**Module – 15**

**Python DB**

**& Framework**

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

**Ans. When we talk about embedding HTML in Python with frameworks like Flask or Django, we usually mean creating web pages dynamically using Python code. Instead of writing static HTML files only, these frameworks let you mix Python with HTML using templates.**

* **In Flask, you put HTML inside template files (usually in a templates/ folder) and then use the Jinja2 template engine. Inside the HTML, you can insert Python variables like {{ name }} or run small loops/conditions. Your Python routes pass data to these templates.**
* **In Django, it’s similar but comes with its own templating system. You define HTML files with placeholders ({{ variable }}) and template tags ({% if %}, {% for %}), and Django fills them in when rendering pages.**

**The main idea:  
Python handles the logic (like fetching data, processing input), while HTML templates handle the presentation. You don’t literally write HTML *inside* Python code—you keep them separate but linked through a template system.**

1. **Generating dynamic HTML content using Django templates.**

**Ans. In Django, generating dynamic HTML is done with the template system. Here’s the idea in simple words:**

* **Views (Python code): Decide what data to send.**
* **Templates (HTML with placeholders): Define *how* the data looks.**
* **Template Tags: Let you insert variables ({{ }}) and logic ({% %}) into HTML.**
* **Python (views.py) passes a dictionary (context) with data.**
* **Django replaces {{ name }} with "Alice".**
* **The {% for %} loop generates list items dynamically.**

1. **Integrating CSS with Django templates.**

**Ans.**

* **Create a static/ folder inside your Django app.**
* **Inside static/, create a subfolder with your app’s name (e.g., myapp/) and add your CSS file (e.g., style.css).**
* **Write your CSS styles in style.css.**
* **In settings.py, make sure this is set:**

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

* **In your HTML template (e.g., home.html), load static files at the very top:**

**{% load static %}**

* **Inside the <head> section of your HTML, link the CSS file:**

**<link rel="stylesheet" href="{% static 'myapp/style.css' %}">**

* **Run the server with python manage.py runserver.**
* **Open your web page in the browser — the CSS styles will be applied.**

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

**Ans. Steps to Serve Static Files in Django**

* **Enable static files in settings.py**

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

**STATICFILES\_DIRS = [BASE\_DIR / "static"]**

* **Create a static folder**

**- You can create a global static/ folder in your project root, OR**

**- Inside each app, create an app\_name/static/app\_name/ folder.**

* **Load static in templates**

**{% load static %}**

* **Link static files in HTML**

**<link rel="stylesheet" href="{% static 'myapp/style.css' %}">**

**Before closing </body> for JavaScript:**

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

* **Then Run The Server : python manage.py runserver**

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

**Ans. Key Idea**

* **Django handles the server-side logic (sending data to templates).**
* **JavaScript adds client-side interactivity (dynamic actions in the browser without reloading the page).**
* **You include JavaScript just like CSS: through Django’s static files system.**

1. **Linking external or internal JavaScript files in Django.**

**Ans. For Internal JS (your own file)**

* **Put your JavaScript file in static/.**
* **myapp/static/myapp/script.js**
* **Inside your template:**
* **{% load static %}**
* **<script src="{% static 'myapp/script.js' %}"></script>**

**For or External JS (like CDN)**

* **Just paste the script tag in your template:**

**<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>**

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

**Ans. Django: Web Development Framework**

* **What it is:  
  Django is a high-level Python web framework that helps developers build secure, scalable, and maintainable web applications quickly.**
* **Philosophy:**
  + **“Batteries included” → comes with built-in tools (authentication, admin panel, ORM, forms, security).**
  + **Don’t repeat yourself (DRY) → encourages reusable and clean code.**

