★Module 16: Python DB and Framework

1. HTML in Python

Q1:Introduction to embedding HTML within Python using web frameworks like Django or Flask.

Ans: Embedding HTML within Python is a standard practice in web development frameworks like Django and Flask, enabling dynamic generation of web pages using Python code and HTML templates.

➢ **Overview**

Web frameworks such as Django and Flask allow developers to separate business logic (Python code) from presentation (HTML), while still enabling them to work together through templating engines. These engines let you insert Python variables and logic directly into HTML files.

➢ **How It Works**

- Templates: HTML files embedded with special template syntax (e.g., {{ variable }} in Django or Flask) that allow the inclusion of data dynamically generated by Python code.

- Rendering: The framework processes the template, replaces variables and logic blocks with their evaluated values (using Python), and returns a finished HTML page to the user's browser.

➢ **Example in Django**

Django uses a template engine where HTML files are stored in a templates directory. In your Python view, you pass variables (context) to the template, and inside the HTML you can use:

```
Xml
```

```
<h1>Welcome, {{ user_name }}!</h1>
```

This will be replaced by the actual username provided by your view function.

➢ **Example in Flask**

Flask also uses templates (typically with Jinja2 syntax, similar to Django), where you define HTML files and pass variables in your Python routes:

```
Xml
```

```
<h1>Your balance: {{ balance }}</h1>
```

➢ The variable `balance` is supplied from your Flask view (Python function) and rendered in the browser.

➢ **Benefits**

- Separation of Concerns: Keeps Python business logic separate from presentation while allowing flexible, dynamic content generation.
- Reusability: Templates can be reused with various data sets.
- Maintainability: Clear structure, easier debugging and testing of both Python and HTML.

➢ **Getting Started**

- In Django, use the `render()` function in your views and standard ".html" files with template tags.
- In Flask, use `render_template()` in your routes, placing templates in a "templates" folder.

Q2:Generating dynamic HTML content using Django templates.

Ans: Dynamic HTML content in Django is generated using templates, which combine static HTML with Django's template language to embed dynamic data from Python views.

➢ **Core Principles**

- Templates: Django templates are HTML files with special tags and syntax that allow developers to insert variables, run loops, and use conditional logic.
- Context: Dynamic data is passed from your Python view to the template using a "context" (a dictionary of variables).

➢ **Example Workflow**

1. View Function (Python)

- Define a view that gathers data (from a database, for example), then passes this data as context to the template.
- Example:

```
Python

def post_list(request):
    posts = Post.objects.all()
    return render(request, 'post_list.html', {'posts': posts})
```

Here, `posts` is a queryset of Post objects sent to the template.

2. Template File (HTML with Django Syntax)

- Use double curly braces `{{ }}` for variables and `{% %}` for control structures.

```
Xml

<ul>
     {% for post in posts %}
     <li>{{ post.title }}</li>
     {% endfor %}
</ul>
```

- This will dynamically list all post titles from the data passed by the view.

➢ **Features and Advantages**

- Separation of Logic and Presentation: Templates separate business logic from UI, making code more maintainable.
- Template Inheritance: You can create a base template and extend it for different pages, keeping code DRY and reusable.
- Dynamic Content: You can easily loop over data, insert variables, and use conditional statements to customize output.

➢ **Key Commands and Patterns**

- Use `render(request, 'template.html', context)` in views.
- Place templates in the directory specified in Django's settings.
- Variables in templates are rendered using `{{ variable }}`.
- Control logic (loops, if/else) uses `{% for x in y %}`, `{% if condition %}`.

2. Css In Python

Q1:Integrating Css With Django?

Ans:

1. Create a Static Directory

- Inside your Django app, create a folder named `static` and, within it, a subfolder for CSS—such as `static/css/`.
- Place your CSS file (e.g., `style.css`) in this directory.

2. Configure Static Files in Settings

- Ensure `STATIC_URL` is set in your project's `settings.py` (usually `STATIC_URL = '/static/'`).

3. Reference CSS in Your Template

- At the top of your HTML template, load Django's static files:

```
Xml

    {% load static %}
```

- Link your CSS file in the `<head>` section of your HTML template:

```
xml

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

-
- This tells Django to serve the CSS file when your page loads.

4. Example Template Structure

```
     Xml

 {% load static %}
<!DOCTYPE html>
```

```
<html>

<head>

    <title>Django CSS Example</title>

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

</head>

<body>

    <h1>Hello, styled Django!</h1>

</body>

</html>
```

This connects your `style.css` file to the template and applies styles.

Key Points

- Static files must reside in the designated `static` directory.
- Use `{% load static %}` and `{% static 'path/to/file' %}` to correctly reference assets.
- For deployment, you may need to run Django's `collectstatic` command to gather all static files.

Following this pattern allows Django apps to maintain clean separation of code and design while leveraging reusable, centralized CSS styling.

Q2:How to serve static files (like CSS, JavaScript) in Django.

Ans:

➢ **Static Files Setup**

- Static Directory: Create a `static/` folder in your app or project where CSS, JS, and other static files will be stored.
- App-level and Project-level: Django searches for static files inside each app's `static/` folder and in locations specified by `STATICFILES_DIRS` in `settings.py`.

➢ **Configure Django Settings**

- In `settings.py`, ensure you have:
- `Python`

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

- For extra locations:

  `python`

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

- For deployment, add:

  `Python`

```
STATIC_ROOT = BASE_DIR / "staticfiles"
```

- These settings tell Django where to find and how to serve static files.

➢ **Reference Static Files in Templates**

- Load the static tag library at the top of each template:

  `Xml`

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

- Reference static files like:

  `Xml`

```
<link rel="stylesheet" href="{% static 'css/style.css' %}">
<script src="{% static 'js/app.js' %}"></script>
```

- Place files like `style.css` or `app.js` inside `static/css/` or `static/js/` directories, respectively.

➢ **During Development**

- Django serves static files automatically at `STATIC_URL` when running `python manage.py runserver`.
- In production, configure your web server (e.g., Nginx, Apache) to serve static files, and use `python manage.py collectstatic` to gather all static files in one directory.

3. JavaScript with Python

Q1:Using JavaScript for client-side interactivity in Django templates.

Ans: Using JavaScript in Django templates enables client-side interactivity, enhancing user experience by handling events, updating content dynamically, and validating forms without reloading the page.

1. Place JavaScript Files in Static Directory

- Similar to CSS, store your JS files in the static directory:

  text

```
your_app/
  └── static/
        └── your_app/
              └── js/
                    └── script.js
```

- Add your JavaScript code in `script.js`, for example:

  js

```js
function showAlert() {
  alert("Hello from Django!");
}
```

2. Reference JavaScript in Templates

- Load the static template tag at the beginning of your HTML template:

  text

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

- Inside the `<head>` or right before the closing `</body>`, link to your JS file:

  xml

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

3. Example Usage in Template

xml

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>Django JavaScript Example</title>
    <script src="{% static 'your_app/js/script.js'
%}"></script>
</head>
<body>
    <h1>Click the button for a message</h1>
    <button onclick="showAlert()">Click me!</button>
</body>
</html>
```

When users click the button, the JavaScript function runs and shows a popup alert.

