Name : *Soliya* Nirbhay

Python DB and Framework

1. HTML in python

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

- Web frameworks like Django and Flask allow you to build websites using Python.
- To display content on a web page, you need to use HTML.
- These frameworks make it easy to combine Python code (for logic) with HTML code (for design and layout).
- This is done using templates.
- Templates are HTML files that can include dynamic content from Python.
- You use special tags (like {{ }} for variables) to insert Python data into the HTML.
- In Flask, you use render_template() to send data from Python to an HTML file.
- In Django, you use the render() function to connect a view to an HTML template.

- Generating dynamic HTML content using Django templates.

- Django uses templates to create dynamic HTML pages.
- A template is an HTML file that can display data from Python code.
- To show dynamic content, Django uses special tags:
  - `{{ }}` – To show variables (like names, numbers)
  - `{% %}` – To add logic (like if, for loops)
- In the view, we pass data to the template.
- The template shows that data inside the HTML.
- This helps create web pages that change based on the user or data from the database.

2. CSS in Python

- Integrating CSS with Django templates.

- To style your web pages in Django, you can use CSS.
- CSS files are placed in a folder called static.
- Django uses the {% static %} tag to link CSS files in HTML templates.
- Steps:

  Create a static folder in your app.

  Add your CSS file (e.g., style.css) inside it.

  Load the static files in your HTML using:

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

  Link the CSS file in the <head> section:

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

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

- In Django, static files are files like CSS, JavaScript, and images that are used to design and add functionality to web pages.
- These files do not change often, so they are called static.
- To use static files in Django:
  - You must place them in a special folder called static.
  - Django uses the {% static %} tag to include these files in HTML templates.
  - You must also tell Django where to find these files by setting a path in the settings.py file.
- Django automatically serves static files during development (when DEBUG = True).
- In production, you need to set up a web server to serve them.

3. JavaScript with Python

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

- JavaScript is used in Django templates to make web pages interactive on the client side (in the browser).
- It can be used for things like showing alerts, handling button clicks, form validation, or updating content without reloading the page.
- In Django:
  - JavaScript files are stored in the static folder.
  - You load them into your HTML template using the {% static %} tag.
  - The JavaScript code runs in the browser, not on the server.
- By linking JavaScript in Django templates, you can make your web pages more dynamic and interactive, such as responding to user actions without refreshing the page.

- Linking external or internal JavaScript files in Django.

- In Django, you can use JavaScript files to add interactivity to your web pages.
- These files can be either internal (written by you) or external (from a CDN like jQuery or Bootstrap).

- 1. Internal JavaScript Files:
    - Save your .js file (e.g., main.js) in the static folder of your Django app.
    - In your HTML template:

        Load static files:

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

        Link the JS file:

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

- 2. External JavaScript Files:
    - Use the full URL of the JavaScript file (e.g., from a CDN).
    - Directly add in your template:

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

4. Django Introduction

- Overview of Django: Web development framework.

- Django is a high-level Python web framework used to build secure, fast, and scalable web applications.
- It helps developers build websites by providing a structure and reusable components.

- Key Features:
    - MVC Pattern (MVT in Django):

        Django follows the Model-View-Template architecture to separate logic, data, and design.
    - Built-in Admin Panel:

        Automatically provides an admin interface to manage your database.
    - ORM (Object-Relational Mapping):

        Allows you to work with databases using Python code instead of SQL.
    - URL Routing:

        Easily map URLs to views (functions or classes that handle requests).
    - Security:

        Protects against common attacks like SQL injection, XSS, and CSRF.

- Reusable Apps:

  You can build and reuse modular apps across multiple projects.

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

➢ Scalability:

Django can handle large projects and high traffic.

It's used by big companies like Instagram and Pinterest.

➢ Security:

Django protects against common security threats like:

- SQL injection
- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)

➢ Rapid Development:

Django includes many built-in features (like admin panel, forms, authentication), so you can build web apps quickly.

➢ Reusability:

Apps and code written in Django can be reused in other projects, making development efficient.

➢ Built-in Admin Interface:

Django provides an automatic admin panel to manage data, users, and models without writing extra code.