**Key Features**

* **MTV Architecture (Model–Template–View, similar to MVC).**
  + **Model → Handles database (tables, queries).**
  + **Template → Manages presentation (HTML + CSS + JS).**
  + **View → Contains logic (Python functions/classes).**
* **ORM (Object-Relational Mapper)**
  + **Let’s you work with databases using Python objects (no raw SQL needed).**
* **Built-in Admin Interface**
  + **Auto-generated dashboard for managing data**
* **Security**
  + **Protects against SQL injection, XSS, CSRF, and more.**
* **Scalability & Reusability**
  + **Apps are modular → you can reuse them across projects.**

1. **Advantages of Django (e.g., scalability, security).**

**Ans. Advantages of Django**

* **Fast development with built-in tools**
* **Scalable for high-traffic websites**
* **Strong security features (SQL injection, XSS, CSRF protection)**
* **“Batteries included” – comes with admin, authentication, forms, caching**
* **ORM for easy database management**
* **Reusable and maintainable code (modular apps, DRY principle)**
* **Large community and excellent documentation**
* **Cross-platform support (Linux, Windows, macOS)**

1. **Django vs. Flask comparison: Which to choose and why.**

**Ans.**

| **Feature** | **Django** | **Flask** |
| --- | --- | --- |
| **Type** | **Full-stack framework** | **Microframework** |
| **Philosophy** | **“Batteries included” – comes with everything** | **Minimalist – add only what you need** |
| **ORM** | **Built-in** | **Needs extensions (e.g., SQLAlchemy)** |
| **Admin Panel** | **Built-in** | **None by default** |
| **Authentication** | **Built-in** | **Needs extensions** |
| **Templating** | **Django templates** | **Jinja2 (flexible)** |
| **Forms** | **Built-in** | **Needs extensions** |
| **Security** | **Built-in protections (XSS, CSRF, SQL injection)** | **Limited, extensions needed** |
| **Learning Curve** | **Steeper** | **Easier, beginner-friendly** |
| **Scalability** | **High, good for large projects** | **High, lightweight for microservices** |
| **Use Cases** | **Large apps, e-commerce, CMS, social networks** | **Small apps, APIs, prototypes** |
| **When to Choose** | **Rapid development, complex projects, built-in features** | **Small projects, full control, lightweight apps** |

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

**Ans. Importance of a Virtual Environment**

* **Dependency Isolation – Each project can have its own packages and versions without conflicts.**
* **Protects Global Python – Keeps system-wide Python installation clean and safe.**
* **Reproducibility – Using requirements.txt ensures others can recreate the same environment.**
* **Easy Management – Activate/deactivate environments as needed for different projects.**
* **Better Collaboration – Team members can share the same environment setup.**

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

**Ans. Django follows the MVT pattern, which is very similar to MVC (Model–View–Controller).**

* **Model → Handles data and database logic (tables, queries).**
* **View → Contains business logic (Python functions/classes that process requests and return responses).**
* **Template → Manages presentation (HTML, CSS, JS for the frontend).**

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

**Ans. What it is:  
A ready-to-use web interface that Django automatically generates for managing your application’s data.**

**Purpose:**

* **Let’s you add, edit, delete, and view database records without writing SQL queries.**
* **Designed for site administrators, not end users.**

**Key Feature:**

* **Auto-generated from your models (models.py).**
* **Supports authentication and permissions (only staff/admin users can log in).**
* **Search, filter, and manage data easily.**
* **Extensible → you can customize the admin with your own forms, actions, and layouts.**

**Django’s admin panel is a powerful tool for managing application data, built automatically from your models, and saves tons of development time.**

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

**Ans. Django’s admin works out-of-the-box, but customizing it helps you:**

* **Display important fields in lists.**
* **Add search and filter options.**
* **Control which fields can be edited.**
* **Improve usability for managing data.**

**Ways to Customize**

* **Register models with ModelAdmin - control how each model appears in the admin.**
* **list\_display - choose which fields are shown in the list view.**
* **list\_editable - make certain fields editable directly in the list view.**
* **readonly\_fields - prevent specific fields from being edited.**
* **list\_filter - add filters in the sidebar for quick filtering.**
* **search\_fields - enable a search bar to look up records.**
* **ordering - define the default order of records.**
* **fields / fieldsets - control the layout of fields in the edit form.**
* **Custom forms - override forms for validation or special input handling.**
* **Custom admin actions - add batch actions (e.g., “Mark as Published”).**

