2030/2040

PYTHON DB & FRAMEWORK

Presented to

Prashant Shukla Sir

TOPS Technologies

HTML in Python

Que.1
Embedding HTML within Python using web frameworks like Django or Flask is a common way to build dynamic web applications. Both Django and Flask allow you to write Python code to generate HTML, manage data, and handle requests. Here's an introduction to both frameworks and how they facilitate embedding HTML.

1. Flask (Micro-framework)
Flask is a micro-framework for Python that is lightweight and easy to get started with. It provides the essential tools for web development without enforcing strict patterns, making it ideal for smaller projects or when you need more flexibility.

2. Django (Full-fledged framework)
Django is a more feature-rich web framework designed for larger applications. It comes with built-in tools for handling forms, authentication, admin interfaces, and more.

Que.2
Generating dynamic HTML content using Django templates allows you to create web pages that change based on user input or data from a database. Django templates allow you to insert dynamic content using variables, loops, conditionals, and template tags, making it easy to generate personalized or data-driven content.

1. Basic Setup
First, ensure that you have Django installed and a basic project set up. If you haven't set up a Django project yet, follow these steps:

1. Basic Setup

Step 1. Install Djnago:

```
pip install django
```

Step 2. Creating A Django Project:

```
django-admin startproject myproject
cd myproject
```

Step 3. Creating a Django App:

```
python manage.py startapp myapp
```

Step 4: Add the app to INSTALLED_APPS in settings.py: In myproject/settings.py, add 'myapp' to the INSTALLED_APPS list:

```
INSTALLED_APPS = [
    ...
    'myapp',
    ...
]
```

2. Creating Views and Templates

Step 1. Creating a View to Pass Dynamic Content:
fIn your views.py file inside the myapp directory, create a view that passes dynamic content to the template.

Step 2. Creating the Template:
Create a folder named templates inside your app folder (myapp) and add an HTML file (e.g., dynamic_template.html).

Step 3. Configuring URL Patterns:
In order to view this dynamic page, you need to map the view to a URL.

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

urlpatterns = [
    path('dynamic/', views.dynamic_page,
name='dynamic_page'),
]
```

In your project’s urls.py (myproject/urls.py), include the myapp URLs:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),
]
```

Step 4. Running the Server:

```
python manage.py runserver
```

2. CSS in Python

Que.1
Step 1. Create a Static Folder:
First, make sure you have a static directory in your project. This directory is where you will store all your static files, including CSS, JavaScript, and images.

Step 2. Configure Static Files in settings.py:
Django uses the STATIC_URL and STATICFILES_DIRS settings to manage static files. Add or check the following settings in your settings.py:

```
STATIC_URL = '/static/'

# This line is optional, but useful if you have static files outside of each app's static folder.
STATICFILES_DIRS = [
    BASE_DIR / "static",  # This tells Django to also look for static files in a global 'static' folder at the project level.
]
```

Step 3. Load Static Files in Templates:
In your Django template, use the {% load static %} tag to access the static files, then link to the CSS file.

Step 4. Use {% block %} for Base Template (Optional):
If you're using a base template to manage common structure (e.g., header, footer), you can use blocks to load different CSS files for each page.

Que.2

Step 1. Setting up Static Files in Development:
Django uses the STATIC_URL setting to define where static files are accessible on the web. It also uses the STATICFILES_DIRS setting to specify additional directories where static files can be located.

```
# The URL where static files will be served from
(in development or production)
STATIC_URL = '/static/'

# Directory where static files will be stored
during development
STATICFILES_DIRS = [
    BASE_DIR / "static",  # If you have a global
static folder at the project level
]

# Directory for collecting static files for
production (you can define this later for
production)
STATIC_ROOT = BASE_DIR / "staticfiles"  # Used
only for production
```

Then
Make sure your static files are placed in the correct directory. Typically, this is inside each app's static folder.

Load Static Files in Templates

In your templates, use Django's {% load static %} tag to include static files.

```
{% load static %}

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
content="width=device-width, initial-scale=1.0">
    <title>My Website</title>
    <link rel="stylesheet" href="{% static
'myapp/css/style.css' %}">
</head>
<body>
    <h1>Welcome to My Website</h1>
    <script src="{% static 'myapp/js/script.js' %}">
</script>
</body>
</html>
```

The {% static 'myapp/css/style.css' %} will be replaced with the correct URL pointing to the static file.
Step 4: Serve Static Files in Development
Django automatically serves static files during development when DEBUG=True. So, if you're in development mode (i.e., DEBUG=True in settings.py), Django will handle serving static files for you.

