Module 9 : Python DB and Framework

1. HTML in Python

(Q1) Introduction to embedding HTML within Python using web frameworks like Django or Flask.

Ans: When building web applications with **Python**, you don't embed raw HTML directly into Python scripts like PHP or inline scripting. Instead, frameworks like **Django** and **Flask** use **template engines** to render HTML pages dynamically using data from Python.

□ **Flask Example:**

Flask uses the **Jinja2** templating engine.

```
# app.py
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def home():
    return render_template("index.html", name="Alice")

<!-- templates/index.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Welcome</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

When you run this Flask app, navigating to `/` renders the HTML with `{{ name }}` replaced by "Alice".

□ **Django Example:**

Django uses its own templating engine (very similar to Jinja2).

```
# views.py
from django.shortcuts import render

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

<!-- templates/index.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Home</title>
</head>
<body>
```

```
    <h1>Welcome, {{ name }}!</h1>
</body>
</html>
```

In Django, you typically place templates inside an app's `templates/` folder, and render them using the `render()` function in views.

(Q2) Generating dynamic HTML content using Django templates.

Ans: Django templates let you **inject dynamic data**, **loop through items**, and **control logic flow** in HTML using **template tags and variables**.

 Template Variables:

```
<p>Hello, {{ user.first_name }}!</p>
```

 Loops:

```
<ul>
  {% for product in products %}
    <li>{{ product.name }} - ${{ product.price }}</li>
  {% endfor %}
</ul>
```

 Conditionals:

```
{% if user.is_authenticated %}
  <p>Welcome back, {{ user.username }}!</p>
{% else %}
  <p>Please log in.</p>
{% endif %}
```

 Including Templates:

Break your HTML into smaller chunks and reuse them:

```
{% include "navbar.html" %}
```

 Template Inheritance:

```
<!-- base.html -->
<html>
<head><title>{% block title %}My Site{% endblock %}</title></head>
<body>
  {% block content %}{% endblock %}
</body>
</html>

<!-- index.html -->
{% extends "base.html" %}

{% block title %}Home{% endblock %}
{% block content %}
```

```
  <h1>Welcome to the Homepage</h1>
{% endblock %}
```

2. CSS in Python

(Q1) Integrating CSS with Django templates.

Ans: To use **CSS** in Django templates:

1. **Create a static folder** inside your Django app (or project).
2. **Place your CSS file** in the static folder (e.g., `static/css/styles.css`).
3. **Load the static files** in your template using the `{% load static %}` tag.
4. **Link the CSS** in your HTML with a relative path via `{% static %}`.

☐ **Example:**

☐ Project structure:

```
myproject/
├── myapp/
│   ├── static/
│   │   └── css/
│   │       └── styles.css
│   ├── templates/
│   │   └── index.html
```

☐ `styles.css`:

```
body {
    background-color: #f0f0f0;
    font-family: Arial, sans-serif;
}
```

☐ `index.html`:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>Styled Page</title>
    <link rel="stylesheet" href="{% static 'css/styles.css' %}">
</head>
<body>
    <h1>Welcome to Django</h1>
</body>
</html>
```

(Q2) How to serve static files (like CSS, JavaScript) in Django.

Ans: **Step-by-step:**

1. **Set up `STATIC_URL`** in `settings.py`:

```
# settings.py
STATIC_URL = '/static/'
```

2. **Optionally define `STATICFILES_DIRS`** (for global static files):

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

3. **During development**, Django serves static files automatically when `DEBUG = True`.
4. **For production**, you'll need to collect and serve them using `collectstatic` and a proper web server (e.g., Nginx).

□ Bonus: Serving Static Files in Development

In your `urls.py` (only when `DEBUG = True`):

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # your URL patterns
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

3. JavaScript with Python

(Q1) Using JavaScript for client-side interactivity in Django templates.

Ans: Django templates render HTML on the server, but you can freely include **JavaScript** in those HTML files to add dynamic behavior like:

- Handling button clicks
- Form validation
- Fetching data (via AJAX or Fetch API)
- Animations, etc.

□ Example: Button Click Alert

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>JS Example</title>
</head>
<body>
    <h1>Click the button</h1>
    <button onclick="sayHello()">Click me</button>

    <script>
```

```
        function sayHello() {
            alert("Hello from JavaScript!");
        }
    </script>
</body>
</html>
```

You can also insert Django template variables into JavaScript:

```
<script>
    const username = "{{ user.username }}";
    console.log("Logged in user:", username);
</script>
```

□ **Note**: Always escape data from Django in JS to avoid XSS. Use the `escapejs` filter:

```
js
CopyEdit
const safeData = "{{ some_data|escapejs }}";
```

(Q2)Linking external or internal JavaScript files in Django.

Ans: **Step-by-step:**

1. Create a JavaScript file inside your app's `static/` directory.
2. Use `{% load static %}` and the `{% static 'path/to/file.js' %}` tag to link it.

□ **Example File Structure:**

```
myapp/
├── static/
│   └── js/
│       └── script.js
├── templates/
│   └── index.html
│
```

□ **`script.js`:**

```
function greetUser() {
    alert("Welcome to the site!");
}
```

□ **`index.html`:**

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>JS Linked</title>
    <script src="{% static 'js/script.js' %}"></script>
</head>
<body>
    <button onclick="greetUser()">Greet</button>
```

```
</body>
</html>
```

4. Django Introduction

(Q1) Overview of Django: Web development framework.