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

**Ans. Key Points**

* **Path (route, view, name) maps a URL to a view function/class.**
* **Dynamic segments: <int:id>, <str:username>, <slug:slug> etc.**
* **Name allows reverse URL resolution ({% url 'home' %} in templates).**

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

**Ans.**

* **Create a template: Make an HTML file that will serve as the page layout. You can include placeholders (variables) for dynamic content.**
* **Pass data from the view: In your view function, prepare the data you want to display on the template.**
* **Render the template: The view combines the template and the data, producing a complete HTML page.**
* **URL routing: Make sure the URL pattern points to the view so that when a user visits that URL, the rendered template is returned.**

1. **Using JavaScript for front-end form validation.**

**Ans.**

* **Access form elements: JavaScript can detect input fields in a form (like text, email, password).**
* **Listen for events: Use events like submit or input to check data when the user types or tries to submit the form.**
* **Validate data: Check if inputs meet requirements (e.g., required fields are filled, email format is correct, passwords match).**
* **Provide feedback: Show error messages next to fields or highlight them so the user knows what to fix.**
* **Prevent form submission if invalid: JavaScript can stop the form from being sent to the server until all validations pass.**

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

**Ans.**

* **Default Database (SQLite)**
  + **Django comes pre-configured with SQLite, so out of the box, no extra setup is needed.**
  + **Database file is usually db.sqlite3 in your project directory.**
  + **Just run python manage.py migrate to create the necessary tables.**
* **Using MySQL (or another external database)**
  + **Install the MySQL driver: e.g., mysqlclient.**
  + **Update settings.py: Modify the DATABASES section with your database name, username, password, host, and port.**
  + **Run migrations: python manage.py migrate to create tables in MySQL.**
  + **Django supports multiple databases (PostgreSQL, MySQL, SQLite, Oracle).**
    - **All database operations (queries, inserts, updates) are done via Django’s ORM.**
    - **Changing the database doesn’t require changing your models or queries; you just update settings.py.**

1. **Using the Django ORM for database queries.**

**Ans. The Django ORM (Object-Relational Mapper) lets you interact with the database using Python objects instead of raw SQL.**

**Basic Operations:**

* **Create: Add new records by creating a model instance and saving it.**
* **Read: Fetch records with methods like .all(), .filter(), .get().**
* **Update: Modify fields on an object and save it, or use .update().**
* **Delete: Remove objects with .delete().**

**Query Features:**

* **Filters: Use lookups like exact, contains, gt (greater than), lt (less than).**
* **Chaining: Combine multiple filters for complex queries.**
* **Relationships: Follow foreign keys and many-to-many fields directly.**
* **Aggregates & annotations: Perform counts, sums, averages, etc.**

**Advantages:**

* **No need to write SQL directly.**
* **Works across multiple databases (SQLite, MySQL, PostgreSQL, etc.).**
* **Safer (prevents SQL injection).**
* **Easier to maintain and read.**

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

**Ans.**

**Django ORM (Object-Relational Mapper):**

* **The ORM is a layer that lets you interact with the database using Python classes and objects, instead of writing raw SQL.**
* **Each model (class) in Django maps to a database table.**
* **Each instance of a model maps to a row in that table.**

**QuerySets:**

* **A QuerySet is a collection of objects retrieved from the database.**
* **It’s like a list, but it’s built from database queries and can be refined before execution.**
* **QuerySets are lazy → they don’t hit the database until the results are actually needed.**

**Uses of QuerySets:**

* **Retrieving all objects: e.g., all books.**
* **Filtering: Get objects that match certain conditions (like published in 2024).**
* **Single object retrieval: Get a specific object by ID or unique field.**
* **Updating: Change data for one or multiple objects.**
* **Deleting: Remove objects from the database.**
* **Aggregation/Annotation: Summarize data (counts, averages, etc.).**