➢ Clean and Organized Code (MVT Architecture):

Django uses the Model-View-Template (MVT) pattern, which helps keep the code clean and separated.

➢ Large Community & Documentation:

Django has strong community support and excellent documentation, making it easier to learn and troubleshoot.

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

Feature	Django	Flask
Type	Full-stack web framework	Micro (lightweight) web framework
Built-in Features	Many (Admin panel, ORM, auth, forms, etc.)	Minimal (You add what you need)
Architecture	MVT (Model-View-Template)	MVC (Model-View-Controller)
Flexibility	Less flexible, follows Django way	Highly flexible, developer's choice

Learning Curve	Moderate	Easy and beginner-friendly
Development Speed	Fast (due to built-in tools)	Depends on developer's setup
Community Support	Large and mature	Large and active

5. Virtual Environment

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

- A virtual environment in Python is a self-contained folder that contains its own version of the Python interpreter and its own set of installed packages.
- It allows developers to create an isolated environment for each Python project.
- This is important because different projects may require different versions of the same packages, and using a virtual environment avoids conflicts between them.
- By using a virtual environment:
    - You can install and manage packages without affecting other projects.
    - You keep your global Python installation clean and stable.
    - It helps ensure that the project works the same way on different systems or when shared with others.

- Using venv or virtualenv to create isolated environments.

- In Python, we use tools like venv or virtualenv to create isolated environments for each project.
- These tools help manage project-specific packages without interfering with the global Python setup.

- venv (Built-in)
    - venv is included with Python 3.3 and above.
    - It creates a folder that contains a separate Python environment.
    - Example:

        python -m venv myenv

- virtualenv (External tool)
    - virtualenv works like venv but supports older versions of Python and has more features.
    - You install it using pip.

- Example:

```
pip install virtualenv
virtualenv myenv
```

6. Project and App Creation

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

➢ Here's a step-by-step guide to creating a Django project and individual apps within the project:

➢ Step 1: Install Django

- If you haven't installed Django yet, use pip:

```
pip install django
```

➢ Step 2: Create a New Django Project

- Run the following command to create a Django project (e.g., myproject):

```
django-admin startproject myproject
```

- Navigate into the project directory:

```
cd myproject
```

- This creates a directory structure like:

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

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

➢ manage.py

It's a command-line utility that helps interact with your Django project.

Responsibilities:

- Runs server: python manage.py runserver
- Applies migrations: python manage.py migrate

- Creates migrations: python manage.py makemigrations
- Creates superuser: python manage.py createsuperuser
- Runs tests: python manage.py test

➢ urls.py

Maps URLs (web addresses) to views.

Responsibilities:

- Routes incoming requests to the correct view function.
- Organizes the navigation paths of your app or project.

➢ views.py

Contains the logic for processing requests and returning responses.

Responsibilities:

- Fetch data from models.
- Render templates (HTML pages).
- Return HTTP responses (like JSON or HTML).

7. MVT Pattern Architecture

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

➢ 1. Model (models.py)

- Handles data and business logic
- Communicates with the database (create, retrieve, update, delete records)
- Each model typically maps to a table in your database

```
class Student(models.Model):

  name = models.CharField(max_length=100)

  roll_no = models.IntegerField()
```

➢ 2. View (views.py)

- Contains the logic of your app
- Fetches data from the model and sends it to the template
- Returns an HTTP response (could be HTML, JSON, redirect, etc.)

```
def show_students(request):
    students = Student.objects.all()
    return render(request, 'students.html', {'students': students})
```

- Template (.html files)
    - Deals with presentation
    - Renders the dynamic data passed by views into HTML

```
<!-- students.html -->
<h2>Student List</h2>
<ul>
 {% for student in students %}
   <li>{{ student.name }} (Roll No: {{ student.roll_no }})</li>
 {% endfor %}
</ul>
```

8. Django Admin Panel

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

- What is Django Admin Panel?

  Django admin is a ready-made web interface that lets you view, add, update, and delete data from your database — without writing any extra code.

