Module 16) Python DB and Framework

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

       In web development with Python, frameworks such as **Flask** and **Django** allow developers to create dynamic web pages by **combining Python logic with HTML templates**. Rather than mixing HTML directly inside Python files (which is not good practice), these frameworks use **template engines** to embed dynamic content into HTML.

    b. Generating dynamic HTML content using Django templates.

       Django uses a **template engine** to produce dynamic HTML pages. Instead of writing static HTML files, Django allows developers to embed **variables**, **loops**, **conditions**, and **template tags** inside HTML. When a user requests a page, Django renders the template with the required data, producing a fully dynamic webpage.

2. CSS in Python:
    a. Integrating CSS with Django templates.

       Integrating CSS with Django templates involves using Django's static file system, which provides an organized and efficient way to manage and include stylesheets.
       By keeping CSS in dedicated static directories and referencing them through template tags, Django ensures that web pages are consistently styled, maintainable, and easy to deploy.

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

       Static files include **CSS**, **JavaScript**, **images**, and other resources that do not change dynamically. Django provides a dedicated system called **django.contrib.staticfiles** to manage and serve these files efficiently during development and production.
       Serving static files in Django involves using its built-in static file management system. This system organizes frontend resources, ensures they are correctly

referenced in templates, and allows Django to serve them during development while delegating static file delivery to a proper web server in production. This approach keeps projects structured, efficient, and scalable.

3. JavaScript with Python:
    a. Using JavaScript for client-side interactivity in Django templates.

       Django is a **server-side** framework, meaning it handles backend logic such as processing requests, accessing databases, and rendering templates.
       However, the **frontend** of a Django application—what runs inside the user's browser—can be made interactive using **JavaScript**.
       JavaScript in Django templates performs the same role as in any web project:
       - Manipulating page content dynamically
       - Handling user events
       - Validating forms
       - Making asynchronous requests (AJAX)
       - Creating animations, effects, and UI interactions

    b. Linking external or internal JavaScript files in Django.

       JavaScript files are treated as **static files**, just like CSS or images. Django provides a built-in system to organize, locate, and serve these static resources. JavaScript can be added to templates either **internally** (within the template itself) or through **external JavaScript files** stored in Django's static directories.
       Linking JavaScript in Django—whether internal or external—is done through Django's structured static file system. Internal JavaScript is written directly in templates for small tasks, while external JavaScript files are stored in static directories and linked through Django templates for maintainability and reuse. Django's static handling ensures proper paths, efficient delivery, and clean separation of frontend and backend responsibilities.

4. Django Introduction:
    a. Overview of Django: Web development framework.

       Django is a **high-level, open-source web development framework** written in Python. It is designed to help developers build secure, scalable, and maintainable web applications quickly. Django follows the **MTV architecture**

(Model–Template–View), which promotes clean separation of concerns between data, business logic, and presentation.
Django is a powerful, fully-featured web framework that simplifies building secure, scalable, and maintainable web applications. With its MTV architecture, built-in components, robust security features, and “batteries included” philosophy, Django remains one of the most popular and effective frameworks for Python-based web development.

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

Django is a widely used, high-level Python web framework known for its efficiency, reliability, and strong design philosophy. It offers several advantages that make it a preferred choice for building modern web applications.
Django offers powerful advantages such as **scalability**, **security**, **rapid development**, and **clean architecture**, making it one of the most reliable and efficient frameworks for building modern web applications. Its strong community support, built-in tools, and emphasis on best practices make Django suitable for both beginners and professional developers.

c. Django vs. Flask comparison: Which to choose and why.

Django and Flask are two popular Python web frameworks, but they differ significantly in philosophy, features, and use cases. Understanding these differences helps developers select the right framework for their project.

- **Django** is a powerful, full-featured framework suited for large, data-driven, secure applications.
- **Flask** is minimalistic and flexible, ideal for small apps, experimentation, and services requiring customization.

