

Module : Python DB and Framework

❖ **HTML in Python**

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

- ❖ Embedding HTML within Python using web frameworks like **Django** or **Flask** refers to the process of generating dynamic web pages by combining Python backend logic with HTML templates. These frameworks use template engines to pass data from Python code to HTML files, ensuring a clear separation between application logic and presentation. This approach improves code organization, maintainability, and allows the development of interactive and scalable web applications.

2. Generating dynamic HTML content using Django templates.

- ❖ Generating dynamic HTML content using **Django templates** allows web pages to display content that changes according to user input or data from the server. In Django, the view handles the application logic and sends required data to the template. The template then uses placeholders and template tags to insert this data into the HTML structure.

Django templates support features such as variable display, conditional statements, and loops, which help in showing dynamic content like lists, user information, or messages. This method follows the principle of separation of concerns, where Python manages logic and HTML handles presentation, making the application easy to maintain and secure.

❖ **CSS in Python**

1. **Integrating CSS with Django templates**

- Integrating CSS with Django templates is essential for improving the visual appearance and user interface of a web application. Django uses a static files system to manage CSS, JavaScript, and images separately from HTML templates. CSS files are stored in a dedicated static directory and linked to HTML templates using template tags.
- By connecting CSS files to Django templates, developers can apply consistent styling across multiple web pages. This approach supports reusability, as the same CSS file can be used by different templates. Integrating CSS with Django templates also follows the principle of separation of concerns, where design and styling are kept separate from application logic, resulting in cleaner, more maintainable web applications.

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

- In Django, static files such as CSS, JavaScript, and images are managed using the static files framework. These files are

kept separate from templates and Python code to maintain a clean project structure.

Loading Static Files in Templates

Django templates use a special template tag to access static files.

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

❖ JavaScript with Python

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

- JavaScript is used in Django templates to add client-side interactivity such as button clicks, form validation, and dynamic content updates. Django renders HTML on the server, while JavaScript runs in the browser to handle user actions. JavaScript can be included using `<script>` tags or external static files. Django template tags allow passing data safely from the backend to JavaScript.

➤ Example:

```
<button id="btn">Click</button>  
<script>  
  document.getElementById("btn").addEventListener("click",  
function () {  
  alert("Hello from Django!");  
});  
</script>
```

2. Linking external or internal JavaScript files in Django.

In Django, JavaScript files are linked using the static files system. External JavaScript files are stored inside the static folder of an app and loaded in templates using the `{% static %}` template tag. Internal JavaScript can also be written directly inside the HTML template using `<script>` tags. Using external JavaScript files is recommended for better code organization and reusability.

Example (External JS):

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

❖ Django Introduction

1. Overview of Django: Web development framework

Django is a high-level Python web development framework used to build secure and scalable web applications quickly. It follows the Model–View–Template (MVT) architecture pattern. Django provides built-in features like authentication, URL routing, ORM, and admin panel. It emphasizes rapid development and clean, reusable code. Django is widely used for developing dynamic and database-driven websites.

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

Django provides high security by protecting applications from common attacks like SQL injection, XSS, and CSRF. It is highly scalable and can handle large traffic with proper configuration.

Django enables rapid development through its built-in features and reusable components. It includes an automatic admin interface that saves development time. Django also supports clean and maintainable code using its MVT architecture.

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

Django is a full-featured web framework that provides built-in tools like authentication, admin panel, and ORM, making it suitable for large and complex applications. Flask is a lightweight and flexible framework that gives developers more control and is ideal for small or simple projects. Django follows a structured approach, while Flask allows more customization. Choose Django for scalable, secure, and database-driven applications, and Flask for lightweight apps or learning purposes.

❖ Virtual Environment

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

A virtual environment is an isolated Python setup created for a specific project. It allows developers to install and manage project-specific packages without affecting the global Python installation. This helps avoid dependency conflicts between different projects. Virtual environments also make projects easier to maintain and deploy. Overall, they ensure a clean, stable, and reproducible development environment.

2. Using venv or virtualenv to create isolated environments.

venv and virtualenv are tools used to create isolated Python environments for projects. They allow each project to have its own Python interpreter and dependencies, preventing version conflicts. The environment can be activated before installing packages so they remain project-specific. This helps in maintaining clean and consistent development setups.

Example:

```
python -m venv myenv  
myenv\Scripts\activate  
virtualenv myenv
```

❖ Project and App Creation

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

First, install Django and create a new project using the `django-admin startproject` command. Next, move into the project directory and run the development server to check setup. To create an app, use the `python manage.py startapp` command. Finally, register the app in `INSTALLED_APPS` inside `settings.py`.

Example:

```
django-admin startproject myproject  
  
cd myproject  
  
python manage.py startapp myapp  
  
python manage.py runserver
```

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

- `manage.py` is a command-line utility used to run and manage Django project tasks like starting the server and creating apps. `urls.py` is responsible for mapping URLs to specific views in the application. `views.py` contains the logic that handles user requests and returns responses such as web pages or JSON data. Together, these files control how requests are processed in a Django project.

❖ **MVT pattern Architecture**

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

Django uses the Model–View–Template (MVT) architecture to organize web applications. The Model manages the database and handles data operations. The View processes user requests and contains application logic. The Template is responsible for displaying data as HTML to the user. When a

request is made, Django maps the URL to a view. The view interacts with the model and renders a template. The final response is then sent back to the user's browser.

❖ Django Admin Panel

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

- Django provides a powerful built-in admin panel that allows developers to manage application data easily. It is automatically generated and requires minimal configuration. The admin panel supports creating, updating, deleting, and viewing database records. It includes authentication and permission management by default. Models can be registered to appear in the admin interface. This feature saves development time and simplifies backend management.