Benefits of JavaScript in Django Templates

- Adds dynamic behavior and responsiveness on the client side.
- Enables asynchronous data fetching with AJAX without full page reloads.
- Facilitates validation and interactive UI elements like modals, tabs, and animations.

Key Points

- Keep JavaScript files in the static folder, referenced with `{% static %}` tag.
- Inline JavaScript can be used but is less maintainable than separate files.
- Use client-side scripts to complement Django's server-side rendering seamlessly.

Q2:Linking external or internal JavaScript files in Django.

Ans: Linking external or internal JavaScript files in Django templates follows a similar pattern as linking CSS files, using Django's static file management system.

1. Place JavaScript Files in Static Directory

- Organize your JS files inside the static folder of your app like this:

```text
your_app/
    └── static/
            └── your_app/
                    └── js/
                            └── script.js
```

- This keeps your JS files managed properly by Django.

2. Load Static and Reference JS in Template

- At the top of your Django HTML template, load the static files template tag:

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

- Link your JavaScript file using the `{% static %}` tag inside a `<script>` tag:

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

- Place this tag in the `<head>` or just before the closing `</body>` for better page load performance.

3. Example of Template Linking JavaScript

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

```
<!DOCTYPE html>
<html>
<head>
    <title>Django with JS</title>
    <script src="{% static 'your_app/js/script.js'
%}"></script>
</head>
<body>
    <h1>JavaScript Integration</h1>
    <button onclick="showAlert()">Click me</button>
</body>
</html>
```

- The `script.js` file should have the JavaScript function like:

  js

  ```
  function showAlert() {
      alert("Hello from Django JavaScript!");
  }
  ```

➢ **Handling Dynamic Data Between Django and JavaScript**

- To pass dynamic data from Django to JS, embed it within `<script>` tags in the template using Django template variables:
- xml

```
<script>
    var data = "{{ my_data|escapejs }}";
</script>
<script src="{% static 'your_app/js/script.js' %}"></script>
```

- Then, your external JS can use the `data` variable without direct templating.

4. Django Introduction

Q1:Overview of Django: Web development framework.

Ans: Django is a high-level, open-source Python web framework designed to facilitate rapid, secure, and maintainable web development. It follows the Model-Template-View (MTV) architectural pattern, which separates data handling (Model), user interface (Template), and business logic (View), helping developers organize code efficiently.

Overview and Key Features

- Batteries-Included Philosophy: Django comes with many built-in features such as an Object-Relational Mapper (ORM) for database interactions, an automatic admin interface for managing data, robust authentication and authorization systems, and form handling utilities. This reduces the need for third-party tools or extensive coding from scratch.
- ORM (Object-Relational Mapping): Allows developers to work with databases through Python objects instead of raw SQL, simplifying data operations while improving security and maintainability.
- Security: Django includes built-in protection against common web vulnerabilities, including SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF), making it a secure choice for web applications.
- URL Routing: It provides flexible and clean URL dispatcher patterns, allowing developers to create SEO-friendly and user-friendly URLs.
- Templating Engine: Django’s templating engine supports inserting dynamic data into HTML through template tags and filters, enabling seamless integration of Python code and web pages.

- Rapid Development: With ready-to-use components and a comprehensive admin panel, developers can build and deploy applications faster, making Django ideal for projects with tight deadlines.
- Scalability and Versatility: Django is capable of handling high-traffic sites and supports building diverse web applications, from blogs and content management systems to complex e-commerce platforms and RESTful APIs.
- Community and Documentation: Django has a large, active community and excellent documentation, which facilitates learning, troubleshooting, and extending functionality.

How Django Works

Django handles web requests through its MVT components:

- Model: Defines data structure and database schema.
- View: Contains business logic to process requests and provide data to templates.
- Template: Renders HTML pages dynamically with data passed from views.

Q2:Advantages of Django (e.g., scalability, security).

Ans:

Scalability

- Django can handle growing traffic and complex applications efficiently. It supports scalable architectures used by large platforms like Instagram that manage millions of users.

Security

- Provides built-in protections against common vulnerabilities such as SQL injection, XSS, CSRF, and clickjacking. This robust security framework helps safeguard applications without extensive custom coding.

Rapid Development

- Its "batteries-included" philosophy offers built-in components like an admin panel, ORM, authentication, and form handling, allowing faster development and prototype building.

Maintainability and Clean Code

- Django encourages clean, reusable, and maintainable code via its MVT architecture, adherence to best practices, and DRY principle. This reduces long-term maintenance efforts and costs.

Rich Ecosystem and Community

- A large, active community provides extensive libraries, packages, and documentation, helping developers solve challenges faster and extend functionality effectively.

Versatility and Flexibility

- Suitable for various use cases: from simple websites and blogs to complex data-driven platforms, real-time apps, and RESTful APIs thanks to Django REST framework and other tools.

Cost-Effectiveness

- Reduces development time and cost due to reusable components and comprehensive out-of-the-box features, helping businesses launch products faster and maintain them easily.

SEO-Friendly Features

- Supports clean URL routing and sitemap generation to improve website search engine optimization, contributing to better discoverability online.

Q3:Django vs. Flask comparison: Which to choose and why.

Ans: Django and Flask are two popular Python web frameworks, each with distinct philosophies and use cases. Choosing between them depends on project requirements, developer expertise, and desired flexibility.

Django

- Type: Full-stack, “batteries-included” framework.
- Philosophy: Provides a comprehensive set of built-in tools such as an ORM, admin interface, authentication system, templating, and form handling to cover most web app needs out of the box.
- Best For: Large projects, complex applications, startups needing rapid development, or teams who want conventions and ready-made components.
- Scalability & Security: Comes with strong security features and is proven at handling traffic-heavy, scalable applications.
- Learning Curve: Steeper due to many concepts and components.
- Community and Ecosystem: Mature, large community with extensive third-party packages.
- Development Speed: Rapid prototyping with many built-in solutions.