Ans: **Django** is a **high-level Python web framework** that encourages **rapid development** and **clean, pragmatic design**.

 Key Features:

- Follows the **Model-View-Template (MVT)** architecture.
- Comes with **built-in admin interface**, **ORM**, **authentication system**, **form handling**, and more.
- Emphasizes **reusability**, **scalability**, and **security**.

 Core Components:

Component	Description
Model	Defines database schema (using Python classes)
View	Business logic, processes requests, and returns responses
Template	HTML files with placeholders for dynamic content

 Example Use Cases:

- Blogs, e-commerce websites
- Enterprise web apps
- News sites (e.g., The Washington Post, Instagram)

(Q2) Advantages of Django (e.g.,scalability,security).

Ans:

Advantage	Explanation
Scalability	Used by large-scale apps (e.g., Instagram), supports caching and load balancing
Security	Protects against SQL injection, XSS, CSRF, and clickjacking automatically
Batteries Included	Comes with many features out of the box: admin panel, user auth, form handling
Rapid Development	Built-in tools (like `django-admin`) help you scaffold apps fast

ORM Support	Interact with databases using Python objects instead of raw SQL
Template System	Clean separation of logic and presentation via Django templates
Community & Docs	Mature, stable framework with excellent documentation and a huge community

(Q3) Django vs. Flask comparison: Which to choose and why.

Ans:

Feature	Django	Flask
Philosophy	"Batteries-included" – many built-in tools	Minimalistic and flexible
Architecture	MVT (Model-View-Template)	You define your own architecture
Admin Interface	Built-in and powerful	Not built-in
ORM	Built-in (Django ORM)	Optional (can use SQLAlchemy)
Learning Curve	Steeper (more to learn upfront)	Easier for beginners
Project Size Suitability	Ideal for large, scalable apps	Great for small to medium projects
Customization	Less flexible due to conventions	Highly flexible and lightweight
Community	Large and well-established	Also strong, especially for APIs

⍰ Choose Django if:

- You want a **fully-featured framework** out of the box.
- You're building a **large, database-heavy** or enterprise-level app.
- You need **user authentication**, **admin dashboard**, and **security** handled for you.

☐ Choose Flask if:

- You prefer **full control** over components.
- You're building a **simple API** or microservice.
- You're new to web development and want a **gentler learning curve**.

5. Virtual Environment

(Q1) Understanding the importance of a virtual environment in Python projects.

Ans: A **virtual environment** is an isolated Python environment where you can install packages and dependencies **specific to a single project** — without affecting your global Python installation or other projects.

Why It's Important:

Benefit	Description
Dependency Isolation	Prevents conflicts between different projects using different package versions.
Project Portability	Easy to share environment with others via `requirements.txt`.
Cleaner Development	Keeps your global Python clean and uncluttered.
Version Control	Different projects can use different Python or package versions safely.

Real Example:

- Project A needs `Django 4.2`
- Project B needs `Django 3.2`
  → Without virtual environments, you **can't install both** on the same system without conflicts.

(Q2) Using venv or virtualenv to create isolated environments.

Ans: **Option 1: Using `venv` (built-in in Python 3.3+)**

Step-by-step:

1. **Create a virtual environment:**

   ```
   python -m venv env
   ```

   This creates a folder named `env/` with its own Python interpreter and `pip`.

2. **Activate the environment:**
   - On **Windows**:

     ```
     .\env\Scripts\activate
     ```

   - On **macOS/Linux**:

     ```
     source env/bin/activate
     ```

3. **Install packages inside the environment:**

   ```
   pip install django
   ```

4. **Deactivate when done:**

   ```
   deactivate
   ```

□ Option 2: Using `virtualenv` (third-party tool, works for older Python versions)

Install it (if not already):

```
pip install virtualenv
```

Then create and activate:

```
virtualenv myenv
source myenv/bin/activate  # or .\myenv\Scripts\activate on Windows
```

✓Bonus: Save and Reuse Environments

To share your environment with others:

```
pip freeze > requirements.txt
```

To install from `requirements.txt` in a new environment:

```
pip install -r requirements.txt
```

6. Project and App Creation

(Q1) Stepsto create a Django project and individual apps within the project.