5. Virtual Environment:

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

A **virtual environment** is an isolated workspace that allows Python projects to have their own independent libraries, packages, and dependencies. It ensures that each project runs in a controlled environment without interfering with other projects on the same system.
Using virtual environments is considered a best practice in Python development because it provides structure, consistency, and reliability.

A virtual environment is essential for Python development because it:

- Manages dependencies
- Prevents version conflicts
- Keeps the system clean
- Makes projects reproducible
- Improves security and stability
- Simplifies deployment

b. Using venv or virtualenv to create isolated environments.

In Python development, **isolated environments** are essential to prevent conflicts between project dependencies. Tools like **venv** and **virtualenv** allow developers to create separate environments for each project so that libraries and versions do not interfere with each other.
venv and virtualenv are essential tools for Python developers. They help create **isolated environments** where each project maintains its own dependencies, avoiding version conflicts and ensuring stable, reproducible development. While venv is built-in and suitable for most use cases, virtualenv offers added flexibility for advanced scenarios.

6. Project and App Creation:

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

Django follows a structured approach where a **project** acts as the main container and **apps** represent different functional modules within that project. Each app focuses on a specific purpose such as authentication, blog management, or product handling.
Steps to Create a Django Project:

1. Set up virtual environment & install Django.
2. Use Django's command to create a project.
3. Review the generated project structure.
4. Optionally run the development server to test the setup.

Steps to Create a Django App:

1. Use Django's command to create an app.
2. Register the app inside INSTALLED_APPS.
3. Create models, views, templates, and URLs within the app.
4. Apply migrations if models are created.

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

   Django follows a modular structure where different files handle different responsibilities. Three core files—**manage.py**, **urls.py**, and **views.py**—play essential roles in project management, request handling, and application logic.

   - Manage.py - manage.py is a command-line utility automatically created when a Django project is initialized.
     It helps developers interact with the Django project in various ways.
   - Urls.py - urls.py defines how URLs of the website are mapped to views. It acts as the **navigation system** of a Django application.
   - Views.py - views.py contains **view functions or class-based views** that define what happens when a user visits a specific URL.

7. MVT Pattern Architecture:
   a. Django's MVT (Model-View-Template) architecture and how it handles request-response cycles.

      Django follows the **MVT architecture**, which is similar to MVC but tailored to Django's design principles.
      MVT separates the application into three core components:

      1. Model –
         - Represents the structure of the data used in the application.
         - Defines fields, relationships, and behaviors.
         - Interacts with the database through Django's **ORM (Object Relational Mapper).**
         - Automatically generates database queries without writing SQL.
      2. View –
         - Contains Python functions or class-based views.
         - Receives HTTP requests and decides what data to send back.
         - Interacts with the Model to fetch, update, or process data.
         - Selects the appropriate template and prepares context data.
      3. Template –
         - Handles the **visual presentation** using HTML.
         - Uses Django's template language for dynamic content (loops, variables, conditions).
         - Displays data passed from the view.

8. Django Admin Panel:
    a. Introduction to Django's built-in admin panel.

        Django provides a powerful, built-in **admin panel** that allows developers and authorized users to manage the application's data through a user-friendly web interface. It is one of Django's most popular features and highlights its "batteries-included" philosophy.
        The admin panel is automatically generated based on the project's models, which makes database management simple and fast without writing additional code.
        Django's built-in admin panel is a powerful tool that provides an automatically generated, secure, and customizable interface for managing application data. It significantly speeds up development, supports efficient data management, and reduces the need to build custom backend tools early in a project. Because of its ease of use and strong feature set, the admin panel is one of the standout advantages of Django.

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

        Django's built-in admin panel offers a powerful, auto-generated interface for managing database records. However, Django also provides extensive options to **customize the admin panel**, allowing developers to control how data is displayed, searched, filtered, and edited. These customizations improve usability and help administrators manage data efficiently.
        Customizing the Django admin interface enhances the default functionality by improving usability, efficiency, and control over database management. Through features such as custom model display options, inline editing, bulk actions, form customization, and user permissions, Django enables developers to build a highly effective and tailored admin environment. This makes the admin panel one of the most powerful tools for managing project data.

