Module 9-Python DB and Framework

1. HTML in Python:

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

- Web frameworks like Django and Flask allow Python programs to generate dynamic web pages.
- They use template engines (Django Template Language or Jinja2) to combine Python data with HTML.
- HTML files contain placeholders that are filled with values passed from Python code.
- This method keeps design (HTML) and logic (Python) separate, making development easier.
- It helps create interactive, data-driven websites without writing static HTML for every page.

2.Generating dynamic HTML content using Django templates.

- Django uses the Django Template Language (DTL) to create dynamic HTML pages.

- Python views send data to templates through a context dictionary.

- Templates display this data using {{ variables }} and perform simple logic like loops and conditions.

- This allows HTML pages to change dynamically based on database values or user input.

- It helps keep the web app clean by separating business logic in views and presentation in templates.

2. CSS in Python:

1.Integrating CSS with Django templates.

- Django allows adding CSS to templates using the static files system.
- CSS files are stored in a static folder inside the app or project.
- In the HTML template, you load static files using {% load static %} at the top.
- Then you link your CSS file with <link rel="stylesheet" href="{% static 'css/style.css' %}">.
- This separates styling from HTML and makes the webpage design clean and reusable.

Folder Structure (Example)

```
myproject/
|
 ├── myapp/
|    ├── static/
|   |   └── css/
|   |       └── style.css
|   ├── templates/
|   |   └── home.html
|   └── views.py
|
├── myproject/
|   └── settings.py
└── manage.py
```

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

- **Django has a built-in static files app** (django.contrib.staticfiles) that manages CSS, JS, and images.
- Every Django app can have a **static/** folder inside it to store files:

  ```
  myapp/
  └── static/
      └── myapp/
          └── style.css
  ```
- In **settings.py**, Django uses:

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

  This tells Django how static files will appear in the browser URL.

- You can also define a **STATICFILES_DIRS** list for project-level static files (optional):

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

3. JavaScript with Python:

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

- JavaScript runs in the browser and makes web pages interactive without reloading.
- In Django templates, JavaScript is added through static files using: **<script src="{% static 'js/script.js' %}"></script>**
- Django provides the HTML structure, while JavaScript handles actions like button clicks, form validation, animations, etc.
- JavaScript can also use Django template data (like variables) to update the page dynamically.
- Together, Django + JavaScript create fast, user-friendly, and interactive web applications.

2.Linking external or internal JavaScript files in Django.

1. Django uses the static files system to include internal JS files stored in the static/js/ folder.

2. In the template, load static files using:

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

3.Link an internal JS file like this:

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

4.External JavaScript (like CDN links) can be added directly in the template:

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

5.Both internal and external scripts help add interactivity to Django pages.

4. Django Introduction:

1. Overview of Django: Web development framework.

2. Django is a **high-level Python web framework** used to build secure and scalable web applications.
3. It follows the **MVT architecture** (Model–View–Template) for clean separation of data, logic, and UI.
4. Django provides many built-in features like authentication, admin panel, ORM, and form handling.
5. It helps developers build websites **faster** by reducing repetitive tasks (“batteries included”).

6. Django is suitable for small to large applications like e-commerce sites, CRMs, and social platforms.

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

- **Highly Scalable** – Handles large traffic and big projects efficiently.
- **Secure** – Protects against common attacks (SQL injection, XSS, CSRF).
- **Fast Development** – Built-in features reduce coding time ("batteries included").
- **Reusable Code** – Apps and components can be reused across projects.
- **Strong ORM** – Easy interaction with databases without writing SQL manually.
- **Large Community** – Plenty of documentation, tutorials, and third-party packages.
- **Automatic Admin Panel** – Saves time by providing a ready-made backend interface.
- **Clean Architecture (MVT)** – Keeps project structured and maintainable.

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

- **Django** is a full-featured framework with built-in tools (admin, authentication, ORM).

- **Flask** is a lightweight micro-framework that gives more freedom and flexibility.

- Choose **Django** if you want:
  - Faster development
  - Large projects
  - Built-in features
  - Strong security

- Choose **Flask** if you want:
  - Full control over components
  - Small/medium apps
  - Simple, minimal setup

In short:

- **Django = "All-in-one", best for big applications**
- **Flask = "Do-it-yourself", best for small/custom applications**

5. Virtual Environment Theory:

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

1. A virtual environment keeps project libraries **separate** from the system's global Python installation.
2. It allows each project to have its **own versions** of packages without conflicts.
3. Helps maintain **clean and organized** dependencies.
4. Prevents errors when different projects need **different library versions**.
5. Makes the project easier to **share and deploy**, since dependencies are clearly isolated.

