**Module 9**

Python DB and Framework

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

* When building web applications using Python, frameworks like **Django** and **Flask** allow embedding HTML within Python code to dynamically generate web pages. This is done using **templates**, which separate business logic from presentation.

**Using Django to Embed HTML**

Django is a full-featured web framework that follows the **MVC (Model-View-Controller)** pattern.

1. Generating dynamic HTML content using Django templates.

* Django's **template system** allows you to generate dynamic HTML content by embedding Python-like logic within the template files. This makes it easy to pass data from views to templates and dynamically render web pages.

1. **CSS in Python**
2. Integrating CSS with Django templates.

* Integrating CSS with Django templates is straightforward. We can use both **internal** and **external** CSS in Django templates.
* Organizing Static Files:

**your\_project/**

**│── your\_app/**

**│ ├── static/**

**│ │ ├── css/**

**│ │ │ ├── styles.css**

**│ │ ├── images/**

**│ │ ├── js/**

* Configure Static Settings:

**STATIC\_URL = '/static/'**

**STATICFILES\_DIRS = [**

**os.path.join(BASE\_DIR, "static"),**

**]**

**STATIC\_ROOT = os.path.join(BASE\_DIR, 'staticfiles')**

1. How to serve static file (like CSS, JavaScript) in Django.

* Static files like **CSS, JavaScript, and images** are managed using the **static files framework**.
* Organizing Static Files:

**your\_project/**

**│── your\_app/**

**│ ├── static/**

**│ │ ├── css/**

**│ │ │ ├── styles.css**

**│ │ ├── images/**

**│ │ ├── js/**

1. **JavaScript in Python**
2. Using JavaScript for client-side interactivity in Django templates.

* Django is a backend framework, but you can easily integrate JavaScript in its templates to enhance **client-side interactivity**. You can use JavaScript for **event handling, dynamic content updates, form validation, AJAX calls, and animations**.

1. Linking External or internal JavaScript file in Django.

* You can link JavaScript in Django templates in **two ways**:
* **Internal JavaScript** (written inside the HTML file)
* **External JavaScript** (stored in a separate .js file inside the static/ folder).

1. **Django Introduction:**
2. Overview of Django: Web development framework.

* **Django** is a high-level Python web framework that enables rapid development of secure and maintainable websites. It follows the **Model-View-Template (MVT)** architectural pattern and emphasizes **reusability, scalability, and security**.

1. Advantage of Django (e.g., scalability, security).

* Django is a powerful and feature-rich framework that offers several advantages for web development:

**Scalability:** Whether you're building a small personal project or a large, high-traffic website, Django is designed to handle it. Its architecture allows you to manage increasing numbers of users and data without needing a complete overhaul. Think of it as a solid foundation that can grow as your needs do.

**Security Focused:** Django takes security seriously right from the start. It comes with built-in protections against common web threats like SQL injection and cross-site scripting. This helps you build more secure applications without having to implement everything from scratch.

1. Django VS Flask comparison: Which to choose and why.

| **Feature** | **Django** | **Flask** |
| --- | --- | --- |
| Framework Type | Full-stack web framework | Micro web framework |
| Architecture | Model-View-Template (MVT) | No specific architecture |
| Built-in Features | Includes many built-in features | Minimalistic, only essential features |
| Admin Panel | Built-in admin panel | No built-in admin panel |
| ORM | Includes a powerful ORM | No built-in ORM |
| Template Engine | Uses Django Template Language (DTL) | Uses Jinja2 template engine |
| Security | Built-in protection against vulnerabilities | Requires manual implementation |
| Scalability | Suitable for large-scale applications | Suitable for small to medium projects |
| Community Support | Large and active community | Strong and supportive community |
| Flexibility | Less flexible, more opinionated | More flexible, allows more freedom |
| Learning Curve | Steeper learning curve | Easier learning curve |

1. **Virtual Environment:**
2. Understanding the importance of a virtual environment in Python

* A **virtual environment** in Python is an isolated workspace that allows developers to install dependencies and packages without affecting the global Python environment. It is crucial for managing project-specific dependencies efficiently.