- Why is it Useful?
    - Saves time — no need to build your own backend UI
    - Helps manage your app's data easily
    - Only accessible to authorized users (like superusers)

- How to Use It (Steps)
    - Create a Django project and app
    - Run migrations → python manage.py migrate
    - Create superuser → python manage.py createsuperuser

- Run server → python manage.py runserver
- Open admin panel → Go to http://127.0.0.1:8000/admin
- Login using superuser credentials

➢ Show Your Models in Admin

In your app's admin.py file:

```
from .models import Student
admin.site.register(Student)
```

- Customizing the Django admin interface to manage database records.

➢ What is Admin Customization in Django?

Customizing the Django admin interface means changing how your model data is displayed and managed in the admin panel.

➢ Why Customize?

- To show important fields
- To make searching and filtering easier
- To make the admin panel more user-friendly

➢ Basic Customization Options

- You do this in your app's admin.py file using a class like this:

```
class StudentAdmin(admin.ModelAdmin):
    list_display = ('name', 'roll_no')      # shows these columns
    search_fields = ('name',)               # adds a search box
    list_filter = ('course',)               # adds filter options
```

- Then register it:

```
admin.site.register(Student, StudentAdmin)
```

➢ Result:

- A table with name and roll number
- A search box to search students by name
- A sidebar filter by course

9. URL Patterns and Template Integration

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

- In Django, urls.py is used to map URLs (web addresses) to view functions.
- It tells Django what code to run when a user visits a specific URL.
- Without URL patterns, Django wouldn't know which view should handle a request like /home/ or /students/.

- Basic Setup
    - Step 1: Import Required Modules

      from django.urls import path

      from . import views
    - Step 2: Define URL Patterns

      urlpatterns = [

      path('', views.home, name='home'), # root URL

      path('about/', views.about, name='about'), # /about/

      ]

      Here:

      'about/' is the URL path

      views.about is the function that will handle the request

      name='about' lets you refer to this URL in templates

- How It Works

  When a user visits http://127.0.0.1:8000/about/:
    - Django looks in urls.py
    - Finds path('about/', views.about)
    - Calls the about() function in views.py

- Integrating templates with views to render dynamic HTML content.

- Templates are HTML files used to display data dynamically in Django.
- They allow you to combine HTML with Python variables.

- Why Use Templates?
    - To create dynamic webpages
    - To show data from the database (like names, posts, etc.)
    - To separate design (HTML) from logic (Python code)

- How to Integrate Templates with Views
    - 1. Create a Template File

      Inside your app folder, make a folder named templates, then another folder with your app name:

      myapp/

```
└── templates/
        └── myapp/
                └── home.html
```

home.html

```
<h1>Hello, {{ name }}!</h1>
```

- 2. Write a View That Uses the Template

  In views.py:

```
from django.shortcuts import render

def home(request):
    return render(request, 'myapp/home.html', {'name':
    'Alice'})
```

  'myapp/home.html' is the path to the template

  {'name': 'Alice'} is the data passed to the template

- 3. Add URL Pattern

  In urls.py:

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

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

10. Form Validation using JavaScript

- Using JavaScript for front-end form validation.

➢ It's the process of checking if the user's input is correct before the form is submitted to the server.

➢ Why Use JavaScript for Validation?
  - Gives instant feedback to the user
  - Helps avoid unnecessary server requests
  - Improves user experience

➢ Common Checks with JavaScript
  - Required fields are not empty

- Email is valid
- Password is strong enough
- Numbers are in the correct range

➢ Simple Example

- HTML Form:

```
<form onsubmit="return validateForm()">
  Name: <input type="text" id="name"><br>
  Email: <input type="email" id="email"><br>
  <button type="submit">Submit</button>
</form>
```

- JavaScript Validation:

```
<script>
  function validateForm() {
    const name = document.getElementById("name").value;
    const email = document.getElementById("email").value;

    if (name === "") {
      alert("Name is required");
      return false;
    }

    if (!email.includes("@")) {
      alert("Enter a valid email");
      return false;
    }

    return true; // Allow form submission
  }
</script>
```

11. Django Database Connectivity (MySQL or SQLite)

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

➢ Connecting Django to a database is a key step in setting up your project.
➢ By default, Django uses SQLite, but you can switch to MySQL or other databases easily. Here's how you can connect to both:

- Option 1: Using SQLite (default)
    - SQLite is the default database Django uses.
    - It requires no setup—just make sure this is in your settings.py:

```
# settings.py

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

- Option 2: Using MySQL

    - 1. Install MySQL and Python Connector

      Install the MySQL server and the Python MySQL client:

      ```
      pip install mysqlclient
      ```

      If you're on Windows and mysqlclient doesn't work, you can use PyMySQL as an alternative:

      ```
      pip install pymysql
      ```

      And add this in your __init__.py of your main project folder (next to settings.py):

      ```
      import pymysql
      pymysql.install_as_MySQLdb()
      ```

    - 2. Configure settings.py

      Update the DATABASES setting like this:

      ```
      # settings.py

      DATABASES = {
         'default': {
            'ENGINE': 'django.db.backends.mysql',
            'NAME': 'your_database_name',
            'USER': 'your_mysql_user',
            'PASSWORD': 'your_mysql_password',
            'HOST': 'localhost',  # or your database server IP
            'PORT': '3306',
         }
      }
      ```

- Using the Django ORM for database queries.

➢ ORM (Object-Relational Mapping) is a programming technique that allows you to interact with a database using object-oriented code instead of SQL.

➢ In Django, the ORM lets you:
  - Create, retrieve, update, and delete records in the database using Python.
  - Map Python classes to database tables.
  - Avoid writing raw SQL queries.
  - Switch databases (e.g., from SQLite to MySQL) without changing your application logic.

➢ Key Concepts in Django ORM

Model -> A class that defines the structure of a database table.
Field -> A class attribute in a model representing a column in the table.
Object -> A single record in the table, represented as a Python object.
QuerySet -> You get a QuerySet when using queries like all(), filter(), etc.

➢ Read Records

```
Student.objects.all()                 # All records
Student.objects.get(roll_number=1)    # Single record by condition
Student.objects.filter(is_active=True)  # Multiple records by condition
Student.objects.first()               # First record
Student.objects.last()                # Last record
```

12.ORM and QuerySets

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

➢ What is Django ORM?

Django's Object-Relational Mapper (ORM) is a powerful tool that allows you to communicate with your database using Python instead of writing SQL queries.

  - Each model (class) maps to a database table.
  - Each field in the model maps to a column.
  - Each instance of the model represents a row in the database.

- What is a QuerySet?

  A QuerySet is a collection of database records (rows) that match a query.

  It is similar to a list of objects, but it's lazy and optimized for database operations.

- Example: Define a Model

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

class Student(models.Model):
    name = models.CharField(max_length=100)
    roll_number = models.IntegerField(unique=True)
    email = models.EmailField()
    is_active = models.BooleanField(default=True)
```

- Using QuerySets to Interact with the Database

  - 1. Retrieve Data

```
# Import the model
from myapp.models import Student
```

Action	Query	Description
Get all students	Student.objects.all()	Returns a QuerySet of all students
Filter students	Student.objects.filter(is_active=True)	Get students with specific conditions
Get one student	Student.objects.get(roll_number=1)	Returns one object
Check if exists	Student.objects.filter(name="Ali").exists()	Returns True or False

  - 2. Create Data

```
Student.objects.create(name="Ali", roll_number=1,
email="ali@example.com")
```

  - 3. Update Data

```
student = Student.objects.get(roll_number=1)
student.name = "Alicia"
student.save()
```

- 4. Delete Data

```
student = Student.objects.get(roll_number=1)
student.delete()
```

13. Django Forms and Authentication

- Using Django's built-in form handling.

- Forms are used to collect input from users—like name, email, passwords, comments, etc.

- In Django, forms are Python classes that:
  - Create HTML form elements
  - Validate user input
  - Process data easily and securely

- Types of Forms in Django

  forms.Form – For general purpose forms (not linked to models).

  forms.ModelForm – Automatically creates a form from a Django model (saves data directly to the database).