Ans : □ 1. Create a Virtual Environment (Recommended)

```
python -m venv env
source env/bin/activate       # or .\env\Scripts\activate on Windows
```

□ 2. Install Django

```
pip install django
```

□ 3. Create a Django Project

```
django-admin startproject myproject
cd myproject
```

This creates a structure like:

```
myproject/
├── manage.py
├── myproject/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
```

▯ 4. Create an App Inside the Project

```
python manage.py startapp myapp
```

Now structure becomes:

```
myproject/
├── myapp/
│   ├── admin.py
│   ├── apps.py
│   ├── models.py
│   ├── views.py
│   ├── urls.py         ← (create manually)
│   └── ...
```

▯ 5. Add the App to `INSTALLED_APPS` in `settings.py`

```
# myproject/settings.py
INSTALLED_APPS = [
    ...,
    'myapp',
]
```

(Q2) Understanding the role of manage.py, urls.py, and views.py.

Ans: **▯ 1. `manage.py`**

- A **command-line utility** to manage your project.
- You use it to run the server, make migrations, create superusers, etc.

Examples:

```
python manage.py runserver          # Start development server
python manage.py makemigrations     # Create migration files
python manage.py migrate            # Apply migrations
```

▯ 2. `urls.py`

Handles **URL routing** — it maps incoming URLs to the appropriate view functions.

Project-level (`myproject/urls.py`)

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),   # Delegates to app URLs
]
```

App-level (`myapp/urls.py`) — create this manually

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

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

□ 3. `views.py`

Contains **functions or classes** that define what to display when a URL is accessed.

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello, world!")
```

You can also return templates:

```
from django.shortcuts import render

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

7. MVT Pattern Architecture

(Q1) Django's MVT (Model-View-Template) architecture and how it handles request-response cycles.

Ans: □ **What is MVT?**

MVT stands for:

Component	Role
Model	Manages data and business logic (Database interaction)
View	Receives user request and returns a response (via logic)
Template	Handles presentation (HTML + dynamic content)

□ How Django Handles a Request-Response Cycle

Here's a step-by-step explanation of what happens when a user accesses a Django-powered webpage:

□ 1. Request Sent by Browser

A user visits `http://example.com/products/`.

□ 2. URL Dispatcher (`urls.py`)

Django looks in `urls.py` to **match the URL** to a **view function**.

```
# project/urls.py
urlpatterns = [
    path('products/', include('store.urls')),
]
python
CopyEdit
# store/urls.py
urlpatterns = [
    path('', views.product_list, name='product_list'),
]
```

□ 3. View Function is Called (`views.py`)

The matched view runs the logic. It can query the database and pass data to a template.

```
# views.py
from django.shortcuts import render
from .models import Product

def product_list(request):
    products = Product.objects.all()
    return render(request, 'products.html', {'products': products})
```

□ 4. Model is Used for Data (if needed)

Models interact with the database using Django’s ORM.

```
# models.py
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=8, decimal_places=2)
```

□ 5. Template is Rendered (`templates/products.html`)

The template receives data from the view and renders dynamic HTML.

```
<!-- products.html -->
<h1>Product List</h1>
<ul>
  {% for product in products %}
    <li>{{ product.name }} - ${{ product.price }}</li>
  {% endfor %}
</ul>
```

□ 6. Response is Sent Back to the Browser

The final rendered HTML is returned to the browser as an HTTP response.

8. Django Admin Panel

(Q1) Introduction to Django's built-in admin panel.

Ans: The **Django Admin Panel** is an **auto-generated web interface** that allows you to:

- Manage your app's data (create, read, update, delete records)
- Manage users and permissions
- Monitor models registered in the project
- Access a secure backend with user login

□ How to Use the Admin Panel:

Step 1: Enable Admin in Project URLs

```
# project/urls.py
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Step 2: Create a Superuser

```
python manage.py createsuperuser
```

Enter a username, email, and password when prompted.

Step 3: Run the Server

```
python manage.py runserver
```

(Q2) Customizing the Django admin interface to manage database records.

Ans : □ Step 1: Register Your Model

In `admin.py` of your app:

```
# myapp/admin.py
from django.contrib import admin
from .models import Product

admin.site.register(Product)
```

This makes your `Product` model appear in the admin panel.

□ Step 2: Customize Admin Display

You can customize **how** models appear and behave in the admin using a `ModelAdmin` class.