1. Using venv or virtualenv to create isolated environments.

* **venv** and **virtualenv** are tools used to create isolated environments for managing project-specific dependencies.

1. **Project and App Creation**
2. 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: Navigate to the Project Directory

cd Myproject

Step 4: Run the Django Development Server

python manage.py runserver

Open http://127.0.0.1:8000/ in the browser. You should see the Django welcome page

Step 5: Create an App Inside the Project

python manage.py startapp Myapp

Step 6: Register the App in settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'Myapp',

]

Step 7: Define a Model (Database Table) in models.py

from django.db import models

class Student(models.Model):

name = models.CharField(max\_length=100)

age = models.IntegerField()

Step 8: Run Migrations (Apply Database Changes)

python manage.py makemigrations

python manage.py migrate

Step 9: Create a URL Pattern for the App

from django.urls import path

from . import views

urlpatterns = [

path('', views.home, name='home'),

]

Step 10: Run the Server and Test

python manage.py runserver

Open http://127.0.0.1:8000/Myapp/ in the browser

1. Understanding the role of manage.py, urls.py, and views.py.

* Three key files play an essential role in handling the application's functionality and request flow:

1. manage.py

Purpose: This is a command-line utility for managing the Django project.

Common Uses:

Running the development server: python manage.py runserver

Creating database migrations: python manage.py makemigrations

Applying migrations: python manage.py migrate

Creating superusers: python manage.py createsuperuser

Running custom Django commands

2. urls.py

Purpose: Defines URL patterns and routes incoming HTTP requests to the appropriate views.

Key Concepts:

Uses the urlpatterns list to map URLs to views.

path() or re\_path() functions are used to define URL patterns.

Can include multiple urls.py files for modularity.

from django.urls import path

from . import views

urlpatterns = [

path('', views.home, name='home'),

path('about/', views.about, name='about'),

]

views.py

Purpose: Handles the business logic and renders responses to requests.

Types of Views:

Function-based views (FBV): Uses functions to handle requests.

Class-based views (CBV): Uses classes to handle requests for better reusability.

from django.http import HttpResponse

def home(request):

return HttpResponse("Welcome to the home page!")

def about(request):

return HttpResponse("About us page")

1. **MVT Pattern Architecture**
2. Django’s MVT (Model-view-Template) architecture and how it handles request-rosponse cycle.

* Django follows the MVT (Model-View-Template) architecture, which is a variant of the traditional MVC (Model-View-Controller) pattern. It is designed to separate concerns, making development easier and more scalable.

1. Components of MVT:

Model (M)

Represents the database structure and handles data storage.

Uses Django’s ORM (Object-Relational Mapping) to interact with the database.

from django.db import models

class Product(models.Model):

name = models.CharField(max\_length=100)

price = models.DecimalField(max\_digits=10, decimal\_places=2)

stock = models.IntegerField()

View (V)

Handles the business logic and processes HTTP requests.

Fetches data from the Model and passes it to the Template.

from django.shortcuts import render

from .models import Product

def product\_list(request):

products = Product.objects.all()

return render(request, 'product\_list.html', {'products': products})

Template (T)

Defines the presentation layer (HTML, CSS, JavaScript).

Renders dynamic content based on data passed from the View.

<h1>Product List</h1>

<ul>

{% for product in products %}

<li>{{ product.name }} - ${{ product.price }}</li>

{% endfor %}

</ul>

Django Request-Response Cycle

Step-by-Step Process

Client Sends a Request

The browser sends an HTTP request to the Django server (e.g., https://example.com/products/).

URL Routing (urls.py)

Django's URL dispatcher (urls.py) maps the request to the appropriate view.

from django.urls import path

from .views import product\_list

urlpatterns = [

path('products/', product\_list, name='product\_list'),

]

View Processes the Request (views.py)

The corresponding view function is executed.

It interacts with the Model to retrieve data.

It then renders an HTML template with the data.