- How Form Handling Works in Django
  - 1. Create the Form Class

    Example using forms.Form:

```
from django import forms

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

    Or using forms.ModelForm (connected to a model):

```
from django import forms
from .models import Student

class StudentForm(forms.ModelForm):
    class Meta:
        model = Student
        fields = ['name', 'roll_number', 'email']
```

- 2. Use the Form in a View

```
from django.shortcuts import render, redirect
from .forms import StudentForm

def student_register(request):
    if request.method == 'POST':
        form = StudentForm(request.POST)
        if form.is_valid():
            form.save()  # Saves to the database
            return redirect('success_page')  # Redirect after saving
    else:
        form = StudentForm()  # Empty form

    return render(request, 'register.html', {'form': form})
```

- 3. Create the HTML Template

```
<!-- register.html -->
<h2>Student Registration</h2>
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Register</button>
</form>
```

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

➢ Django has a built-in system to manage:
- User registration (sign up)
- User login/logout
- Password hashing and checking
- Password change/reset
- Access control (permissions, user roles)

➢ Main Features
- Sign Up (User Registration)

  This allows new users to create an account by entering a username, password, and other details.
- Login

This lets existing users log in to the site using their username and password. Django automatically handles session creation for logged-in users.

- Logout

  This logs the user out and ends their session. It's important for security and privacy, especially on shared devices.

- Password Change

  Logged-in users can change their password through a secure form. Django makes sure the old password is verified before updating.

- Password Reset

  If users forget their password, they can reset it using their email. Django sends a reset link that lets them create a new password.

➢ How It Works

- Django uses a User model to store and manage user details.
- It provides ready-made forms and views for common tasks like login and signup.
- It also handles security (like password hashing and CSRF protection) automatically.
- You can customize how users register or what happens after login/logout.

14.CRUD Operations using AJAX

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

➢ AJAX stands for Asynchronous JavaScript and XML.
➢ AJAX lets your web page talk to the server in the background without refreshing the entire page.
➢ It helps improve user experience by making the web app faster and smoother.

➢ What Can AJAX Do?

- Submit a form without reloading the page
- Load new data (like search results or messages) without refreshing
- Update a part of the page (like a table or a div)
- Check something live (like username availability)

- How AJAX Works in Django
    - User interacts with the page (like clicking a button or submitting a form).
    - JavaScript (AJAX) sends a request to the Django server.
    - Django view processes the request and returns data (usually as JSON).
    - JavaScript receives the data and updates the page dynamically.

- AJAX = Send/receive data without reloading the page
- In Django, AJAX works with views that return JSON responses
- Frontend uses JavaScript or jQuery to make AJAX calls
- Great for fast, dynamic, and user-friendly web apps

15. Customizing the Django Admin Panel

- Techniques for customizing the Django admin panel.

- The Django admin panel is a built-in web interface that allows you to:
    - Add, edit, and delete data from your models
    - Manage users, content, and more
    - Handle all this through a friendly UI

- Ways to Customize the Admin
    - Change List Display
        You can choose which fields show in the list view of your models.
        Example: Instead of showing all fields, only show name and date.
    - Search and Filter
        Add search boxes and filters to quickly find records.
        This is helpful if you have lots of data and want to narrow it down easily.
    - Field Grouping (Fieldsets)
        Group related fields into sections, so forms are cleaner and easier to fill.
        Useful for organizing long forms into neat sections.
    - Read-Only Fields
        Make certain fields non-editable in the admin.
        This is helpful for fields like "created date" or "user ID" that should not be changed.
    - Custom Admin Actions
        You can create buttons that perform bulk actions, like marking multiple items as “active” or “approved.”

This saves time when managing many entries.

- Custom Templates and CSS

  You can even override the admin's default look and feel using your own HTML/CSS.

  This is used when you want the admin to match your company's design.

- Inline Editing

  You can manage related models directly inside the parent model's admin page.

  For example, manage a product and its reviews on the same page.

16. Payment Integration Using Paytm

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

- A payment gateway is a service that processes online payments securely between your website and the customer's bank (or wallet).
- Popular examples include:
  - Paytm
  - Razorpay
  - Stripe
  - PayPal

- Basic Flow of Payment Integration
  - User clicks "Pay Now" on your website.
  - Django app sends payment details (amount, order ID, etc.) to the payment gateway.
  - Gateway redirects user to its payment page.
  - User completes the payment (with card, UPI, etc.).
  - Gateway sends a response back to your Django server (success or failure).
  - You verify the response and update the order status in your database.
- What You Need to Integrate Paytm in Django
  - Merchant Account on Paytm (with API key and merchant ID)
  - A Django view to handle payment requests
  - A callback view to handle the gateway's response
  - Checksum generation & verification (Paytm provides helper code)
  - Optionally: a payment success/failure page

17. GitHub Project Deployment

- Steps to push a Django project to GitHub.

➢ Create a GitHub Repository
  - Go to https://github.com
  - Click “New” to create a new repository
  - Name your repo (e.g., my-django-project)
  - You can leave it empty (no README, .gitignore, etc.)
  - Click “Create repository”

➢ Initialize Git in Your Django Project Folder
  - In your project directory on your computer:

    cd path/to/your/project/

    git init
  - This sets up Git to track your project.

➢ Add Files to Git

  git add .

  This stages all files for commit.

➢ Make Your First Commit

  git commit -m "Initial commit"

  This saves your changes with a message.

➢ Connect to GitHub Repository
  - Use the link provided by GitHub (HTTPS or SSH).
  - Example:

    git remote add origin https://github.com/your-username/my-django-project.git

➢ Push Your Code to GitHub

  git branch -M main # Rename to main if needed

  git push -u origin main

  This uploads your code to GitHub.

➢ Add a .gitignore File
  - To avoid uploading unnecessary files (like migrations, .pyc, or __pycache__), create a .gitignore file:

    # .gitignore