✅Example: Custom Product Admin

```
# myapp/admin.py
from django.contrib import admin
from .models import Product

class ProductAdmin(admin.ModelAdmin):
    list_display = ('name', 'price', 'in_stock')
    search_fields = ('name',)
    list_filter = ('category', 'in_stock')
    ordering = ('name',)

admin.site.register(Product, ProductAdmin)
```

□ Common Customizations:

Feature	Admin Option
List columns	`list_display = ('field1', 'field2')`
Search bar	`search_fields = ('field',)`
Filters sidebar	`list_filter = ('field',)`
Sorting	`ordering = ('field',)`
Editable fields	`fields = ('field1', 'field2')`
Inline models	Use `TabularInline` or `StackedInline` classes

⍰ Example of Inline Editing

If a `Book` model has a foreign key to `Author`, you can allow editing books directly in the Author admin page.

```
class BookInline(admin.TabularInline):
    model = Book
    extra = 1

class AuthorAdmin(admin.ModelAdmin):
    inlines = [BookInline]

admin.site.register(Author, AuthorAdmin)
```

9. URL Patterns and Template Integration

(Q1) Setting up URL patterns in urls.py for routing requests to views.

Ans: Django uses **URL patterns** to match incoming browser requests to the correct view function.

⍰ Basic Routing Flow:

1. **User enters a URL** (e.g., `/about`)
2. Django looks in `urls.py` to find a matching path
3. It calls the corresponding **view function**

⍰ Step-by-Step Example

□ Project-level `urls.py` (e.g., `myproject/urls.py`)

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),  # Delegates routing to app-level
]
```

□ App-level `urls.py` (e.g., `myapp/urls.py`)

You create this manually in your app folder.

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

urlpatterns = [
    path('', views.home, name='home'),  # Route for homepage
    path('about/', views.about, name='about'),  # Route for about page
]
```

(Q2) Integrating templates with views to render dynamic HTML content.

Ans : In Django, **views** can return a full HTML page rendered using **templates**.

▯ Step-by-Step Example

▯ **views.py**

```
from django.shortcuts import render

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

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

▯ Project Folder Structure:

```
myproject/
│
├── myapp/
│   ├── views.py
│   ├── urls.py
│   └── templates/
│       └── home.html
│       └── about.html
│
```

✅Create a `templates` folder inside the app and place your `.html` files there.

▯ **home.html**

```
<!DOCTYPE html>
<html>
<head>
    <title>Home</title>
</head>
<body>
    <h1>Welcome to the Home Page!</h1>
</body>
</html>
```

▯ Ensure Template Settings Are Correct

In `settings.py`, make sure `'APP_DIRS': True` is enabled in the `TEMPLATES` section:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'APP_DIRS': True,
        ...
    },
]
```

✅Request-to-Response Flow

```
Browser → URL → urls.py → views.py → render(template.html) → HTML response
```

10. Form Validation using JavaScript

(Q1) Using JavaScript for front-end form validation.

Ans: ▯ **Why Use JavaScript Validation?**

- Instant feedback for users
- Prevents submitting empty or incorrect fields
- Reduces server-side validation errors

▯ Example: Basic HTML Form with JavaScript Validation

▯ `form.html`

```
<!DOCTYPE html>
<html>
<head>
  <title>Contact Form</title>
  <script>
    function validateForm() {
      const name = document.forms["contactForm"]["name"].value;
      const email = document.forms["contactForm"]["email"].value;

      if (name === "") {
        alert("Name must be filled out");
        return false;
      }

      // Basic email check
      const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
      if (!emailPattern.test(email)) {
        alert("Please enter a valid email address");
        return false;
      }

      return true; // Allow form submission
    }
  </script>
</head>
<body>
  <h2>Contact Us</h2>
  <form name="contactForm" onsubmit="return validateForm()">
    <label>Name:</label><br>
    <input type="text" name="name"><br><br>
    <label>Email:</label><br>
    <input type="text" name="email"><br><br>

    <input type="submit" value="Submit">
  </form>
</body>
</html>
```

✅What's Happening:

- `onsubmit="return validateForm()"`: JavaScript runs when the user tries to submit the form.
- If validation fails (`return false`), the form is **not submitted**.
- If valid, form submission continues (`return true`).

✅Best Practices

Tip	Why it Helps
Validate both on client and server	JavaScript can be bypassed, so always validate in Django too
Use regex for email/phone formats	Ensures correctness of user input
Highlight invalid fields visually	Improves accessibility
Use `HTML5` validation attributes (`required`, `pattern`)	Basic validation without JavaScript

11. Django Database Connectivity (MySQL or SQLite)

(Q1) Connecting Django to a database (SQLite or MySQL).

Ans: Django supports multiple databases like **SQLite**, **MySQL**, and **PostgreSQL**. By default, Django uses **SQLite**, but you can switch to MySQL or others easily.

▯ A. Default: SQLite (Already Configured)

Django uses SQLite by default in `settings.py`:

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

✅No additional configuration needed — great for development.

▯ B. MySQL Setup Example

1. **Install MySQL Client for Python**:

```
pip install mysqlclient
```

2. **Update `settings.py`:**

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'your_db_name',
        'USER': 'your_username',
        'PASSWORD': 'your_password',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

3. **Create the Database** in MySQL manually or via a GUI like phpMyAdmin or MySQL Workbench.
4. **Apply Migrations:**

```
python manage.py migrate
```

(Q2) Using the Django ORM for database queries.