Flask

- Type: Micro-framework, lightweight and minimalistic.
- Philosophy: Provides the basics to get started with web development. Developers add only the libraries they need.
- Best For: Small to medium applications, simple APIs, microservices, developers wanting fine-grained control, or learners just starting with web development.

- Scalability & Security: Requires more manual work to implement features; flexibility but less “ready out of box.”
- Learning Curve: Gentler due to simplicity and flexibility.
- Community and Ecosystem: Active and growing; many extensions available but less comprehensive than Django.
- Development Speed: Slower for large apps due to manual configuration but faster for small/simple apps.

Comparison Table

Feature	Django	Flask
Framework Type	Full-stack (batteries included)	Micro-framework (minimalist)
Built-in Features	ORM, admin panel, auth, templating, forms	Minimal basics, extensions required
Complexity	More complex, opinionated	Simple, flexible, unopinionated
Use Cases	Large, complex apps, rapid dev	Small apps, APIs, microservices
Scalability	Proven scalable, secure	Scalable with manual setup
Community	Large, mature	Active, growing
Learning Curve	Steeper	Gentler
Flexibility	Less (convention over configuration)	High (build what you need)
Development Speed	Fast for complex apps	Fast for small/simple apps

Which to Choose and Why?

- Choose Django if you want an all-in-one framework that handles most requirements with minimal setup, prioritize security and scalability, and build larger, complex applications quickly.

- Choose Flask if you prefer flexibility, want to build lightweight apps or APIs with custom components, or are learning web development and want to start simple.

5. Virtual Environment

Q1:Understanding the importance of a virtual environment in Python projects.

Ans: A virtual environment in Python projects is crucial for managing dependencies and isolating project-specific packages. It acts like a separate “toolbox” dedicated to each project, ensuring no conflicts arise between libraries or versions used across different projects.

Importance and Advantages

- Dependency Isolation: Each project maintains its own set of libraries, avoiding clashes like needing different versions of the same package simultaneously in different projects.
- Avoids Global Impact: Installing or upgrading packages in one project won’t affect system-wide Python or other projects, preventing accidental breakage or pollution of the global environment.
- Reproducibility: When sharing projects, virtual environments enable replicating the exact dependency setup by sharing files like `requirements.txt` that list installed packages and versions.
- Simplifies Collaboration: Team members can easily recreate the same environment ensuring consistent behavior across development machines.
- Clean and Organized: Keeps your development workspace tidy since dependencies are isolated within project folders, making maintenance easier.

- Supports Multiple Python Versions: You can create virtual environments with different Python interpreters if needed, isolating everything from the system Python.

Q2:Using `venv` or `virtualenv` to create isolated environments.

Ans: Using `venv` or `virtualenv` to create isolated Python environments lets you manage project-specific dependencies cleanly without interfering with the global Python installation.

venv (Standard Library Module)

- Availability: Comes built-in with Python 3.3+; no separate installation needed.
- Create Environment:

```
Text

    python3 -m venv envname
```

- This creates an `envname` directory containing a new Python interpreter and isolated package storage.
- Activate Environment:

On Windows:

```
Text

    envname\Scripts\activate
```

- **On Mac/Linux:**

```
Text

    source envname/bin/activate
```

- Deactivate (to exit):

```
Text
```

```
    deactivate
```

- Install Packages inside the activated environment using pip, independent from the system Python.

virtualenv (Third-Party Tool)

- Availability: Needs to be installed via pip:
- `text`

```
pip install virtualenv
```

- Create Environment:
- `text`

```
virtualenv envname
```

- Similar to venv, it creates an isolated environment.
- Activation/Deactivation are the same as with `venv`.
- Benefits over venv: Supports older Python versions, works with legacy environments, sometimes preferred by specific workflows.

Why Use Them?

- They isolate project dependencies so multiple projects can use different versions of the same library.
- Avoid conflicts and maintain clean development setups.
- Facilitate reproducible development environments across teams.

6. Project and App Creation

Q1:Steps to create a Django project and individual apps within the project.

Ans: Creating a Django project and individual apps within it involves a few straightforward command-line steps and basic setup. Here's a step-by-step guide:

1. Install Django (if not installed)

bash

```
pip install django
```

2. Create Django Project

Use the `django-admin` tool to start a new project:

Bash

```
      django-admin startproject projectname
```

- This creates a folder named `projectname` with core project files such as `settings.py`, `urls.py`, and `wsgi.py`.

3. Navigate into Project Directory

bash

```
cd projectname
```

4. Create an App Within the Project

Use the Django management command to create an app:

Bash

```bash
python manage.py startapp appname
```

- This creates an `appname` directory with files like `models.py`, `views.py`, `admin.py`, and others for app-specific development.

5. Register the App in Project Settings

- Open `projectname/settings.py`
- Add `'appname',` to the `INSTALLED_APPS` list to include the app in your project:

  python

  ```python
  INSTALLED_APPS = [
      # existing apps,
      'appname',
  ]
  ```

6. Run the Development Server to Verify Setup

Bash

```bash
python manage.py runserver
```

- Access `http://127.0.0.1:8000/` in your browser to see the Django welcome page.

Q2:Understanding the role of `manage.py`, `urls.py`, and [`views.py`](views.py).

Ans: In Django, the files `manage.py`, `urls.py`, and `views.py` have distinct roles in managing the project and handling web requests:

manage.py

- A command-line utility automatically created with every Django project.
- Serves as a wrapper around the Django administrative commands.
- Used to run development server, create apps, apply database migrations, run tests, and execute custom management commands.
- It sets the project's environment and enables easy management from the project directory.

urls.py

- Acts as the URL routing configuration for the project or an individual app.
- Maps URL patterns (web addresses) to specific views that process requests.
- Supports modular URL design by including other app URLconfs.
- Enables clean and human-readable URL schemes by linking URLs to Python code.

views.py

- Contains the functions or classes (views) that handle web requests.
- Views receive a request, process any business logic or data fetching, and return a response (often by rendering an HTML template).
- Can be function-based or class-based and determine what users see when visiting a URL.
- Responsible for connecting the URL routing to the actual content and logic of your site.

7. MVT Pattern Architecture

Q1:Django’s MVT (Model-View-Template) architecture and how it handle request-response cycles.

Ans: Django's MVT (Model-View-Template) architecture is a variation of the traditional MVC pattern tailored for web development. It cleanly separates the concerns of data, business logic, and user interface, facilitating efficient handling of web request-response cycles.