9. URL Patterns and Template Integration:
    a. Setting up URL patterns in urls.py for routing requests to views.

        In Django, the file **urls.py** plays a critical role in the request–response cycle. It determines **which view function** should handle a user's request based on the URL they visit. This process is known as **URL routing** or **URL mapping.**
        The urls.py file acts as Django's navigation system, connecting URLs (entered by the user) to the appropriate views (which generate responses).

Setting up URL patterns in urls.py is fundamental to how Django routes requests to the correct views. URL patterns define the structure of a web application's navigation and ensure clean, maintainable, and scalable routing. By connecting URLs to views, Django enables a smooth request–response cycle and organized project architecture.

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

   In Django's MVT architecture, **views** handle the logic of processing requests, while **templates** manage the visual presentation. Integrating these two components allows Django to generate **dynamic HTML pages**, where content changes based on data or user interactions.
   Integrating templates with views is essential for rendering dynamic HTML content in Django. Views gather and prepare data, while templates display this data in a structured and visually appealing format. This combination enables Django to generate personalized, data-driven web pages while maintaining a clean separation between logic and presentation.

10. Form Validation using Javascript:
    a. Using JavaScript for front-end form validation.

       JavaScript plays a crucial role in enhancing user experience by validating form inputs **directly in the browser** before the data is sent to the server. This process is known as **front-end form validation**.
       Front-end validation ensures that users provide correct and complete information, reducing errors, improving performance, and minimizing unnecessary server requests.
       Using JavaScript for front-end form validation helps ensure that form inputs are correct, properly formatted, and complete before being submitted. It provides immediate feedback, reduces server load, improves efficiency, and enhances the overall user experience. Although essential for user-friendly interfaces, it must always be combined with backend validation for total reliability and security.

11. Django Database Connectivity (MySQL or SQLite):
    a. Connecting Django to a database (SQLite or MySQL).

Django is a **database-driven web framework** that uses databases to store and manage application data. It provides built-in support for multiple database systems and connects to them through its **Object Relational Mapper (ORM)**, which allows developers to interact with databases using Python instead of SQL.
Connecting Django to a database involves configuring database settings and using Django's ORM to manage data. **SQLite** is ideal for beginners and development due to its simplicity, while **MySQL** is suitable for production environments because of its performance and scalability. Django's database abstraction allows developers to work efficiently with either database without changing application logic.

b. Using the Django ORM for database queries.

Django uses an **Object Relational Mapper (ORM)** to interact with databases. Instead of writing SQL statements, developers work with Python classes and objects to perform database operations. The ORM automatically translates Python instructions into SQL queries behind the scenes.
This approach provides convenience, portability, and clean design, making database manipulation much easier.
Django's ORM enables developers to interact with databases using Python code instead of SQL queries. It simplifies data handling by mapping database tables to Python classes and rows to objects. Through features such as QuerySets, field lookups, filtering, aggregations, and CRUD operations, the ORM provides a powerful, efficient, and secure way to manage database data within Django applications.

12. ORM and QuerySets:
    a. Understanding Django's ORM and how QuerySets are used to interact with the database.

Django uses an **Object Relational Mapper (ORM)** to interact with databases in an object-oriented way. Instead of writing raw SQL queries, developers work with Python objects that represent database tables and records.
A key concept within Django's ORM is the **QuerySet**, which represents a collection of database objects retrieved from the database.
Django's ORM provides a powerful and secure way to interact with databases using Python objects instead of SQL. **QuerySets** are a core component of the ORM, representing collections of database records that can be filtered,

ordered, and reused efficiently. With features like lazy evaluation and database independence, QuerySets make data access in Django both efficient and developer-friendly.