Ans: The **Django ORM** allows you to interact with the database using **Python code** instead of raw SQL.

▯ 1. Define a Model

```
# models.py
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=6, decimal_places=2)
    in_stock = models.BooleanField(default=True)
```

▯ 2. Make Migrations

```
python manage.py makemigrations
python manage.py migrate
```

▯ 3. Query the Database with ORM

Operation	Code Example
Get all records	`Product.objects.all()`
Filter records	`Product.objects.filter(in_stock=True)`
Get one object	`Product.objects.get(id=1)`
Order by field	`Product.objects.order_by('price')`
Create new record	`Product.objects.create(name="Pen", price=2.5)`
Update existing record	`p = Product.objects.get(id=1); p.name = "New"; p.save()`
Delete a record	`Product.objects.get(id=1).delete()`
Count records	`Product.objects.count()`

12. ORM and QuerySets

(Q1) Understanding Django's ORM and how QuerySets are used to interact with the database.

Ans : What is Django ORM?

ORM (Object-Relational Mapper) is a system Django uses to:

- Interact with the database using **Python classes** (models)
- Eliminate the need to write raw SQL
- Automatically map your Python model objects to **relational database tables**

▯ How It Works:

1. **Models (Python classes)** represent tables in the database.
2. Each **instance of a model** is a row in the table.
3. **QuerySets** are collections of model instances (like results from SELECT statements in SQL).

✅Step-by-Step Example

□ 1. Define a Model

```
# models.py
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    published_year = models.IntegerField()
```

This creates a database table like:

id	title	author	published_year
1	"Django Basics"	Alice Brown	2023

□ 2. Create Migrations & Apply Them

```
python manage.py makemigrations
python manage.py migrate
```

□ What is a QuerySet?

A **QuerySet** is a **collection of objects** from your database that you can **filter, search, and manipulate**.

□ Common QuerySet Operations

Task	Code Example	SQL Equivalent
All objects	`Book.objects.all()`	`SELECT * FROM book;`
Filter	`Book.objects.filter(author="Alice Brown")`	`WHERE author = 'Alice Brown'`
Get one	`Book.objects.get(id=1)`	`SELECT * FROM book WHERE id = 1`
Create	`Book.objects.create(title="New", author="X", published_year=2024)`	`INSERT INTO ...`
Order	`Book.objects.order_by('published_year')`	`ORDER BY published_year`

Task	Code Example	SQL Equivalent
Limit	`Book.objects.all()[:5]`	`LIMIT 5`
Count	`Book.objects.count()`	`SELECT COUNT(*)`

▯ Chaining QuerySets

You can chain operations to build complex queries:

```
Book.objects.filter(author="Alice").order_by('-published_year')[:3]
```

This gets the **3 most recent books** by **Alice**.

▯ Update & Delete with ORM

```
book = Book.objects.get(id=1)
book.title = "Updated Title"
book.save()  # UPDATE

Book.objects.get(id=2).delete()  # DELETE
```

✓ Advantages of Django ORM + QuerySets

Benefit	Why It Matters
Database-agnostic	Works with SQLite, MySQL, PostgreSQL, etc.
Safe	Protects against SQL injection
Powerful	Easy filtering, ordering, joins
Pythonic	No need to write raw SQL for most cases

▯ Example: Using QuerySets in a View

```
# views.py
from django.shortcuts import render
from .models import Book

def book_list(request):
    books = Book.objects.filter(published_year__gte=2020).order_by('-
published_year')
    return render(request, 'book_list.html', {'books': books})
```

13. Django Forms and Authentication

(Q1) Using Django’s built-in form handling.

Ans: Django provides a powerful **forms framework** to handle form rendering, validation, and processing.

▯ 1. Creating a Form Class

You can use `forms.Form` or `forms.ModelForm`.

✓Basic Form Example:

```
# forms.py
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)
```

▯ 2. Using the Form in a View

```
# views.py
from django.shortcuts import render
from .forms import ContactForm

def contact_view(request):
    form = ContactForm(request.POST or None)
    if form.is_valid():
        # Process the data (e.g., save to DB, send email)
        print(form.cleaned_data)
        return render(request, 'thank_you.html')
    return render(request, 'contact.html', {'form': form})
```

▯ 3. Template to Render the Form

```
<!-- contact.html -->
<form method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Send</button>
</form>
```

✓`form.as_p` renders form fields in `<p>` tags. You can also use `form.as_table` or manually style each field.

(Q2) Implementing Django’s authentication system (sign up, login, logout, password management).