Components and Roles

<u>Model:</u>

- Represents the data layer.
- Defines the data structure and schema of the application via Python classes.
- Interacts with the database using Django’s ORM to retrieve, create, or modify data.

<u>View:</u>

- Acts as the business logic layer and controller.
- Handles user HTTP requests.
- Processes user input, interacts with Models to fetch or update data.
- Passes data to Templates for rendering responses.
- Can be function-based or class-based views.

<u>Template:</u>

- Manages the presentation layer.
- Comprises HTML files embedded with Django Template Language (DTL) syntax.
- Dynamically renders data supplied by Views to produce user-facing web pages.

Request-Response Cycle in MVT

1. Request Reception: When a user makes a request (visiting a URL), Django's URL routing system maps it to the appropriate View.
2. Processing in View: The View logic executes, fetching data from the Model as needed, performing any business logic.
3. Rendering Template: The View renders a Template, injecting dynamic data into the HTML structure.
4. Response Delivery: The fully rendered HTML page is sent back as an HTTP response to the user's browser.

This separates concerns, makes the codebase organized, and simplifies development, debugging, and scaling.

Analogy

- The Model is the recipe (data).
- The View is the chef (logic and preparation).
- The Template is the presentation and plating that the customer sees (UI).

Django automates controller responsibilities by integrating URL routing and view dispatching, streamlining development.

8. Django Admin Panel

Q1:Introduction to Django's built-in admin panel.

Ans: Django's built-in admin panel is a powerful, ready-to-use web interface that allows developers and site administrators to manage the application's data easily through CRUD (Create, Read, Update, Delete) operations.

Key Features of Django Admin Panel

- Automatic Interface Generation: Once your models are defined and registered with the admin site, Django automatically creates an administrative UI to manage database records without additional coding.
- User Management: Supports creating superusers with full access, staff users with limited permissions, and managing authentication and authorization within the admin dashboard.
- Model Registration: You register your application models in the `admin.py` file to make them accessible and manageable via the admin panel.
- Customizability: Allows extensive customization of displayed fields, search capabilities, filters, inline editing, and grouping through `ModelAdmin` classes for tailored admin experiences.
- Security: Access is secured via user authentication, ensuring only authorized personnel can manage sensitive application data.
- Productivity Booster: Provides quick data manipulation and site configuration, helping developers and admins save time while maintaining the application.

Typical Setup and Access

1. Create a Superuser:
   `Text`

   ```
   python manage.py createsuperuser
   ```

2. This creates an admin user who can log in to the admin site.
3. Run Server:

   Text

   ```
   python manage.py runserver
   ```

4. Access Admin Panel:

   Navigate to `http://127.0.0.1:8000/admin/` in a browser and log in with superuser credentials.

5. Register Models: Add in `admin.py` of your app:

python

```
from django.contrib import admin

from .models import MyModel

admin.site.register(MyModel)
```

The Django admin panel is a cornerstone tool for rapid development, data management, and administrative tasks, making it one of Django's standout features for building scalable web applications with minimal effort.

Q2:Customizing the Django admin interface to manage database records.

Ans:aCustomizing the Django admin interface allows you to tailor how database records are displayed and managed, improving usability and efficiency when handling data.

Key Customization Options

1. Register Models with Custom `ModelAdmin`

- In your app's `admin.py`, create a class inheriting from `admin.ModelAdmin` and specify customization options:
  python

```
from django.contrib import admin

from .models import MyModel


class MyModelAdmin(admin.ModelAdmin):

    list_display = ('field1', 'field2', 'field3')  #
Columns shown in list view

    search_fields = ('field1', 'field2')           #
Fields searchable via search bar

    list_filter = ('field3',)                       #
Filters shown to refine the list view

    ordering = ('field1',)                          #
Default ordering of records


admin.site.register(MyModel, MyModelAdmin)
```

2. Inline Editing

- Show related models inline on the parent model's edit page using `TabularInline` or `StackedInline`:
  python

```
from django.contrib import admin
from .models import ParentModel, ChildModel


class ChildModelInline(admin.TabularInline):
    model = ChildModel
    extra = 1  # Number of empty forms to display
class ParentModelAdmin(admin.ModelAdmin):
    inlines = [ChildModelInline]


admin.site.register(ParentModel, ParentModelAdmin)
```

3. Custom Form Behavior

- Override forms or widgets to add validations or custom fields using the `form` attribute in `ModelAdmin` or by customizing form classes.

4. Fieldsets for Organizing Fields

- Group fields into sections in the change form:

python

```
class MyModelAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {'fields': ('field1', 'field2')}),
        ('Advanced options', {'fields': ('field3',), 'classes':
('collapse',)}),
    )
```

5. Actions and Custom Buttons

- Add custom actions to perform batch operations on selected records.
- Define methods with `@admin.action` decorator to integrate custom commands.

9. URL Patterns and Template Integration

Q1:Setting up URL patterns in `urls.py` for routing requests to views.

Ans: Setting up URL patterns in Django's `urls.py` file is essential for routing incoming HTTP requests to the appropriate view functions or classes that handle the request and generate a response.

1. Import Required Modules

At the top of `urls.py`, import the necessary functions and your views:

```
Python

   from django.urls import path

   from . import views  # Import views from the same app
```

2. Define URL Patterns

Create a list named `urlpatterns` containing `path()` calls mapping URLs to views:

```
python

urlpatterns = [

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

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

```
    path('items/<int:item_id>/', views.item_detail,
name='item_detail'),  # URL with parameter
]
```

- The first argument to `path()` is the URL pattern (empty string `''` for root).
- The second argument is the view function or class-based view's `.as_view()`.
- The optional `name` helps with reverse URL lookups in templates and code.
- You can use path converters like `<int:>`, `<str:>`, `<slug:>`, to capture URL parameters.

3. Include App URLs in Project URLs

In your project's main `urls.py` (outside apps), include app-level URLconfs for modular routing:

```python
from django.urls import path, include

urlpatterns = [
    path('appname/', include('appname.urls')),  # Delegate
app URLs under /appname/
]
```

How Django Uses URL Patterns

When a request is made:

- Django checks each pattern in `urlpatterns` sequentially.
- Upon finding a match, it calls the corresponding view.
- Captured parameters from the URL are passed as arguments to the view.

This routing mechanism connects URL paths to the Python code that generates the response, making your web application navigable and organized.

Q2:Integrating templates with views to render dynamic HTML content.

Ans: Integrating templates with views in Django allows you to render dynamic HTML content by passing data from Python code (views) to HTML files (templates).

1. Create HTML Template

- Place your template files (e.g., `home.html`) inside the `templates` directory within your app or project.
- Use Django Template Language (DTL) to insert dynamic content using `{{ variable }}` syntax.

  Example `home.html`:

xml