2.Using venv or virtualenv to create isolated environments.

1. venv (built-in) and virtualenv (external tool) are used to create isolated Python environments.

2. They let each project have its own dependencies without affecting other projects.
3. Create an environment using:
4. python -m venv myenv

        or

    virtualenv myenv
5. Activate the environment:
    - Windows: myenv\Scripts\activate
    - Mac/Linux: source myenv/bin/activate
6. After activation, any packages you install will stay inside that environment, keeping the project clean and independent.

6. Project and App Creation:

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

Steps to Create a Django Project and Apps

A. Creating a Django Project

1. Install Django:

    pip install django

2. Create a new project:

    django-admin startproject myproject

3. Go inside the project folder:

    cd myproject

4. Run the development server to test:

    python manage.py runserver

B. Creating an App inside the Project

1. Create an app:

python manage.py startapp myapp

2. Add the app name (myapp) to **INSTALLED_APPS** in settings.py.
3. Create views, templates, URLs, and models inside the app as needed.

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

1. manage.py

- A command-line tool for controlling your Django project.
- Used to run the server, create apps, make migrations, and manage the project.
- Example commands:
  - python manage.py runserver
  - python manage.py startapp appname

```
manage.py
  |
  ├── Starts the development server
  ├── Creates apps
  ├── Runs migrations
  └── Helps manage the entire project
```

2. urls.py

- Handles **URL routing** (decides which page opens for which URL).
- Connects a URL path to a specific **view function**.
- Example: /home → home() view.

3. views.py

- Contains the functions or classes that **process requests** and return **responses**.
- Views decide what data to show and which HTML template to use.
- Example: return a web page when a button/page is clicked.

7. MVT Pattern Architecture:

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

1. MVT = Model–View–Template

1. **Model** – Manages data and database tables.
2. **View** – Contains logic; takes request, processes data, and returns response.
3. **Template** – HTML file that controls how data is displayed to the user.

How Django Handles a Request–Response Cycle

1. User enters a URL in the browser.
2. The request goes to **urls.py**, which decides which **view** should handle it.
3. The **view** interacts with the **model** (if needed) to fetch or save data.
4. The view sends data to a **template** to generate an HTML page.
5. Django sends the final HTML back to the user as a **response**.

When a user visits a page:

1. Django checks the URL
2. Sends request to the correct view
3. View takes data from the model
4. Passes data to the template

5. Template creates the webpage
6. User sees the result in the browser

8. Django Admin Panel Theory:

1.Introduction to Django's built-in admin panel.

- Django provides a **ready-made admin panel** to manage your website's data without writing extra code.
- It is automatically created when you start a new project and run migrations.
- You can log in using a **superuser account** created with:
  - python manage.py createsuperuser
  - The admin panel lets you **add, edit, delete, and view** database records easily.
  - By registering models in admin.py, they become visible and manageable in the admin interface.
  - It saves time and is especially useful for managing content, users, products, or any database data.

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

Customizing the Django Admin Interface

1. You can customize the admin panel to make managing database records easier and more user-friendly.

2. Customization is done inside **admin.py** for each app.

3. You can change how models appear using options like:
    - **list_display** → show selected fields in the list view
    - **search_fields** → add a search bar
    - **list_filter** → filter results by fields
    - **ordering** → set default sorting

4. You can also customize forms, field layouts, and add custom actions.

5. These customizations help admins manage large datasets more efficiently.

9. URL Patterns and Template Integration Theory:

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

Setting Up URL Patterns in urls.py

1. URL patterns tell Django **which view function** should run for a specific URL.

2. You define URL routes inside **urls.py** using the path() function.

3. Each path connects a URL (e.g., /home/) to a view (e.g., home() function).

4. Example in urls.py:

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

urlpatterns = [
    path('home/', views.home),    # When user visits /home/, call home() view
]
```

5. When a user enters a URL, Django checks urls.py, finds the matching path, and sends the request to the mapped view.

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

Integrating Templates with Views to Render Dynamic HTML

1. Django views send data to HTML templates using the render() function.
2. Templates contain placeholders (like {{ name }}) to display dynamic data.
3. In the view, you pass a **context dictionary** to the template.
4. Django combines the template + context to create a dynamic HTML page.

Example

views.py

```
from django.shortcuts import render


def home(request):
    data = {"title": "Welcome to Django"}
    return render(request, "home.html", data)
```

home.html

