()
  item = Product.objects.get(id=1)
  # update
  item.price = 52000
  item.save()
  # delete
  item.delete()
  ```

- ORM makes queries simpler, safer, and database-independent.

22) Understanding Django's ORM and how QuerySets are used to interact with the database.

- Django's ORM lets you work with database tables using Python classes called models. Instead of writing SQL, you use QuerySets to fetch and manipulate records. A QuerySet represents a collection of objects from the database that you can filter, sort, update, or delete.

- **Example:**

  ```
  products = Product.objects.filter(category="Electronics")
  ```

- QuerySets are lazy, so they only run the query when needed. They make database operations safer, cleaner, and portable across different database systems.

23) Using Django's built-in form handling.

- Django's built-in form handling helps you create forms, validate input, and process submitted data safely.

- You define a form class, render it in a template, and then check if it's valid when the user submits it.

- Django handles validation, error messages, and converting data to Python types.

- **Example:**

```
from django import forms

class ContactForm(forms.Form):

    name = forms.CharField()
    email = forms.EmailField()
```

- This system makes working with forms more secure and organized.

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

- Django provides a complete authentication system that handles user accounts, sign up, login, logout, and password management. You can use built-in views and forms to avoid writing everything manually.

- **Example setup in urls.py:**

```
from django.urls import path

from django.contrib.auth import views as auth_views

from . import views
```

```
urlpatterns = [
    path("signup/", views.signup, name="signup"),
    path("login/", auth_views.LoginView.as_view(), name="login"),
    path("logout/", auth_views.LogoutView.as_view(), name="logout"),
]
```

- **Sign up view example:**

```
from django.contrib.auth.forms import UserCreationForm
from django.shortcuts import render, redirect


def signup(request):
    form = UserCreationForm(request.POST or None)
    if request.method == "POST" and form.is_valid():
        form.save()
        return redirect("login")
    return render(request, "signup.html", {"form": form})
```

- Django also includes built-in password reset and change views that can be wired the same way. The system manages sessions and security features automatically.

25) Using AJAX for making asynchronous requests to the server without reloading the page.

- AJAX lets the browser send and receive data from the server in the background without reloading the whole page. You can update part of a page dynamically, which makes the interface faster and smoother.

- **Example with JavaScript fetch:**

```
fetch("/update-cart/", {
  method: "POST",
  headers: {"Content-Type": "application/json"},
  body: JSON.stringify({ product_id: 5 })
})
.then(response => response.json())
.then(data => {
  document.getElementById("cart-count").innerText = data.count;
});
```

- The server returns JSON, and JavaScript updates only the needed section. This improves user experience and reduces unnecessary page loads.

26) Techniques for customizing the Django admin panel.

- You can customize the Django admin panel by registering models with a ModelAdmin class and using options that change how data appears and is managed.

- **Common techniques:**
    - list_display to control columns in the list view.
    - search_fields and list_filter to make records easier to find.
    - list_editable for inline editing.
    - fieldsets to organize form layout.
    - readonly_fields to protect generated fields.
    - inlines to edit related models on the same page.
    - Custom actions for bulk operations.

- These features help tailor the admin to your project and improve data management.

27) Introduction to integrating payment gateways (like Paytm) in Django projects.

- Integrating a payment gateway like Paytm in a Django project involves adding a secure way to accept online payments from users.

- You create an order on your server, send payment details to Paytm, and then handle the response after the transaction.

- Paytm provides API keys, callback URLs, and sample code for generating and verifying signatures.

- The Django view sends payment data to Paytm, and the callback view checks the transaction status and updates the order in the database.

- This setup ensures secure payment processing and smooth user checkout.

28) Steps to push a Django project to GitHub.

- Here are the basic steps to push a Django project to GitHub:

    - Create a new repository on GitHub.

    - Open your project folder in a terminal.

    - Initialize Git:
      ```
      git init
      ```

    - Add all files:
      ```
      git add .
      ```

    - Commit the project:
      ```
      git commit -m "Initial commit"
      ```

    - Add the GitHub repo as a remote:
      ```
      git remote add origin https://github.com/username/repository.git
      ```

    - Push the code:
      ```
      git push -u origin master
      ```

- Make sure to create a .gitignore file and exclude the venv/ folder, migrations cache, and __pycache__ files.

- That publishes your Django project to GitHub.

29) Introduction to deploying Django projects to live servers like PythonAnywhere.

- Deploying a Django project to a live server like PythonAnywhere means moving your local app to a hosted environment so others can access it online.

- You upload your code (usually from GitHub), set up a virtual environment, install dependencies, configure the database, and point the web app to your wsgi.py file.

- After updating static files and setting allowed hosts, you reload the server.

- PythonAnywhere handles hosting, domain setup, and HTTPS, which makes deployment easier and suitable for beginners.

30) Setting up social login options (Google, Facebook, GitHub) in Django using OAuth2.

- Django using OAuth2 providers like Google, Facebook, or GitHub by using a library such as django-allauth.

- It handles authentication flow, permissions, and callbacks for you.

- **Basic steps:**

  1. Install and configure django-allauth.
  2. Add it to INSTALLED_APPS and update AUTHENTICATION_BACKENDS.
  3. Include allauth URLs in your main urls.py.
  4. Create OAuth credentials in Google, Facebook, or GitHub developer consoles.
  5. Add client ID, secret key, and callback URL to Django settings.
  6. Use the built-in login templates or link to /accounts/login/.
     This allows users to sign in with their existing accounts instead of creating a new one.

31) Integrating Google Maps API into Django projects.

- You can integrate Google Maps into a Django project by loading the Maps JavaScript API in a template and passing location data from a view.

- First, get an API key from Google Cloud and enable Maps JavaScript API.
- Then include the script in your HTML and place a container where the map will render.
- Use JavaScript to initialize the map and add markers based on coordinates sent from the server.
- This allows you to display dynamic locations such as store addresses, user check-ins, or delivery tracking inside your Django pages.

****