```
<!DOCTYPE html>

<html>

<head>

    <title>My Site</title>

</head>

<body>

    <h1>Welcome, {{ user_name }}!</h1>

    <p>Your favorite color is {{ favorite_color }}.</p>

</body>

</html>
```

2. Define View to Render Template

- In `views.py`, import `render` and create a function that takes a request and returns a rendered template with context data:

python

```python
from django.shortcuts import render

def home_view(request):
    context = {
        'user_name': 'Alice',
        'favorite_color': 'blue'
    }
    return render(request, 'home.html', context)
```

- The `context` dictionary holds variables that the template will use.

3. Map URL to the View

- In `urls.py`, link a URL pattern to `home_view`:

  python

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


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

How It Works

- User visits the URL mapped to `home_view`.
- `home_view` processes the request, prepares data in `context`.

- `render()` merges template (`home.html`) with context variables to create HTML.
- Server sends generated HTML response to user's browser.

10. Form Validation using JavaScript

Q1:Using JavaScript for front-end form validation.

Ans: Using JavaScript for front-end form validation improves user experience by providing immediate feedback on form inputs, reducing errors before data is submitted to the server.

How Front-End JavaScript Validation Works

- JavaScript scripts run in the user’s browser.
- Hooks into form elements events (e.g., submit, input, change).
- Checks input values against validation rules (e.g., required fields, format, length).
- Displays errors instantly near fields or as alerts.
- Prevents form submission if validation fails.

Basic Example of JavaScript Form Validation

xml

```
<form id="myForm" onsubmit="return validateForm()">

  <label for="email">Email:</label><br>

  <input type="text" id="email" name="email"><br>

  <small id="errorMsg" style="color:red;"></small><br>
  <input type="submit" value="Submit">
</form>

<script>
function validateForm() {
  const emailInput =
document.getElementById('email').value.trim();
  const errorMsg = document.getElementById('errorMsg');
  errorMsg.textContent = '';
  // Basic email regex pattern
  const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

  if (emailInput === '') {
    errorMsg.textContent = 'Email is required.';
    return false;  // Prevent form submit
  }
  if (!emailPattern.test(emailInput)) {
    errorMsg.textContent = 'Please enter a valid email
address.';
    return false;
  }
  return true;  // Allow form submission
}
</script>
```

Benefits of Front-End Validation

- Improves UX: Users get instant feedback on input errors.
- Reduces Server Load: Minimizes invalid data requests.
- Customizable: Tailor validations and error messages to your needs.

Important Note

- Always complement front-end validation with server-side validation for security and data integrity.

11. Django Database Connectivity (MySQL or SQLite)

Q1:Connecting Django to a database (SQLite or MySQL).

Ans: Connecting Django to a database like SQLite or MySQL involves configuring the database settings in your project's `settings.py` file.

1. Connecting to SQLite (Default)

SQLite is the default database Django uses for new projects. No setup is usually needed beyond default settings.

In `settings.py`:

`python`

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

- `db.sqlite3` file will be created in your project directory.
- SQLite is file-based, easy for development and small projects.

---

2. Connecting to MySQL

For MySQL, additional steps are required:

Step 1: Install MySQL client library

Bash

```
pip install mysqlclient
```

or for Windows, you might use:

Bash

```
pip install pymysql
```

and include:

Python

```
import pymysql

pymysql.install_as_MySQLdb()
```

Step 2: Configure `settings.py`

python

```
DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.mysql',

        'NAME': 'your_database_name',

        'USER': 'your_mysql_user',

        'PASSWORD': 'your_mysql_password',

        'HOST': 'localhost',  # Or your DB server address
```

```
            'PORT': '3306',       # Default MySQL port
        }
    }
```

Step 3: Create MySQL database

- Manually create the database in MySQL:

Sql

```
CREATE DATABASE your_database_name CHARACTER SET UTF8;
```

Step 4: Migrate your Django models

Bash

```
python manage.py migrate
```

This applies Django's built-in models and your app models to the database.

Q2:Using the Django ORM for database queries.

Ans:aThe Django ORM (Object-Relational Mapper) allows you to interact with your database using Python code instead of raw SQL, translating Python classes and methods into database queries.

Basics of Using Django ORM

1. Define Models

Models represent database tables. For example:

Python

```
        from django.db import models


        class Book(models.Model):
            title = models.CharField(max_length=100)
            author = models.CharField(max_length=50)
            published_year = models.IntegerField()
```

2. Create and Run Migrations

- Create migration files that sync models with the database:

bash

```
     python manage.py makemigrations
     python manage.py migrate
```

3. Basic Queries Using ORM

- **Create Record:**

python

```
     book = Book.objects.create(title='1984', author='George
       Orwell', published_year=1949)
```

- **Retrieve Records:**

python

```
all_books = Book.objects.all()  # Get all records
```

```
some_books = Book.objects.filter(author='George Orwell')  # Filter records
single_book = Book.objects.get(id=1)  # Get a single record by id
```

- **Update Record:**

Python

```
    book = Book.objects.get(id=1)
    book.title = 'Animal Farm'
    book.save()
```

- **Delete Record:**

Python

```
    book = Book.objects.get(id=1)
    book.delete()
```

4. QuerySet Methods

- `filter()`, `exclude()`, `order_by()`, `values()`, `distinct()`, etc.
- Allow chaining for complex queries:

Python

```
books = Book.objects.filter(published_year__gte=1950).order_by('title')
```

- Double underscores __ denote field lookups (e.g., `__gte` means greater than or equal to).

12. ORM and QuerySets

Q1:Understanding Django’s ORM and how QuerySets are used to interact with the database.

Ans: Django's ORM (Object-Relational Mapper) is a powerful abstraction layer that allows developers to interact with databases using Python code instead of writing raw SQL queries. It connects Python classes (called models) with database tables and Python objects with database records, enabling operations like creating, querying, updating, and deleting records in a database in a Pythonic way.

How Django ORM Works

- The ORM maps database tables to Python classes (models) and rows to instances of these classes.
- Developers define models in Python, and Django handles generating and executing SQL queries behind the scenes.
- This allows you to work with data as Python objects rather than SQL statements, simplifying database manipulation and improving code maintainability.

QuerySets

- A QuerySet is essentially a collection (list) of objects of a given model representing database records.
- QuerySets allow retrieving data from the database in various ways: all objects, filtered objects meeting certain conditions, excluded data, or even single objects.
- Common QuerySet methods include `.all()`, `.filter()`, `.exclude()`, `.get()`, `.update()`, and `.delete()`.
- QuerySets are lazy, meaning the actual database query is not performed until the data is needed.
- They also support chaining multiple filters and ordering the data, as well as relationship traversal between models.

Practical Example

If you have a `Book` model, using the ORM you can retrieve all books with:

```
Python

    books = Book.objects.all()
```

You can filter books by author name like:

```
python

books_by_author = Book.objects.filter(author__name='John Doe')
```

You can also update or delete records easily via QuerySets.

Benefits of Django ORM

- Simplifies database interaction by abstracting SQL queries with clean Pythonic code.
- Supports multiple database backends like SQLite, MySQL, PostgreSQL, and Oracle, making it easy to switch databases.
- Improves code readability, maintainability, and reduces SQL injection risks.
- Provides built-in support for aggregations, annotations, and relationship management in a consistent way.

In summary, Django's ORM along with QuerySets enables developers to interact with databases using Python objects and methods, making data handling intuitive and efficient without writing raw SQL queries.

13.Django Forms and Authentication

Q1:Using Django's built-in form handling.

Ans:

Creating Forms

- You define a form by creating a class that inherits from `django.forms.Form` (for manual forms) or `django.forms.ModelForm` (auto-generated from a model).
- Form fields are declared as class attributes, specifying the type of input (e.g., `CharField`, `EmailField`) and validation rules.
- ModelForms automatically generate fields based on the model and can save data directly to the database with minimal code.

Using Forms in Views

- In a view, instantiate the form class, binding it with POST data on submission.
- Check `form.is_valid()` to trigger validation.
- If valid, access cleaned data via `form.cleaned_data` and process or save it.
- If not valid, render the form again with error messages.

Example view handling a form:

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

def my_view(request):
    if request.method == 'POST':
        form = MyForm(request.POST)
        if form.is_valid():
            # Process form.cleaned_data
            return redirect('success_url')
    else:
        form = MyForm()  # empty form for GET request
    return render(request, 'template.html', {'form': form})
```

Rendering Forms in Templates

- In your template, use `{{ form }}` to render the whole form or render fields individually.
- Include `{% csrf_token %}` in the form for security.

➢ **Advanced Features**

- Django supports formsets (multiple forms on a single page), file uploads, and class-based views like `FormView` to handle form workflows with less boilerplate.
- Forms handle error messaging, validation, and reuse seamlessly.

Q2:Implementing Django's authentication system (sign up, login, logout, password management).

Ans: Django provides a full-featured authentication system, supporting user sign-up, login, logout, and password management with both built-in forms and views. Here's how these core features are typically implemented:

Sign Up (Registration)

- Django does not provide a built-in signup URL or view—you must implement this yourself.
- Use Django's `UserCreationForm` for secure user registration.
- Example sign-up view:
- `python`

```
from django.contrib.auth import login
from django.contrib.auth.forms import UserCreationForm
from django.shortcuts import render, redirect

def sign_up(request):
    if request.method == "POST":
        form = UserCreationForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)  # Optional: Logs in after signup
            return redirect("dashboard")  # Redirect to a custom dashboard
or home
    else:
        form = UserCreationForm()
    return render(request, "registration/sign_up.html", {"form": form})
```

- A corresponding template like `registration/sign_up.html` should display the form.

Login and Logout

- Django provides ready-to-use views for login (`django.contrib.auth.views.LoginView`) and logout (`LogoutView`).
- You typically include their URLs in your `urls.py` as:
- `python`

```
from django.urls import path
from django.contrib.auth import views as auth_views

urlpatterns = [
    path("login/", auth_views.LoginView.as_view(), name="login"),
    path("logout/", auth_views.LogoutView.as_view(), name="logout"),
]
```

- Create templates: `registration/login.html` for login.

Password Management

- Django covers all aspects:
- Change password (when logged in): `password_change` and related views.
- Reset password (when forgotten): `password_reset`, `password_reset_confirm`, and related views.
- Just add Django's provided paths to your `urls.py`:

  `python`

```
urlpatterns += [
    path("password_change/", auth_views.PasswordChangeView.as_view(),
name="password_change"),
    path("password_change/done/",
auth_views.PasswordChangeDoneView.as_view(),
name="password_change_done"),
    path("password_reset/", auth_views.PasswordResetView.as_view(),
name="password_reset"),
    path("password_reset/done/",
auth_views.PasswordResetDoneView.as_view(), name="password_reset_done"),
    path("reset/<uidb64>/<token>/",
auth_views.PasswordResetConfirmView.as_view(),
name="password_reset_confirm"),
    path("reset/done/", auth_views.PasswordResetCompleteView.as_view(),
name="password_reset_complete"),
]
```

-
- You must create templates for each view, such as `registration/password_reset_form.html`.

Notes and Best Practices

- All these views require templates in the `registration/` directory.
- Authentication relies on Django's session system, and it is strongly recommended to use Django's built-in forms for security.
- You can customize workflow and templates, and extend the user model if needed for advanced use cases.

14. CRUD Operations using AJAX

Q1:Using AJAX for making asynchronous requests to the server without reloading the page.

Ans: AJAX (Asynchronous JavaScript and XML) allows web pages to send requests to the server and update parts of a page asynchronously without needing a full page reload. In Django, AJAX is commonly used to improve user experience by providing seamless interactions.

How to Use AJAX with Django

1. Setup Frontend AJAX Request: You can use JavaScript's Fetch API or jQuery to send asynchronous requests to Django views.

   Example with jQuery for a GET request:

   javascript

```
$.ajax({
    type: "GET",
    url: "/your-url/",
    data: { key: "value" },
    success: function(response) {
        // Update page content using response data
        $('#result').html(response);
    }
});
```

2. For POST requests, include CSRF token for security:
   javascript

```
$.ajax({
    type: "POST",
    url: "/your-url/",
    data: { key: "value", csrfmiddlewaretoken: '{{ csrf_token }}' },
    success: function(response) {
        // Handle response
    }
});
```

3. Create Django View to Handle AJAX
   The view should check if the request is AJAX (by checking `X-Requested-With` header) and method type, then return an appropriate JsonResponse or HttpResponse.