Ans : Django has a full-featured built-in **authentication system** to handle:

- Sign Up
- Login
- Logout
- Password Change/Reset

□ 1. User Signup View

✓Using Django's `UserCreationForm`:

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

def signup_view(request):
    form = UserCreationForm(request.POST or None)
    if form.is_valid():
        form.save()
        return redirect('login')  # Redirect after successful signup
    return render(request, 'signup.html', {'form': form})
```

□ 2. Login & Logout Views (Using Django Built-ins)

✓Login:

```
# urls.py
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('login/',
auth_views.LoginView.as_view(template_name='login.html'), name='login'),
]
```

✓Logout:

```
urlpatterns += [
    path('logout/', auth_views.LogoutView.as_view(next_page='login'),
name='logout'),
]
```

□ 3. Templates: Login Form

```
<!-- login.html -->
<form method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Login</button>
</form>
```

□ 4. Password Change & Reset

Django includes views for this too!

```
python
```

```
CopyEdit
# urls.py
from django.contrib.auth import views as auth_views

urlpatterns += [
    path('password_change/', auth_views.PasswordChangeView.as_view(),
name='password_change'),
    path('password_reset/', auth_views.PasswordResetView.as_view(),
name='password_reset'),
]
```

14. CRUD Operations using AJAX

(Q1) Using AJAX for making asynchronous requests to the server without reloading the page.

Ans : ▯ **Real-World Use Case:**

Submitting a form or updating part of a page (like a "like" button or live search) without full page reload.

▯ Step-by-Step Example: AJAX POST Request

Let’s say we want to **submit a comment** via AJAX.

▯ 1. Create a View in Django

```
# views.py
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
import json

@csrf_exempt  # Only for demo - better to use CSRF token in production
def submit_comment(request):
    if request.method == 'POST':
        data = json.loads(request.body)
        comment = data.get('comment')
        # Save to database if needed...
        return JsonResponse({'status': 'success', 'message': 'Comment
received!'})
    return JsonResponse({'status': 'fail', 'message': 'Invalid request'},
status=400)
```

▯ 2. Add the URL Pattern

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

urlpatterns = [
    path('ajax/comment/', views.submit_comment, name='submit_comment'),
]
```

▯ 3. HTML + JavaScript AJAX Call

```
<!-- templates/comment_form.html -->
<h2>Leave a Comment</h2>
<textarea id="commentText"></textarea><br>
<button onclick="submitComment()">Send</button>

<p id="response"></p>

<script>
function submitComment() {
    const comment = document.getElementById("commentText").value;

    fetch("/ajax/comment/", {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
            // For security, add CSRF token here in real projects
        },
        body: JSON.stringify({ comment: comment }),
    })
    .then(response => response.json())
    .then(data => {
        document.getElementById("response").innerText = data.message;
    })
    .catch(error => console.error('Error:', error));
}
</script>
```

▯ CSRF Protection for AJAX

In production, use Django's CSRF token:

```
headers: {
    "Content-Type": "application/json",
    "X-CSRFToken": getCookie('csrftoken')
}
```

You can get the CSRF token using JavaScript or include it in a hidden input.

15. Customizing the Django Admin Panel

(Q1)Techniques for customizing the Django admin panel.

Ans: **1. Customize Model Display in Admin List View**

Use `list_display` to show specific fields:

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

class ProductAdmin(admin.ModelAdmin):
    list_display = ('name', 'price', 'in_stock')
```

```
admin.site.register(Product, ProductAdmin)
```

▯ 2. Add Search Functionality

Use `search_fields` to enable admin search bar:

```
class ProductAdmin(admin.ModelAdmin):
    search_fields = ['name', 'description']
```

▯ 3. Add Filters to Sidebar

```
class ProductAdmin(admin.ModelAdmin):
    list_filter = ['category', 'in_stock']
```

▯ 4. Customize Form Layout in Admin

Use `fields` or `fieldsets` to control form appearance:

```
class ProductAdmin(admin.ModelAdmin):
    fields = ['name', 'price', 'category']  # Field order
python
CopyEdit
class ProductAdmin(admin.ModelAdmin):
    fieldsets = (
        ('Basic Info', {'fields': ('name', 'price')}),
        ('Inventory', {'fields': ('in_stock',)}),
    )
```

✓ 5. Inline Editing for Related Models

Let you edit related models directly inside the parent model's admin page.