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

- Django allows customization of the admin interface to manage database records efficiently. Developers can modify how models are displayed by registering them with custom admin classes. Fields can be listed, searched, filtered, and ordered in the admin panel. Forms and layouts can also be customized for better usability. Permissions can be controlled for different users. This makes the admin panel more user-friendly and powerful for data management.

❖ URL Patterns and Template Integration

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

- In Django, urls.py is used to define URL patterns for routing requests to views. Each URL pattern maps a specific URL path to a corresponding view function or class. Django checks the URL patterns in order and finds the first match. The path() or re_path() function is used to define routes. Named URLs help in referencing paths easily within templates. This system controls how user requests reach the correct views.

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

- In Django, templates are used to generate dynamic HTML pages. Views collect data from models or other sources and pass it to templates as context. The render() function is used to combine a template with context data. Templates use Django template language to display variables and logic. This separation keeps business logic and presentation separate. As a result, dynamic and reusable web pages are created efficiently.

❖ **Form validation using JavaScripts**

1. Using JavaScript for front-end form validation.

JavaScript is used for front-end form validation to check user input before submitting the form. It helps ensure fields like name, email, and password are filled correctly. Validation reduces server load by catching errors early. JavaScript can display instant error messages to users. Event listeners are used to validate forms on submit. This improves user experience and data accuracy.

❖ **Django Database**

Connectivity(MySQL)

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

Django can be connected to a MySQL database using PyMySQL as the database driver. First, PyMySQL is installed using pip. Then, the database configuration is updated in the DATABASES section of settings.py with MySQL credentials. PyMySQL is added as the MySQL client so Django can communicate with the database. After configuration, Django migrations are run to create database tables. This setup allows Django to perform database operations using its ORM.

2. Using the Django ORM for database queries.

- Django ORM (Object Relational Mapper) allows developers to interact with the database using Python code instead of writing SQL queries. It maps database tables to Python classes called models. Using ORM, data can be created, retrieved, updated, and deleted easily. Common methods like `objects.create()`, `objects.filter()`, and `objects.get()` are used for queries. The ORM improves code readability and database security. It also supports multiple databases without changing query syntax.

❖ ORM and QuerySets

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

Django's ORM (Object Relational Mapper) provides an easy way to interact with the database using Python code. It represents database tables as models and rows as objects. A `QuerySet` is a collection of database queries that retrieves objects from the database. `QuerySets` are lazy, meaning queries are executed only when data is needed. They allow filtering, ordering, and slicing of data. This makes database interaction simple, efficient, and database-independent.

❖ Django Forms and Authentication

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

Django provides a built-in form handling system to manage user input easily and securely. Forms are created using Django's forms module and are linked to HTML forms. They automatically handle data validation and error messages. Django forms protect against security issues like CSRF attacks. Cleaned data can be accessed after validation. This system reduces manual work and improves code reliability.

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

- The signup function handles user registration by checking if the email already exists in the database. If the passwords match, a new user is created and stored in the database along with profile image and user type. If the email already exists or passwords do not match, an error message is shown.
- The login function authenticates users by matching email and password from the database. On successful login, user details like email and profile are stored in the session. Based on the user type (buyer or seller), the user is redirected to different pages. If login fails, appropriate error messages are displayed.
- The logout function clears the session data and redirects the user to the login page. This ends the user session securely.

❖ **CRUD Operations using AJAX**

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

AJAX is used to send and receive data from the server without reloading the web page. It allows the browser to communicate with Django views asynchronously. Using AJAX improves user experience by updating only specific parts of the page. JavaScript sends requests using `fetch()` or `XMLHttpRequest`. Django processes the request and returns data, usually in JSON format. This technique is commonly used for forms, search, and live updates

❖ **Customizing the Django Admin Panel**

1. Techniques for customizing the Django admin panel.

- Register models in `admin.py` to display them in the admin panel
- Use `list_display` to show specific fields in the list view
- Use `search_fields` to enable searching of records
- Use `list_filter` to filter data easily
- Customize form layout and fields
- Control user permissions and access levels

❖ **Payment Integration Using Paytm**

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

- Payment gateways are used to handle online payments securely
- Razorpay can be integrated with Django for processing payments
- API keys provided by Razorpay are used for authentication
- Django views create and manage payment orders
- JavaScript handles payment checkout on the frontend
- Razorpay verifies payment status after completion
- This integration enables safe and reliable online transactions

❖ **Live Project**

Deployment(PythonAnywhere)

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

- Deployment means making a Django project accessible on the internet
- PythonAnywhere is a cloud platform used to host Django applications
- The Django project is uploaded to the server
- Virtual environment is created on PythonAnywhere
- Project dependencies are installed using pip
- Web app and WSGI file are configured
- After setup, the Django site becomes live for users

❖ **Social Authentication**

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

- Social login allows users to sign in using Google, Facebook, or GitHub accounts
- Django uses OAuth2 protocol for secure third-party authentication
- django-allauth is commonly used to implement social login
- OAuth credentials (Client ID and Secret) are obtained from providers
- Providers are configured in Django settings.py
- URLs and templates are set up for login buttons
- This feature improves user convenience and authentication security

❖ **Google Maps API**

1. **Integrating Google Maps API into Django projects.**

- Google Maps API is used to display maps in Django web applications
- An API key is generated from the Google Cloud Console
- The API key is added to the Django template
- JavaScript is used to load and display the map
- Locations can be shown using markers on the map
- Django views pass location data to templates

- This integration helps in location-based services and navigation