   Example Django view for AJAX request handling:
   python

```
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt


@csrf_exempt  # Only if you handle CSRF separately or use AJAX with tokens
def ajax_view(request):
    if request.headers.get('X-Requested-With') == 'XMLHttpRequest':
        if request.method == 'GET':
            # Process GET request
            data = {"message": "This is an AJAX GET response."}
            return JsonResponse(data)
        elif request.method == 'POST':
            # Process POST request
            value = request.POST.get('key', None)
            data = {"message": f"Received {value}"}
            return JsonResponse(data)
    return JsonResponse({"error": "Invalid request"}, status=400)
```

4. Include URLs:
   Add URL patterns for AJAX view in `urls.py`.
   python

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


urlpatterns = [
    path('ajax/', views.ajax_view, name='ajax_view'),
```

]

5. Update HTML Template
   Include jQuery or plain JS, and write JavaScript code to trigger AJAX calls on events like button clicks or form submissions. Update parts of the page with the response from the server dynamically.

   Important Notes

- Always include CSRF tokens in POST AJAX requests to protect against CSRF attacks.
- Use `JsonResponse` to send structured JSON data back to the frontend.
- AJAX requests improve user experience by preventing full-page reloads and delivering faster interactions.
- You can handle various HTTP methods (GET, POST, PUT, DELETE) via AJAX with Django views.

15. Customizing the Django Admin Panel

Q1:Techniques for customizing the Django admin panel.

Ans:

Customize ModelAdmin

- Use a `ModelAdmin` class to control how a model appears and behaves in the admin.
- Key attributes:
  - `list_display`: Fields to show in the list view.
  - `search_fields`: Add search boxes for specified fields.
  - `list_filter`: Sidebar filters for specific fields.
  - `ordering`: Default ordering of records.
  - `fields` or `fieldsets`: Define layout of the form for adding/editing.
- Example:
  `python`

```python
from django.contrib import admin
from .models import Book

class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'published_date')
    search_fields = ('title', 'author__name')
    list_filter = ('published_date',)
    ordering = ('-published_date',)

admin.site.register(Book, BookAdmin)
```

Customize Form Behavior

- Override or extend default forms used in admin by specifying `form` in `ModelAdmin` to a custom form class.
- Use this to add validation, widgets, or initial data.

Inline Models

- Display related models inline for easier editing.
- Use `StackedInline` or `TabularInline` classes.
- Example:
  `python`

```
class ChapterInline(admin.TabularInline):
    model = Chapter


class BookAdmin(admin.ModelAdmin):
    inlines = [ChapterInline]
```

Admin Site Customization

- Change site header, title, and index page by overriding admin site attributes:
- `python`

```
admin.site.site_header = "My Project Admin"
admin.site.site_title = "My Project Admin Portal"
admin.site.index_title = "Welcome to the Admin Area"
```

Add Custom Actions

- Define custom actions available in the list view to perform batch operations on selected objects.
- Example:
  `python`

```
def mark_published(modeladmin, request, queryset):
    queryset.update(status='published')

mark_published.short_description = "Mark selected books as published"

class BookAdmin(admin.ModelAdmin):
    actions = [mark_published]
```

Use JavaScript and CSS

- Add custom JavaScript or CSS files to admin for more elaborate UI customizations by overriding `Media` class inside `ModelAdmin`.

Admin Templates

- Override admin templates for extensive UI changes by placing custom templates in your app's `templates/admin/` directory, matching default admin template paths.

16.Payment Integration Using Paytm

Q1:Introduction to integrating payment gateways (like Paytm) in Django projects.
Ans:
Prerequisites

- Create a Paytm merchant account and obtain your merchant ID, secret key, and other required credentials.
- Install required libraries (sometimes Paytm provides an SDK or you can use custom Python code for checksum generation and verification).

Basic Steps for Integration

1. Configure Paytm Credentials in Django Settings
   Add your Paytm merchant details to `settings.py`:
   `python`

```
PAYTM_MERCHANT_ID = '<your_merchant_id>'
PAYTM_SECRET_KEY = '<your_secret_key>'
PAYTM_WEBSITE = 'WEBSTAGING'  # or 'DEFAULT' for production
PAYTM_CHANNEL_ID = 'WEB'
PAYTM_INDUSTRY_TYPE_ID = 'Retail'
```

2. Create a Payment Initiation View
   This view renders a form (or payment page) where the user fills transaction details. Upon submission, generate a checksum using the transaction data and secret key to send to Paytm securely.
3. Generate Checksum and Redirect to Paytm
   Use helper functions (usually provided by Paytm SDK or written manually) to generate a checksum hash. Redirect the user to Paytm's payment URL with required parameters and checksum.

4. Create a Callback/Response Handler View
   After payment, Paytm posts transaction status back to your callback URL. You must verify the checksum to confirm the response is genuine, then update the transaction status in your database accordingly.
5. Update URLs
   Map URLs for the payment initiation and callback views.
6. Templates for Payment and Callback
   Create user-facing pages for initiating payment and displaying success/failure messages after callback.

Example Flow Outline (Simplified)

- User visits `/pay/` to initiate payment.
- Django view collects transaction details, generates checksum, and posts to Paytm payment gateway.
- User completes payment on Paytm site.
- Paytm sends payment response to `/callback/` on your site.
- Your callback view verifies and updates payment status, then shows result to user.

Additional Options

- You can use third-party Django packages or SDKs to simplify checksum handling and API calls.
- Handle errors, rollback, and logging carefully to improve reliability.
- For production, use Paytm's live endpoints and credentials, and implement HTTPS for security.

This integration approach ensures secure, reliable payment handling within your Django project using Paytm gateway. A detailed tutorial with example code is available and often includes setting up views, URLs, models, forms, and checksum utilities to complete the integration.

17. GitHub Project Deployment

Q1: Steps to push a Django project to GitHub.

Ans:To push a Django project to GitHub, follow these steps:

1. Initialize Git Repository

- Open a terminal in your Django project root folder.
- Run:

  Bash

  ```
  git init
  ```

2. Create a .gitignore File

- Create a `.gitignore` file to exclude files like `__pycache__`, `.env`, `db.sqlite3`, and other sensitive or unnecessary files.
- Example contents:

  text

```
__pycache__/
*.pyc
db.sqlite3
.env
*.log
/venv/
*.sqlite3
```

3. Stage Files for Commit

- Stage your project files with:
- Bash

  ```
  git add .
  ```

4. Commit Changes

- Commit files with a meaningful message:

  Bash

  ```
  git commit -m "Initial commit of Django project"
  ```

5. Create a GitHub Repository

- Go to GitHub.

- Create a new repository (public or private) without initializing with a README, .gitignore, or license (since those are already in your local repo).

6. Add Remote Repository

- Add GitHub repo as a remote:
- `Bash`

```
git remote add origin https://github.com/username/reponame.git
```

7. Push to GitHub

- Push your local commits to GitHub:
- `bash`