```
<h1>{{ title }}</h1>
```

✓ Output: The page will show **Welcome to Django**, coming from Python.

10. Form Validation using JavaScript Theory:

1.Using JavaScript for front-end form validation.

- **What is Front-End Validation?**

  Front-end validation means checking form input **in the browser** using HTML5 or JavaScript **before** the data is sent to Django's backend. It improves user experience by giving **instant feedback**.

- **Why Use JavaScript in Django Forms?**
  - To validate fields immediately (without page reload).
  - To reduce wrong form submissions.
  - To show custom messages and highlight errors.
  - To complement Django's server-side validation.

- **Types of Front-End Validation**
    - **Required field check**
    - **Length check** (min/max characters)
    - **Format check** (email, phone, regex)
    - **Match check** (password & confirm password)
    - **Numeric/range validation**
    - **Live validation** while typing

- **How It Works in Django**

Even though Django handles backend validation using forms.py, we can add JavaScript to the template (.html) to validate fields before submitting.

Example:

```
<form id="myForm" method="POST">
   {% csrf_token %}
   <input type="text" id="name" required>
   <span id="nameError"></span>
<button type="submit">Submit</button>
</form>


<script>
document.getElementById("myForm").addEventListener("submit",
function(e) {
   let name = document.getElementById("name").value;


   if (name.trim() === "") {
      e.preventDefault();
      document.getElementById("nameError").innerText = "Name is
required";
   }
});
</script>
```

Benefits

- Fast validation
- Better user experience
- Reduces wrong submissions
- Custom error messages

Limitations

- User can bypass JavaScript
- Not secure alone
- Must use Django backend validation also

11. Django Database Connectivity (MySQL or SQLite)

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

Connecting Django to a Database (SQLite or MySQL) — Short Theory

1. Default Database (SQLite)

Django comes with SQLite as the default database.
It requires **no setup** and is ideal for beginners and small projects.

How Django uses SQLite:

- When you create a project, Django automatically configures SQLite in settings.py.
- The database file is created as db.sqlite3.

Default settings.py:

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

2. Connecting Django to MySQL

MySQL is used for large or production-level applications.

Steps to connect:

(1) Install MySQL client

```
pip install mysqlclient
```

(2) Configure database in settings.py:

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

(3) Create database in MySQL