Model Handles Data (models.py)

The view fetches data from the database using Django's ORM.

products = Product.objects.all()

Template Renders the Data (templates/)

The data from the Model is dynamically inserted into the HTML template.

<h1>Product List</h1>

<ul>

<li>Phone - $699.99</li>

<li>Laptop - $1299.99</li>

</ul>

Django Returns the Response

The final rendered HTML page is sent back to the client's browser.

1. **Django Admin Panel**
2. Introduction to Django’s built-in admin panel.

* Django provides a built-in admin panel that allows developers to manage application data easily without writing additional code. It acts as an automatic, web-based interface for database management.

1. Customizing the Django admin interface to manage database records.

* Django’s **admin panel** is highly customizable, allowing you to enhance its functionality and improve the user experience when managing database records.

1. **URL Patterns and Template Integration**
2. Setting up URL patterns in urls.py for routing requests to views.

* urls.py is responsible for mapping URLs to views. It acts as a router that directs incoming requests to the appropriate view function or class.

1. Integrating templates with views to render dynamic HTML content.

* Django’s **template system** allows rendering dynamic HTML pages by combining static content with data from views.

1. **Form Validation using JavaScript**
2. Using JavaScript for front-end form validation.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Django Form Validation</title>

<script>

function validateForm() {

let isValid = true;

let name = document.getElementById("name").value.trim();

let email = document.getElementById("email").value.trim();

let password = document.getElementById("password").value.trim();

document.getElementById("nameError").innerText = "";

document.getElementById("emailError").innerText = "";

document.getElementById("passwordError").innerText = "";

if (name === "") {

document.getElementById("nameError").innerText = "Name is required";

isValid = false;

}

let emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

if (email === "") {

document.getElementById("emailError").innerText = "Email is required";

isValid = false;

} else if (!emailPattern.test(email)) {

document.getElementById("emailError").innerText = "Invalid email format";

isValid = false;

}

if (password.length < 6) {

document.getElementById("passwordError").innerText = "Password must be at least 6 characters";

isValid = false;

}

return isValid;

}

</script>

</head>

<body>

<h2>Django Form with JavaScript Validation</h2>

{% if success %}

<p style="color: green;">{{ success }}</p>

{% endif %}

<form method="POST" action="" onsubmit="return validateForm()">

{% csrf\_token %}

<label for="name">Name:</label>

<input type="text" id="name" name="name">

<span id="nameError" style="color:red;"></span>

<br><br>

<label for="email">Email:</label>

<input type="text" id="email" name="email">

<span id="emailError" style="color:red;"></span>

<br><br>

<label for="password">Password:</label>

<input type="password" id="password" name="password">

<span id="passwordError" style="color:red;"></span>

<br><br>

<button type="submit">Submit</button>

</form>

</body>

</html>

1. **Django Database Connectivity (MySQL or SQLite)**
2. Connecting Django to a database (SQLite or MySQL).

* Django automatically configures SQLite as the default database. The settings can be found in settings.py:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / "db.sqlite3", # Database file stored in the project root

}

}

After setting up SQLite, run the following command to create the database file (db.sqlite3):

python manage.py migrate

1. Using the Django ORM for database queries.

* Django **ORM (Object-Relational Mapping)** allows you to interact with databases using Python **without writing raw SQL queries**. We can **create, read, update, and delete (CRUD)** records efficiently.

1. **ORM and Query Sets**
2. Understanding Django’s ORM and how Query Sets are used to interact with the database.

* Django’s Object-Relational Mapping (ORM) provides a way to interact with the database using Python code instead of writing raw SQL queries. It allows developers to define database structures using models and perform queries using QuerySets.

1. Django ORM Overview

Django ORM acts as an abstraction layer that converts Python objects into database tables and vice versa. This allows seamless database interactions.

2. QuerySets in Django

A QuerySet is a collection of database queries that fetch data from the database.

3. QuerySet Chaining

Django allows chaining multiple QuerySet methods

1. **Django Forms and Authentication**
2. Using Django’s built-in form handling. & Implement Django’s authentication system

Django provides a powerful forms framework to handle user input efficiently. It simplifies form creation, validation, and rendering.