```
*.pyc
__pycache__/
db.sqlite3
.env
/static/
```

- Or use a pre-made Django .gitignore template.
- Then:

```
git add .gitignore
git commit -m "Add .gitignore"
git push
```

18. Live Project Deployment (PythonAnywhere)

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

- Create an Account
  Sign up at pythonanywhere.com

- Upload Your Project
  Use GitHub or manually upload your Django project.

- Set Up Virtual Environment
  Create and activate a virtual environment.
  Install your project's required packages.

- Setup Web App on PythonAnywhere
  Go to the Web tab and create a new web app.
  Choose Manual Configuration and select your Python version.

- Configure WSGI File
  Tell PythonAnywhere where your project is and how to run it.

- Apply Migrations & Collect Static Files
  Run python manage.py migrate
  Run python manage.py collectstatic

- Reload Website
  Click the Reload button in the Web tab.

19. Social Authentication

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

- Use a Library
  We use social-auth-app-django to connect Django with social login providers like Google, Facebook, and GitHub.

- Add Provider Keys
  We register on Google, Facebook, or GitHub to get a Client ID and Secret Key.

- Connect Django with Social Sites
  Add login URLs and settings in Django to use those keys.
  Add social login buttons on your website.

- User Clicks Login
  When the user clicks "Login with Google", they are redirected to Google to give permission.

- Login Success
  If permission is granted, the user is logged into your site.

20. Google Maps API

- Integrating Google Maps API into Django projects.

- Google Maps API lets you display maps, markers, routes, etc., on your website.
- Django shows the map by loading JavaScript from Google Maps in your HTML.
- You need an API key from Google to use the Maps services.

- Get Google Maps API Key
  - Go to Google Cloud Console
  - Create a project
  - Enable Maps JavaScript API
  - Get your API key

- Use the API Key in Your HTML Template
  In your Django template (template.html):
  <!DOCTYPE html>

```
<html>
  <head>
    <title>My Map</title>
    <script
src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY"></script>
    <script>
      function initMap() {
        var location = {lat: 28.6139, lng: 77.2090}; // Example: Delhi
        var map = new google.maps.Map(document.getElementById('map'), {
          zoom: 10,
          center: location
        });
        var marker = new google.maps.Marker({
          position: location,
          map: map
        });
      }
    </script>
  </head>
  <body onload="initMap()">
    <h1>Google Map Example</h1>
    <div id="map" style="height:500px; width:100%;"></div>
  </body>
</html>
```

- Connect This Template in Django View

  - In your views.py:

```
from django.shortcuts import render

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

  - In your urls.py:

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

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