```
CREATE DATABASE your_database_name;
```

3. Apply Migrations

After connecting the database (SQLite or MySQL):

python manage.py makemigrations

python manage.py migrate

This creates necessary tables (auth, admin, sessions, etc.).

4. How Django interacts with Database

- Django uses **ORM (Object Relational Mapping)**.
- You write Python classes (models), and Django converts them into SQL tables.
- No need to write SQL man

2.Using the Django ORM for database queries.

Django ORM (Object–Relational Mapping) is a powerful feature of the Django framework that allows developers to interact with databases using Python objects instead of writing raw SQL queries. It acts as a bridge between the database and the application logic.

Concept of ORM

ORM maps:

- Models (Python classes) → Database tables
- Model attributes → Table columns
- Model objects → Table rows

This abstraction enables developers to perform database operations in an object-oriented manner.

Working of Django ORM

1. A developer defines models in models.py.
2. Django converts models into database tables using migrations.
3. ORM translates Python queries into SQL.
4. The database executes SQL and returns results.

5. ORM converts results back into Python objects.

Advantages

- Simplifies database interaction
- Improves code readability and maintainability
- Reduces dependency on database-specific SQL
- Encourages clean and structured code
- Integrates seamlessly with Django views and templates

12. ORM and QuerySets Theory:

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

Django ORM & QuerySets (Short Theory)

- Django ORM (Object-Relational Mapping) allows developers to interact with the database using **Python objects instead of SQL queries**. Database tables are represented as **models**, rows as **objects**, and columns as **fields**.

- A **QuerySet** is a collection of objects retrieved from the database. It represents one or more records and allows operations like filtering, sorting, and updating data.

- QuerySets are **lazy**, meaning the database query is executed only when the data is actually needed. This improves performance.

Common QuerySet methods include:

- all() – fetch all records
- filter() – fetch matching records
- get() – fetch a single record

Django ORM and QuerySets provide a **secure, readable, and database-independent** way to work with data.

13. Django Forms and Authentication:

1.Using Django’s built-in form handling.

Django provides a built-in **forms framework** that simplifies handling user input and form validation. Forms are created using Python classes, which automatically generate HTML form fields and validate submitted data.

There are two main types of forms:

- **Forms (forms.Form)** – used when data is not directly linked to a database
- **ModelForms (forms.ModelForm)** – used when form data is saved to a database model

Django form handling includes:

- Automatic form rendering in templates
- Built-in data validation
- Error handling and security against invalid input

Forms process data through **GET or POST requests**, making user input handling clean, secure, and efficient.

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

Implementing Django’s Authentication System (Short Theory)

Django provides a built-in **authentication system** to manage user accounts securely. It handles **user registration (sign up), login, logout, and password management**.

Key features include:

- **User Model** for storing user information
- **Authentication views** for login and logout
- **Password hashing** for secure storage
- **Session-based authentication**

Sign up is implemented by creating a user and saving it to the database. Login verifies user credentials and creates a session. Logout ends the user session. Password management includes password change and reset functionality.

Django's authentication system is **secure, reusable, and easy to integrate**, reducing the need to build authentication from scratch.

14. CRUD Operations using AJAX:

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

Using AJAX for Asynchronous Server Requests (Short Theory)

AJAX (Asynchronous JavaScript and XML) allows web applications to **send and receive data from the server without reloading the entire page**. It improves user experience by updating only specific parts of a webpage.

In Django applications, AJAX is commonly used to:

- Submit forms dynamically
- Load data in the background
- Update content without page refresh

AJAX uses JavaScript (often with fetch or XMLHttpRequest) to communicate with Django views, which return data in formats like **JSON**. This approach makes web applications **faster, more interactive, and responsive**.

15. Customizing the Django Admin Panel:

1.Techniques for customizing the Django admin panel.

Django provides a powerful **admin panel** that can be customized to manage application data efficiently. Customization is mainly done using the admin.py file.

Common techniques include:

- **Customizing model display** using list_display, list_filter, and search_fields
- **Organizing forms** with fieldsets
- **Adding inline models** to edit related objects on the same page
- **Custom actions** for bulk operations
- **Overriding admin templates** for advanced UI changes

These techniques make the Django admin panel **more user-friendly, organized, and suitable for project-specific needs**.

16. Payment Integration Using Paytm:

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

Payment gateway integration in Django allows applications to **accept online payments securely**. Gateways like **Paytm** act as intermediaries between the user, merchant, and bank.

In a Django project, integration typically involves:

- Registering as a merchant with the payment gateway
- Generating and managing **API keys**
- Sending payment requests from Django views

- Handling **callback/response URLs** to verify transactions

Security features such as **checksum generation, encryption, and HTTPS** ensure safe payment processing. Integrating gateways like Paytm enables Django applications to support **real-time, secure online transactions**.

17. GitHub Project Deployment

1.Steps to push a Django project to GitHub.

Steps to Push a Django Project to GitHub (Short Theory)

1. **Create a GitHub Repository**
   Create a new repository on GitHub without adding files.
2. **Initialize Git in Project Folder**
   Use git init to initialize Git in your Django project directory.
3. **Add .gitignore File**
   Exclude files like venv, __pycache__, db.sqlite3, and .env.
4. **Check Git Status**
   Use git status to see tracked and untracked files.
5. **Add Files to Staging Area**
   Use git add . to stage all files.
6. **Commit Changes**
   Use git commit -m "Initial commit" to save changes.
7. **Connect to GitHub Repository**
   Add the remote repository URL using git remote add origin.
8. **Push Code to GitHub**
   Use git push -u origin main to upload the project.

18. Live Project Deployment (PythonAnywhere):

1.Introduction to deploying Django projects to live servers like PythonAnywhere.

Deploying a Django project to a live server like **PythonAnywhere** makes the application accessible on the internet. PythonAnywhere is a cloud-based hosting platform that supports Python and Django applications.

The deployment process generally involves:

- Uploading the Django project to the server
- Creating a virtual environment and installing dependencies
- Configuring **WSGI** and project settings
- Setting DEBUG = False and allowed hosts
- Setting up the database and static files

Deploying on PythonAnywhere allows developers to **host, test, and share Django applications online** in a secure and scalable environment.

19. Social Authentication Theory:

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

Social login allows users to **sign in using third-party accounts** such as Google, Facebook, or GitHub. OAuth2 is the standard authorization protocol used for secure authentication without sharing passwords.

In Django, social login is commonly implemented using libraries like **Django Allauth** or **Social Auth**.

The process includes:

- Registering the application with the social provider
- Obtaining **Client ID and Client Secret**

- Configuring OAuth settings in Django
- Redirecting users to the provider for authentication
- Handling callbacks and creating user sessions

Social login improves **user convenience, security, and faster registration** in Django applications.

20. Google Maps API Theory:

1.Integrating Google Maps API into Django projects.

Integrating the **Google Maps API** into a Django project allows developers to display interactive maps, locations, and geographic data on web pages.

The integration process involves:

- Creating a project in Google Cloud Console
- Generating a **Google Maps API key**
- Enabling required Maps services
- Adding the API key to Django templates or views
- Displaying maps using JavaScript

Google Maps API helps build **location-based features** such as place search, route display, and markers in Django applications.