1. Creating a Django Form

Django has a built-in forms module that allows defining forms as Python classes.

Example: Creating a Form

from django import forms

class StudentForm(forms.Form):

name = forms.CharField(max\_length=100, label="Full Name")

age = forms.IntegerField(min\_value=18, label="Age")

email = forms.EmailField(label="Email Address")

2. Using Forms in a Django View

Forms can be rendered and processed inside a Django view.

Example: Displaying a Form in a View

from django.shortcuts import render

from .forms import StudentForm

def student\_view(request):

form = StudentForm()

return render(request, 'student\_form.html', {'form': form})

3. Rendering the Form in a Template

Inside student\_form.html, we can use:

<form method="post">

{% csrf\_token %}

{{ form.as\_p }} <!-- Displays form fields inside <p> tags -->

<button type="submit">Submit</button>

</form>

4. Handling Form Submission

We modify the view to process user input.

def student\_view(request):

if request.method == 'POST':

form = StudentForm(request.POST)

if form.is\_valid(): # Checks validation

name = form.cleaned\_data['name']

age = form.cleaned\_data['age']

email = form.cleaned\_data['email']

# Process the data (e.g., save to database)

return render(request, 'success.html', {'name': name})

else:

form = StudentForm()

return render(request, 'student\_form.html', {'form': form})

ModelForms: Using Forms with Models

Django provides ModelForms to directly work with database models.

Example: Creating a ModelForm

from django.forms import ModelForm

from .models import Student

class StudentModelForm(ModelForm):

class Meta:

model = Student

fields = ['name', 'age', 'email']

1. **CRUD Operations using AJAX**
2. Using AJAX for making asynchronous requests to the server without reloading the page.

* AJAX (Asynchronous JavaScript and XML) allows sending and receiving data from the server without reloading the page. In Django, AJAX is commonly used with JavaScript (Fetch API, jQuery, or Axios) to send data and get responses dynamically.

view.py

from django.http import JsonResponse

from django.views.decorators.csrf import csrf\_exempt

import json

@csrf\_exempt # Disable CSRF for simplicity (Not recommended for production)

def process\_ajax(request):

if request.method == "POST":

data = json.loads(request.body) # Get JSON data from request

name = data.get('name', '')

response\_data = {'message': f'Hello, {name}!'}

return JsonResponse(response\_data) # Return JSON response

return JsonResponse({'error': 'Invalid request'}, status=400)

1. **Customizing the Django Admin Panel**
2. Techniques for customizing the Django admin panel.

* Django’s admin panel is a powerful interface for managing database records. While the default setup is functional, customization can improve usability, appearance, and security.

from django.contrib import admin

from core.models import Person

@admin.register(Person)

class PersonAdmin(admin.ModelAdmin):

list\_display = ('last\_name', 'first\_name')

@admin.register(Person)

class PersonAdmin(admin.ModelAdmin):

list\_display = ('last\_name', 'first\_name')

list\_filter = ('last\_name',)

search\_fields = ('last\_name', 'first\_name')

from django import forms

from core.models import Person

class PersonAdminForm(forms.ModelForm):

class Meta:

model = Person

fields = '\_\_all\_\_'

def clean\_first\_name(self):

if self.cleaned\_data['first\_name'] == 'Spike':

raise forms.ValidationError("No Vampires")

return self.cleaned\_data['first\_name']

@admin.register(Person)

class PersonAdmin(admin.ModelAdmin):

form = PersonAdminForm

fields = ('first\_name', 'last\_name', 'courses')

1. **Payment Integration using Paytm**
2. Introduction to integrating payment gateways(like Paytm) in Django projects.
3. **GitHub Project Deployment**
4. Steps to push a Django project to GitHub.
5. **Live Project Deployment (PythonAnywhere)**
6. Introduction to deploying Django projects to live servers like Python Anywhere.
7. **Social Authentication**
8. Setting up social login options (Google, Facebook, GitHub) in Django using OAuth2.
9. **Google Maps API**
10. Integrating Google Maps API into Django projects.