Last Step :

```
python manage.py runserver
```

3. JavaScript with Python

Que.1
Step 1. Include JavaScript in Django Template:
You can include JavaScript either directly in the template or via an external file.
Option 1: Inline JavaScript in the Template:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>My Django Page</title>
</head>
<body>

   <h1>Welcome to My Django Page</h1>

   <button id="myButton">Click me!</button>

   <script>

document.getElementById('myButton').onclick =
function() {
           alert('Button clicked!');
       };
   </script>
</body>
</html>
```

Option 2: External JavaScript File
You can create a .js file and include it in your template.

Create a JavaScript file (e.g., static/js/main.js):

```
// static/js/main.js
document.getElementById('myButton').onclick =
function() {
    alert('Button clicked!');
};
```

Link the JavaScript file in your template:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Django Page</title>
    <script src="{% static 'js/main.js' %}"></script>
</head>
<body>

    <h1>Welcome to My Django Page</h1>

    <button id="myButton">Click me!</button>

</body>
</html>
```

2. Enable Static Files in Django
To use external static files (like JavaScript), you need to make sure static files are properly configured in your Django settings.

Configure Static Files in settings.py:
```
STATIC_URL = '/static/'

# For development only
STATICFILES_DIRS = [
    BASE_DIR / "static",
]
```

Use {% load static %} in the template to load the static file:
```
{% load static %}
```

3. Use Django Context Variables in JavaScript
You can pass Django context variables into your JavaScript by using template tags.
For example, if you want to pass a Django context variable into your JavaScript:

```
# views.py
from django.shortcuts import render

def my_view(request):
    context = {
        'message': 'Hello from Django!',
    }
    return render(request, 'my_template.html',
context)
```

In the template:

```
<script>
    const messageFromDjango = "{{ message }}";
    alert(messageFromDjango);  // Alerts: "Hello
from Django!"
</script>
```

Que.2
You store your .js file inside Django’s static folder and link it using {% static %}.

(a) Set up your static folder (if not already)

- Make sure you have this in settings.py:

```
STATIC_URL = '/static/'

# (optional) during development
STATICFILES_DIRS = [
    BASE_DIR / "static",
]
```

- Create a folder structure like:

```
your_project/
    static/
        js/
            script.js
```

Example script.js (inside static/js/script.js):

```
// static/js/script.js
console.log("Script loaded!");
document.addEventListener('DOMContentLoade
d', function() {
    alert('Hello from script.js!');
});
```

(b) Load and Link it in the Template

In your Django template (.html file):

```
{% load static %}

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Page</title>
</head>
<body>

    <h1>Welcome!</h1>

    <button id="clickMe">Click Me!</button>

    <!-- Link internal JS file -->
    <script src="{% static 'js/script.js' %}"></script>

</body>
</html>
```

2. Link External JavaScript Files (CDNs or Hosted Scripts)

For third-party libraries (like jQuery, Bootstrap JS, etc.), directly use their URLs.
Example: using jQuery from a CDN:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My Page</title>

  <!-- External JavaScript from a CDN -->
  <script src="https://code.jquery.com/jquery-
3.6.0.min.js"></script>

</head>
<body>

  <h1>Hello World</h1>

  <script>
    $(document).ready(function(){
      alert('Page loaded with jQuery!');
    });
  </script>