```
git push -u origin master
```

- or if you use the main branch:
- `bash`

```
git push -u origin main
```

Important Tips

- Keep secrets like API keys or database passwords out of the repo; use environment variables or `.env` files ignored by git.
- Regularly commit changes and push to keep your repo updated.
- You can add a `README.md` for project documentation.

18. Live Project Deployment (PythonAnywhere)

Q1:Introduction to deploying Django projects to live servers like PythonAnywhere.

Ans: Deploying a Django project to a live server like PythonAnywhere enables making your web application publicly accessible. Here's an introduction to the deployment process:

Setting Up on PythonAnywhere

1. Create a PythonAnywhere Account

   Sign up at

2. pythonanywhere.com
3. and log in.
4. Upload Your Django Project

   You can upload your project files via the PythonAnywhere file manager or use Git to clone your project repository directly on the server.
5. Create a Virtual Environment and Install Dependencies

- On the PythonAnywhere bash console, create a virtual environment matching your Django project's Python version:

```
bash

    python3 -m venv myenv
```

- Activate it:
- bash

```
source myenv/bin/activate
```

- Install requirements using `pip` and your `requirements.txt`:
- bash

```
pip install -r requirements.txt
```

6. Configure Your Django Settings for Production
   - Set `DEBUG = False`.
   - Adjust `ALLOWED_HOSTS` to include your PythonAnywhere domain.
   - Configure static files (`STATIC_ROOT`) for collection.
7. Collect Static Files

   Run:

```
bash

    python manage.py collectstatic
```

8. This collects static files for serving.
9. Set Up the Web App on PythonAnywhere

- Go to the "Web" tab on PythonAnywhere.
- Click "Add a new web app" and select manual configuration with the correct Python version.

- Set the working directory to your project folder.
- Configure the WSGI file to point to your Django project's `wsgi.py`.

10. Configure Static Files
    Under the "Static files" section, add URL and directory mappings for serving static and media files.
11. Reload the Web Application
    After configuration, reload the web app from the PythonAnywhere dashboard to start serving your Django project.

Additional Notes

- Make sure your database is correctly set up on the server (e.g., SQLite works out-of-the-box, for others like MySQL, configure access).
- Secure sensitive credentials using environment variables or the PythonAnywhere secrets manager.
- For continuous deployment, consider pushing updates via Git and pulling them on PythonAnywhere.

19. Social Authentication

Q1: Setting up social login options (Google, Facebook, GitHub) in Django using OAuth2.

Ans:

1. Install Required Package

Install `django-allauth` which supports social authentication providers:

Bash

```
pip install django-allauth
```

2. Update Django Settings (`settings.py`)

- Add apps:

python

```
INSTALLED_APPS = [
    # other apps
    'django.contrib.sites',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'allauth.socialaccount.providers.google',
    'allauth.socialaccount.providers.facebook',
    'allauth.socialaccount.providers.github',
]

SITE_ID = 1
```

- Configure authentication backends:
  python

```
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',  # default
    'allauth.account.auth_backends.AuthenticationBackend',  # allauth
)
```

- Add social account settings (example for Google):
  python

```
SOCIALACCOUNT_PROVIDERS = {
    'google': {
        'SCOPE': ['profile', 'email'],
        'AUTH_PARAMS': {'access_type': 'online'},
        'OAUTH_PKCE_ENABLED': True,
    },
    # Add similar blocks for Facebook and GitHub if needed
}
```

3. URL Configuration

Add allauth URLs in your project's `urls.py`:

python

```
from django.urls import path, include

urlpatterns = [
    # ... other urls
    path('accounts/', include('allauth.urls')),
]
```

4. Create OAuth Apps on Provider Platforms

- Google: Use
- Google Cloud Console , create credentials for OAuth2 client, note the Client ID and Client Secret.
- Facebook: Use
- Facebook for Developers , create an app, configure Facebook Login.
- GitHub: Use
- GitHub Developer Settings , register new OAuth app.
- Set callback/redirect URLs that point to your Django app routes, typically something like:

```text
http://yourdomain.com/accounts/google/login/callback/
```

(Replace `google` with `facebook` or `github` accordingly.)

5. Add Social Application in Django Admin

- Go to Django admin → Social Accounts → Social Applications.
- Add each provider app with Client ID, Secret Key, and associate with your `Site`.

6. Use in Templates

Add social login buttons in your login or signup templates by including the provider URLs, like:

```xml
<a href="{% provider_login_url 'google' %}">Login with Google</a>
<a href="{% provider_login_url 'facebook' %}">Login with Facebook</a>
<a href="{% provider_login_url 'github' %}">Login with GitHub</a>
```

7. Test Authentication Flow

Visit `/accounts/login/`, you should see options to log in using these social providers. Clicking one redirects externally to authenticate and then back to your app.

20. Google Maps API

Q1:Integrating Google Maps API into Django projects.

Ans: Integrating Google Maps API into Django projects allows you to display interactive maps, markers, geolocation data, or use geocoding services. Here's an introduction to how you can set this up:

1. Get Google Maps API Key

- Go to
- Google Cloud Console.
- Create a new project and enable required APIs (e.g., Maps JavaScript API, Geocoding API).
- Generate an API key and restrict it to your domains.

2. Add API Key in Django Settings

- Store your API key securely in `settings.py`:

python

```
GOOGLE_MAPS_API_KEY = 'your_api_key_here'
```

3. Create a Django View to Pass API Key to Template

python

```
from django.conf import settings
from django.shortcuts import render

def map_view(request):
    context = {'google_maps_api_key': settings.GOOGLE_MAPS_API_KEY}
    return render(request, 'map.html', context)
```

4. Create Template with Google Maps JS Library and Map Display

In `map.html`, include Google Maps JS script with your API key and create a div to render the map:

xml

```
<!DOCTYPE html>
<html>
<head>
  <title>Google Maps Integration</title>
  <script src="https://maps.googleapis.com/maps/api/js?key={{
google_maps_api_key }}&callback=initMap" async defer></script>
```

```
  <script>
    function initMap() {
      var location = {lat: 28.6139, lng: 77.2090};  // Example: New Delhi
coordinates
      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>
  <h1>My Google Map</h1>
  <div id="map" style="height: 400px; width: 100%;"></div>
</body>
</html>
```

5. Configure URL Routing

Add a URL pattern to serve the map view in your `urls.py`:

python

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

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

6. Optional: Use Third-Party Django Packages

- Packages like `django-google-maps` can simplify model field integration for addresses and geolocation with admin widgets.