```
from .models import Product, ProductReview

class ProductReviewInline(admin.TabularInline):
    model = ProductReview
    extra = 1  # Number of empty forms

class ProductAdmin(admin.ModelAdmin):
    inlines = [ProductReviewInline]
```

▯ 6. Add Custom Admin Titles

In `settings.py`, personalize the admin site headers:

```
# settings.py
ADMIN_SITE_HEADER = 'My Custom Admin'
ADMIN_SITE_TITLE = 'MySite Admin'
```

Or in `admin.py`:

```
admin.site.site_header = "MyStore Admin Panel"
admin.site.site_title = "MyStore Admin"
admin.site.index_title = "Welcome to the Admin Area"
```

□□ 7. Control Admin Permissions

Only allow certain actions (e.g., read-only view):

```
class ReadOnlyAdmin(admin.ModelAdmin):
    def has_add_permission(self, request):
        return False

    def has_delete_permission(self, request, obj=None):
        return False
```

16. Payment Integration Using Paytm

Q1) Introduction to integrating payment gateways (like Paytm) in Django projects.

Ans: Integrating a payment gateway in Django allows your web application to accept online payments. Services like **Paytm, Razorpay, Stripe, or PayPal** handle the secure transfer of funds between users and your bank account.

□ How Payment Integration Works:

1. **User fills a payment form** (amount, order details).
2. **Your Django backend** sends a request to the payment gateway API.
3. The **gateway redirects the user** to a payment page (or processes it via API).
4. After success/failure, **the gateway sends a callback/response** to your Django server (usually to a `callback` or `webhook` URL).
5. You **verify the transaction** using the gateway's checksum or token and update your database.

□ Key Steps for Paytm Integration (similar to other gateways):

1. **Create a Paytm Merchant Account**: Get your `MERCHANT_ID`, `MERCHANT_KEY`, and `WEBSITE`.
2. **Install SDK or use API**: Some gateways provide Python SDKs; others require REST API calls.
3. **Generate a checksum**: Required to validate request integrity (Paytm uses a checksum hash).
4. **Redirect to Paytm payment page**.
5. **Handle response at a callback URL**.

� Folder Setup (Simplified):

```
myproject/
├── payments/
│   ├── views.py          # payment initiation and callback
│   ├── urls.py
│   └── templates/
│       └── paytm_form.html
```

� Libraries/Tools You Might Use:

- `paytmchecksum` (for checksum generation/verification)
- `requests` (for API calls if needed)
- Django views and CSRF handling

� Security Tips:

- Always verify payment success from the **server-to-server** callback.
- Never trust client-side success messages.
- Use HTTPS and keep your API keys secure.

17. GitHub Project Deployment

(Q1) Stepsto push a Django project to GitHub.

Ans: Pushing your Django project to GitHub allows version control, backup, and team collaboration. Follow these steps:

� 1. Initialize Git Repository

Open your Django project folder in the terminal and run:

```
git init
```

� 2. Create a `.gitignore` File

Prevent sensitive files (like database, secrets, migrations, etc.) from being tracked:

```
touch .gitignore
```

Recommended `.gitignore` for Django:

```
__pycache__/
*.pyc
db.sqlite3
.env
*.log
/media/
/staticfiles/
/venv/
```

□ 3. Add Files to Git

```
git add .
git commit -m "Initial commit"
```

□ 4. Create a GitHub Repository

- Go to https://github.com
- Click **“New”** to create a new repository
- Don’t initialize with a README (you already have one locally)

□ 5. Connect Local Repo to GitHub

Copy the GitHub remote URL and run:

```
git remote add origin https://github.com/your-username/your-repo-name.git
```

□ 6. Push to GitHub

```
git branch -M main
git push -u origin main
```

✅Done!

Your Django project is now hosted on GitHub.

18. Live Project Deployment (PythonAnywhere)

(Q1) Introduction to deploying Django projects to live servers like PythonAnywhere.

Ans : Deploying a Django project to a live server like **PythonAnywhere** makes your web application accessible to the public via the internet. **PythonAnywhere** is a beginner-friendly platform that supports Django out-of-the-box with minimal setup.

▯ Why Use PythonAnywhere?

- **Free tier** available for small projects
- No server setup needed — Python and WSGI support preconfigured
- Built-in database and file manager
- Easy integration with GitHub

▯ Basic Steps to Deploy on PythonAnywhere:

1. **Create an account** at https://www.pythonanywhere.com
2. **Upload your Django project**
   - Option 1: Clone it from GitHub
   - Option 2: Manually upload files via file manager or `scp`
3. **Set up a virtual environment** (recommended for dependency isolation)
4. **Install requirements**
   In Bash console:

   ```
   pip install -r requirements.txt
   ```

5. **Configure the WSGI file**
   Set the path to your `project.settings` file so PythonAnywhere knows how to load your Django app.
6. **Set up a web app** in the **Web** tab
   - Choose **Manual Configuration**
   - Set your source code directory and virtualenv path
   - Enter your WSGI config file path
7. **Set up static and media file paths**
   In the Web tab, under "Static files":
   - URL: `/static/` → Path: `/home/yourusername/yourproject/static`
   - URL: `/media/` → Path: `/home/yourusername/yourproject/media`
8. **Run Migrations and Collect Static Files**

   ```
   python manage.py migrate
   python manage.py collectstatic
   ```

9. **Reload the web app** on the PythonAnywhere dashboard.

▯ Important Notes

- Use **`DEBUG = False`** in production
- Set `ALLOWED_HOSTS = ['yourusername.pythonanywhere.com']`
- Keep your `SECRET_KEY` and other sensitive data secure (use `.env` files)

19. Social Authentication

(Q1) Setting up social login options (Google, Facebook, GitHub) in Django using OAuth2.

Ans: Social login lets users sign in with existing accounts from providers like Google, Facebook, or GitHub. This is commonly done via the **OAuth2** protocol, which securely authorizes third-party apps without sharing passwords.

How to Implement Social Login in Django?

The easiest way is to use the popular **`django-allauth`** package, which supports multiple providers and handles OAuth2 flows seamlessly.

Step-by-Step Setup Using `django-allauth`

1. Install django-allauth