</body>
</html>
```

4. Django Introduction

Que.1
What is Django?

- Django is a high-level, open-source Python web framework.
- It encourages rapid development and clean, pragmatic design.
- Built by experienced developers, it handles much of the hassle of web development, so you can focus on writing your app.

Key Features:

- MTV Architecture: Django follows the Model-Template-View pattern (similar to MVC):
  - Model: Handles the data (database layer).
  - Template: Deals with the presentation (HTML layer).
  - View: Manages the logic and user interaction.
- ORM (Object-Relational Mapping): Easy database management without writing SQL.
- Built-in Admin Panel: Automatically generates a web-based admin interface from your models.

- Security: Protects against common threats like SQL injection, CSRF, XSS, and clickjacking.
- Scalability: Supports high-traffic websites (e.g., Instagram, Pinterest initially used Django).
- Versatility: Can be used for any type of website (content management, e-commerce, scientific computing platforms, etc.).
- DRY Principle (Don't Repeat Yourself): Encourages code reuse and reducing redundancy.

Advantages:

- Fast development cycle.
- A large collection of built-in libraries and packages.
- Excellent documentation.
- A strong and active community.

Typical Use Cases:

- Content management systems (CMS)
- Social networks
- E-commerce sites
- APIs (with Django REST Framework)

Popular Sites Using Django:

- Instagram
- Pinterest
- Mozilla
- Disqus

Que.2

Advantages of Django

- Scalability
    - Django is designed to handle high traffic and large volumes of data.
    - It's used by companies like Instagram and Disqus, proving its ability to scale effectively.
- Security
    - Django has built-in protections against common security threats:
    - SQL injection
    - Cross-site scripting (XSS)
    - Cross-site request forgery (CSRF)
    - Clickjacking
    - It manages user authentication and password storage securely.
- Rapid Development
    - Django allows you to build web applications quickly with fewer lines of code.
    - The built-in features (like admin panel, ORM, form handling) save a lot of development time.
- Versatile and Flexible
    - You can build all kinds of sites: from simple blogs to large e-commerce platforms, APIs, or machine learning applications.

- Built-in Admin Interface
    - Django automatically creates a powerful admin dashboard based on your models, saving time and effort for backend management.
- Excellent Documentation
    - Django has some of the best documentation among open-source projects, making it easier to learn and solve problems.
- Object-Relational Mapping (ORM)
    - The ORM simplifies database operations without writing SQL manually, increasing development speed and minimizing bugs.
- Component-Based and Modular
    - Django is organized into smaller reusable components (apps), making your code modular and easy to maintain.
- Strong Community Support
    - A large, active community means better support, more third-party packages, frequent updates, and long-term stability.
- Portable and Cross-Platform
    - Django runs on Windows, MacOS, Linux, and can work with multiple database systems like PostgreSQL, MySQL, SQLite, etc.

Que.3

Feature	Django	Flask
Type	Full-stack framework	Microframework (lightweight)
Philosophy	"Batteries included" (comes with lots of built-in features)	Minimalistic; add only what you need
Complexity	Better for complex, large-scale apps	Better for small to medium apps
Development Speed	Very fast (many things prebuilt)	Flexible but may need more setup time
Admin Panel	Built-in powerful admin interface	No built-in admin, must create manually
ORM	Built-in ORM (Django ORM)	No default ORM (can use SQLAlchemy separately)
Flexibility	Less flexible (convention-based)	Highly flexible (developer chooses structure)
Learning Curve	Steeper (more tools/features to learn)	Easier for beginners
Community Support	Very large and mature community	Growing and strong community
Use Cases	Large applications (e-commerce, enterprise apps, social networks)	Microservices, APIs, simple apps

5. Virtual Environment

Que.1
Importance of a Virtual Environment in Python Projects
What is a Virtual Environment?

- A virtual environment is an isolated workspace for a Python project.
- It keeps dependencies (libraries, frameworks) separate from other projects and from the global Python installation.

Why is it Important?

1. Dependency Management
   - Different projects may need different versions of the same library.
   - A virtual environment keeps each project's dependencies isolated to avoid version conflicts.
2. Avoids Polluting Global Python
   - Installing packages globally can mess up your main Python setup.
   - Virtual environments prevent clutter by installing libraries only for your project.
3. Easier Collaboration
   - Teams can replicate the same environment using a requirements.txt file.
   - Ensures everyone uses the exact same versions of libraries.

How to Create and Use a Virtual Environment (Quick Example):

```
# Create a virtual environment
python -m venv myenv

# Activate it
# On Windows:
myenv\Scripts\activate
# On Mac/Linux:
source myenv/bin/activate

# Install packages inside the environment
pip install django

# Save dependencies
pip freeze > requirements.txt

# Deactivate environment
deactivate
```

Que.2

Using venv (Built-in, Python 3.3+)

- venv is included with Python 3.3 and above.
- It's the default and recommended tool for creating virtual environments today.

```
# Step 1: Create a virtual environment
python -m venv myenv

# Step 2: Activate the environment
# On Windows:
myenv\Scripts\activate
# On macOS/Linux:
source myenv/bin/activate

# Step 3: Install packages inside the environment
pip install django

# Step 4: Deactivate when done
deactivate
```

Using virtualenv (Third-party package)

- virtualenv works with older Python versions and provides a few extra features (like faster environment creation).
- It must be installed separately.