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

**Ans.**

**Define a Form:**

* **You create a Form class in Python (similar to defining a model).**
* **Each field corresponds to an input in the form (text, email, password, etc.)..**

**Display the Form:**

* **The form can be rendered in a template with automatic HTML generation.**
* **Django takes care of things like labels, widgets, and error messages.**

**Handle Form Submission:**

* **When the user submits, Django collects the data in a request.**
* **You can check if the form is valid (form.is\_valid()).**
* **If valid, process the data (save to database, send email, etc.).**

**Validation & Errors:**

* **Django automatically validates fields (e.g., required, correct format).**
* **Custom validation rules can also be added.**
* **Errors are displayed back in the template without writing extra logic.**

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

**Ans.**

**User Model:**

* **Django comes with a built-in User model (username, password, email, etc.).**
* **You can extend it if you need custom fields.**

**Sign Up (Registration):**

* **Usually implemented using a form.**
* **Can use Django’s UserCreationForm, which already handles creating a user and password hashing.**
* **Once submitted and valid, the user is saved in the database.**

**Login:**

* **Django provides a login view and helper function that checks the entered credentials.**
* **If valid, a session is created, and the user is considered “logged in.”**

**Logout:**

* **A built-in logout view clears the session, removing authentication data.**
* **After logout, the user is anonymous again.**

**Password Management:**

**Django provides views for common password tasks:**

* **Change password (when logged in).**
* **Reset password (via email with token link).**
* **Set password (for new or reset accounts).  
  All these are secure and follow best practices.**

**Access Control:**

* **Use decorators like @login\_required to protect views.**
* **Permissions and groups allow fine-grained access control.**

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

**Ans.**

**What AJAX is:**

* **AJAX = *Asynchronous JavaScript and XML*.**
* **It lets the browser send/receive data from the server without refreshing the page.**
* **Commonly uses JSON instead of XML today.**

**User triggers an event:**

* **(e.g., clicks a button, types in a search box).**

**JavaScript sends a request**

* **JavaScript (often via fetch or jQuery) sends data to the server in the background.**

**Server processes the request**

* **Django handles it with a view and returns JSON or HTML snippet as a response.**

**JavaScript updates the page**

* **The response is used to update only part of the page (e.g., showing search results, updating a cart count).**

**Benefits:**

* **Faster: No full page reloads.**
* **Better UX: Feels more interactive and dynamic.**
* **Efficient: Only needed data is sent/received.**

1. **Techniques for customizing the Django admin panel.**

**Ans.**

**Registering Models with Options:**

* **Instead of just registering a model, you can use a ModelAdmin class to control how it appears.**
* **Options include: list of fields displayed, filters, search fields, ordering, etc.**

**Customizing the List View:**

* **Show selected fields instead of default ones.**
* **Add search boxes and filters in the sidebar.**
* **Define default ordering of records.**
* **Display custom values with methods in ModelAdmin.**

**Customizing the Detail/Edit View:**

* **Control the layout using fields or fieldsets.**
* **Make certain fields read-only.**
* **Use list\_editable to allow editing from the list page.**

**Adding Custom Actions:**

* **Create actions (e.g., "Mark as Published") that can be applied to multiple selected objects at once.**

**Overriding Templates & CSS:**

* **You can override the default admin templates to change the UI.**
* **Add custom CSS or JavaScript for extra styling or interactivity.**

**Custom Widgets & Form Integration:**

* **Replace default widgets with date pickers, rich text editors, or custom inputs.**
* **Use custom ModelForm classes in the admin.**

**Inline Models:**

* **Show related models directly inside the parent model’s admin page (e.g., show comments inline with blog posts).**

**Branding & Theming:**

* **Change site header, title, and index text.**
* **Apply third-party themes like django-grappelli or django-suit for a modern look.**