```
pip install django-allauth
```

2. Update `settings.py`

Add required apps:

```
INSTALLED_APPS = [
    # Django apps
    'django.contrib.sites',

    # Allauth apps
    'allauth',
    'allauth.account',
    'allauth.socialaccount',

    # Providers you want, e.g.:
    'allauth.socialaccount.providers.google',
    'allauth.socialaccount.providers.facebook',
    'allauth.socialaccount.providers.github',
]

SITE_ID = 1

AUTHENTICATION_BACKENDS = (
    "django.contrib.auth.backends.ModelBackend",
    "allauth.account.auth_backends.AuthenticationBackend",
)

# Optional allauth settings:
ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_USERNAME_REQUIRED = False
ACCOUNT_AUTHENTICATION_METHOD = 'email'
LOGIN_REDIRECT_URL = '/'
```

3. Update URLs

```
# urls.py
from django.urls import path, include

urlpatterns = [
    # ...
    path('accounts/', include('allauth.urls')),
]
```

4. Add Social App Credentials

- Go to Django admin panel → **Social applications**
- Add new entries for each provider (Google, Facebook, GitHub)
- Provide **Client ID**, **Secret Key**, and select the site (usually `example.com` or your domain)

▯ Register OAuth Apps on Providers

- **Google**: Google Cloud Console → OAuth 2.0 Client IDs
- **Facebook**: Facebook Developers → Create App → Facebook Login
- **GitHub**: GitHub Developer Settings → OAuth Apps

Set callback/redirect URLs to:

```
http://yourdomain.com/accounts/google/login/callback/
http://yourdomain.com/accounts/facebook/login/callback/
http://yourdomain.com/accounts/github/login/callback/
```

▯ How It Works:

- User clicks **Login with Google/Facebook/GitHub** button
- Redirected to provider login page
- Upon success, redirected back to your site
- Django `allauth` handles authentication & account creation

Optional: Customize Login Buttons in Templates

```
{% load socialaccount %}
{% providers_media_js %}
<a href="{% provider_login_url 'google' %}">Login with Google</a>
<a href="{% provider_login_url 'facebook' %}">Login with Facebook</a>
<a href="{% provider_login_url 'github' %}">Login with GitHub</a>
```

20. Google Maps API

(Q1) Integrating Google Maps API into Django projects.

Ans: Google Maps API lets you embed interactive maps, geolocation features, and place info into your Django web apps.

🚀 Basic Steps to Integrate Google Maps in Django

1. **Get a Google Maps API Key**

- Go to Google Cloud Console
- Enable **Maps JavaScript API**
- Create credentials to get your API key

2. **Add the API Key to Your Django Project**

Store it securely in your `settings.py` or environment variables.

```
# settings.py
GOOGLE_MAPS_API_KEY = 'YOUR_API_KEY'
```

3. **Create a Template with Google Maps Script**

Example template to show a simple map centered on some coordinates:

```
<!DOCTYPE html>
<html>
<head>
  <title>Google Map</title>
  <script
    src="https://maps.googleapis.com/maps/api/js?key={{ google_maps_api_key
}}&callback=initMap"
    async defer></script>
  <style>
    #map {
      height: 400px;
      width: 100%;
    }
  </style>
</head>
<body>
  <h3>My Google Map</h3>
  <div id="map"></div>

  <script>
    function initMap() {
      const center = { lat: 40.7128, lng: -74.0060 }; // New York example
      const map = new google.maps.Map(document.getElementById("map"), {
        zoom: 12,
        center: center,
      });
      const marker = new google.maps.Marker({
        position: center,
        map: map,
      });
    }
  </script>
</body>
</html>
```

4. **Pass API Key from View to Template**

```
# views.py
from django.shortcuts import render
from django.conf import settings

def map_view(request):
    context = {'google_maps_api_key': settings.GOOGLE_MAPS_API_KEY}
    return render(request, 'map.html', context)
```

5. Configure URL

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

urlpatterns = [
    path('map/', views.map_view, name='map'),
]
```

□ Optional Enhancements:

- Add multiple markers from your database
- Use Places API for autocomplete search
- Display routes or polygons
- Use AJAX to load/update map dynamically