```
# Step 1: Install virtualenv (if not already installed)
pip install virtualenv

# Step 2: Create a virtual environment
virtualenv myenv

# Step 3: Activate the environment
# On Windows:
myenv\Scripts\activate
# On macOS/Linux:
source myenv/bin/activate

# Step 4: Install packages inside
pip install flask

# Step 5: Deactivate when done
deactivate
```

6. Project and App Creation

Que.1

1. Set Up a Virtual Environment:

```
python -m venv env
```

Activate it:

```
env\Scripts\activate
```

2. Install Django:

```
pip install django
```

3. Create a Django Project:

```
django-admin startproject projectname
```

4. Create an App Inside the Project

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

5. Register the App in Project Settings

```
INSTALLED_APPS = [
    ...
    'blog',
]
```

6. Start Building the App

- Models in models.py
- Views in views.py
- URLs (create urls.py in app folder if needed)
- Admin site settings in admin.py
- Templates for front-end

Que.2
Understanding the Role of manage.py, urls.py, and views.py in Django

1. manage.py
What it is:
- A command-line tool for interacting with your Django project.

Role:
- Runs important Django commands like:
  - Running the server (python manage.py runserver)
  - Making migrations (python manage.py makemigrations)
  - Applying migrations (python manage.py migrate)
  - Creating superusers (python manage.py createsuperuser)
- Basically, it helps manage the project without manually touching a lot of Django internals.

2. urls.py
What it is:
- The URL dispatcher of Django.

Role:
- Maps URLs (web addresses) to views (Python functions or classes).
- Directs incoming browser requests to the right part of your app.

3. views.py
What it is:

- The file that contains the logic for what users see.

Role:

- Processes user requests and returns responses (like HTML pages, JSON data, etc.).
- Fetches data from the database (if needed), processes it, and sends it to templates.

7. MVT Pattern Architecture

Que.1
Request-Response Cycle in Django
Here’s how Django handles a typical web request:

1. User Sends a Request
    - The user types a URL in their browser or submits a form.
    - A request (usually HTTP) is sent to the Django server.
2. URL Dispatcher (URLs.py)
    - Django checks the request URL against its URL patterns (urls.py file).
    - It finds the matching View function/class based on the URL.
3. View Handles the Request
    - The View function/class gets control.
    - It can:
        - Interact with the Model (query the database).
        - Perform business logic (filter, validate, calculate).
        - Prepare data to be displayed

Step	What Happens
1.	User goes to www.example.com/books/
2.	Django looks in urls.py and finds that /books/ maps to book_list view.
3.	book_list() function queries the Book model to get all books.
4.	book_list() passes the list of books to the books_list.html template.
5.	Template renders a nice HTML page showing all books.
6.	HTML is sent back to the browser as the final response.

Visual Summary

[Browser Request] --> [URL Dispatcher] --> [View] --> [Model (if needed)]
↓
[View passes Data] --> [Template renders Page] --> [Response to Browser]

8. Django Admin Panel

Que.1
Django’s Admin Panel is a powerful feature that allows developers and site administrators to manage and interact with a website’s content through a web interface, without having to write any custom code for creating forms, tables, or views. It is automatically available when you create a Django project and can be highly customized to meet the needs of your site.

Features of Django Admin Panel:

1. Automatic Interface for Models
    - Django automatically generates a user-friendly interface based on your models (defined in models.py).
    - This allows site administrators to create, read, update, and delete entries in the database without writing any custom views or forms.
2. Customizable
    - You can customize how models appear in the admin interface.
    - Add filters, search functionality, and actions to make the admin panel more user-friendly.
3. Secure
    - Django's admin is secured with user authentication. Admin users are authenticated via the Django authentication system.
    - Permissions can be finely controlled to limit what a user can see or modify.

How to Enable and Use Django Admin Panel

1. Setting Up Django Admin

Step 1: Install Django If you haven't installed Django, do so using pip:

```
pip install django
```

Step 2: Create a Django Project If you don't already have a Django project, create one:

```
django-admin startproject myproject
cd myproject
```

Step 3: Create an App Apps in Django are modular components. Create an app where you will define your models:

```
python manage.py startapp myapp
```

Step 4: Register Models inadmin.py In the app's admin.py file, you need to register models so they appear in the admin panel.

Step 5: Create Superuser To access the admin panel, you need a superuser account. Create one by running:

```
python manage.py createsuperuser
```

Step 6: Run the Development Server Start Django’s development server to access the admin panel:

```
python manage.py runserver
```

2. Customizing the Admin Panel

Django allows you to customize how models are displayed in the admin interface. Here are some common customizations:

- List Display Control which fields are displayed in the list view.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'published_date')

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

- Search Functionality Allow searching for records based on specific fields.