13. Django Forms and Authentication:
    a. Using Django's built-in form handling.

       Django provides a powerful **built-in form handling system** that simplifies the process of collecting, validating, and processing user input. This system helps developers create secure, reusable, and well-structured forms without manually handling validation or data cleaning.
       Django forms act as a bridge between **HTML forms**, **user input**, and **backend logic.**
       Django's built-in form handling system provides a structured, secure, and efficient way to manage user input. By using Forms and ModelForms, developers can easily validate data, prevent common security issues, and integrate forms seamlessly with models and views. This system reduces development effort while ensuring reliable and maintainable web applications.

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

       1. Create Django app & Project
       2. User Registration (Sign Up)
          a. Create forms.py
          b. Create View (views.py)
          c. URL Configuration
          d. Signup Template
       3. Login & Logout (Built-in)
          a. Create Login Template
          b. Logout (No Template Needed)
       4. Password Management
       5. Protect Pages (Login Required)
       6. Redirect Settings (Important)

14. CRUD Operations using AJAX:
    a. Using AJAX for making asynchronous requests to the server without reloading the page.

AJAX (Asynchronous JavaScript and XML) is used to send and receive data from the server **in the background without reloading the web page**.

- The browser sends a request to the server using JavaScript (fetch, XMLHttpRequest, or Axios).
- The server processes the request and returns data (usually in **JSON** format).
- The web page updates only the required part dynamically.

15. Customizing the Django Admin Panel:

    a. Techniques for customizing the Django admin panel.

    Django's **admin panel** can be customized to improve **usability, appearance, and control** over how data is managed.

    - Custom ModelAdmin classes - Control how models appear using options like list_display, search_fields, list_filter, and ordering.
    - Custom forms in admin - Use custom ModelForm to add validations, widgets, or readonly fields.
    - Fieldsets & layout customization - Organize fields into sections for better readability.

16. Payment Integration Using Paytm:

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

    Integrating a **payment gateway** in a Django project enables applications to accept **online payments** securely through methods such as UPI, cards, wallets, and net banking. A popular example in India is **Paytm**.

    - Creating an **order or transaction** on the server
    - Sending payment details securely to the payment gateway
    - Redirecting the user to the gateway's payment page
    - Receiving a **callback/response** after payment completion

17. GitHub Project Deployment:

    a. Steps to push a Django project to GitHub.

    1. Create a GitHub repository.
    2. Initialize Git in your Django project.
    3. Create .gitignore file
    4. Check project status

5. Add files to staging
6. Commit the changes
7. Add remote repository
8. Push code to GitHub

18. Live Project Deployment (PythonAnywhere):
    a. Introduction to deploying Django projects to live servers like PythonAnywhere.

       Deploying a Django project means making it **accessible on the internet** so real users can use it instead of running it only on a local machine. One popular beginner-friendly platform for Django deployment is **PythonAnywhere**.
       - Uploading your Django project to the server
       - Setting up a **virtual environment** and installing dependencies
       - Configuring Django settings for production (e.g., DEBUG=False, allowed hosts)
       - Connecting the project to a **web server (WSGI)**
       - Linking static files and database

19. Social Authentication:
    a. Setting up social login options (Google, Facebook, GitHub) in Django using OAuth2.

       Social login allows users to **authenticate using existing accounts** like **Google**, **Facebook**, and **GitHub**, instead of creating a new username and password.
       - Install and configure **django-allauth**
       - Add authentication apps to INSTALLED_APPS
       - Configure OAuth credentials (Client ID & Secret) from each provider
       - Set redirect/callback URLs
       - Enable social providers in Django admin
       - Use provided login buttons in templates

20. Google Maps API:
    a. Integrating Google Maps API into Django projects.

Integrating **maps and location services** into a Django project allows you to display maps, markers, and location-based data such as addresses, routes, or nearby places. This is commonly done using the **Google Maps Platform**.

- Enable Maps API in Google Cloud
- Store the API key securely in Django settings or environment variables
- Pass location data from Django views to templates
- Embed Google Maps script in HTML templates
- Add markers, info windows, or routes using JavaScript