```
class BookAdmin(admin.ModelAdmin):
    search_fields = ['title', 'author']

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

- Filters Add filters to the right sidebar to make it easier to find records.

```
class BookAdmin(admin.ModelAdmin):
    list_filter = ['author', 'published_date']

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

Admin Panel Access Control

Django’s admin panel uses permissions to control who can access what.

- Permissions per User You can assign different permissions to users, limiting their ability to add, change, or delete records.
  - For example, you can create a group (like "Editors") and give them permission to add and change records, but not delete them.
- User Groups and Permissions You can define groups and assign specific permissions to the groups for more fine-grained control.

Advantages of Django Admin

1. Speed and Efficiency
2. Developers can manage content easily without building custom admin interfaces, saving a lot of time during development.
3. Pre-built Features
4. The admin comes with a lot of pre-built functionality, such as user authentication, session management, and model validation, right out of the box.
5. Extensibility
6. While it's useful for many basic content management tasks, you can also extend it to fit complex project needs.

Que.2
Customizing the Django admin interface is a powerful way to manage your database records more efficiently and to tailor the experience to your needs. Django provides several options to enhance the usability and appearance of the admin panel for better content management.

1. Customizing Model Display in the Admin Panel
Django's ModelAdmin class allows you to customize how models are displayed and interacted with in the admin interface.

a. List Display
You can control which fields to display in the list view (the default overview page for the model).

```
# admin.py

from django.contrib import admin
from .models import Book

class BookAdmin(admin.ModelAdmin):
    # Display fields in the list view
    list_display = ('title', 'author', 'published_date',
'is_available')

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

b. List Filters
Add filters to the right sidebar to make it easier for admins to filter records by certain fields.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'published_date',
'is_available')
    list_filter = ('author', 'published_date')

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

c. Search Functionality
Enable search functionality so that admins can easily search for records based on specific fields.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'published_date',
'is_available')
    search_fields = ['title', 'author']

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

d. Pagination
By default, the Django admin paginates records to avoid overwhelming users with too many entries. You can change the number of records per page.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'published_date',
'is_available')
    list_per_page = 20  # Show 20 records per page

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

2. Customizing the Model Form in the Admin Panel
Sometimes, you may want to change the way fields are displayed or organized in the form view (when creating or editing a record).
a. Fieldsets
You can group fields together to organize them into sections in the form.

```
class BookAdmin(admin.ModelAdmin):
    fieldsets = (
        ('Basic Information', {
            'fields': ('title', 'author')
        }),
        ('Publication Info', {
            'fields': ('published_date',)
        }),
        ('Availability', {
            'fields': ('is_available',)
        }),
    )

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

Control the order in which fields appear on the form.

```
class BookAdmin(admin.ModelAdmin):
    fields = ('title', 'author', 'published_date', 'is_available')

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

c. Inline Model Admin

If you have a related model (such as a foreign key), you can use inline models to edit related models directly in the parent model's form.

```
class BookInline(admin.StackedInline):
    model = Book
    extra = 1  # Show 1 extra empty form for adding new
records

class AuthorAdmin(admin.ModelAdmin):
    inlines = [BookInline]

admin.site.register(Author, AuthorAdmin)
```

3. Customizing the Admin Interface with Actions
Django allows you to define custom actions that admins can perform on selected records.
a. Custom Actions
You can define actions that can be applied to multiple records at once.
```
def mark_books_as_unavailable(modeladmin, request,
queryset):
    queryset.update(is_available=False)
    modeladmin.message_user(request, "Books marked as
unavailable.")

mark_books_as_unavailable.short_description = "Mark
selected books as unavailable"

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

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

b. Action Messages
You can show success messages when actions are completed.
```
def mark_books_as_unavailable(modeladmin, request,
queryset):
    queryset.update(is_available=False)
    modeladmin.message_user(request, "Books marked as
unavailable.")

mark_books_as_unavailable.short_description = "Mark
selected books as unavailable"
```

4. Customizing the Admin Dashboard

If you need to create a custom dashboard for the admin interface, you can extend it with custom views or integrate third-party libraries like django-admin-tools.

a. Custom Admin Views

You can add custom views to the admin interface.

```
from django.urls import path
from django.http import HttpResponse
from django.contrib import admin

def custom_view(request):
    return HttpResponse("This is a custom admin view!")

class CustomAdminSite(admin.AdminSite):
    def get_urls(self):
        urls = super().get_urls()
        custom_urls = [
            path('custom/', self.admin_view(custom_view)),
        ]
        return custom_urls + urls

admin_site = CustomAdminSite(name='custom_admin')
```

5. Customizing the Admin Appearance

You can further customize the look and feel of the admin interface by overriding templates or using third-party libraries.

a. Overriding Admin Templates
You can override the default admin templates to change the HTML structure or apply custom CSS styles.

```
# Create a directory: templates/admin/
# Override the base template or any other templates

# Example: custom_base.html
```

b. Using Third-Party Libraries
There are several libraries like django-suit, django-grappelli, and django-admin-interface that provide beautiful, modern designs and enhanced functionality for the Django admin interface.

9. URL Patterns and Template Integration

Que.1

1. Basic URL Configuration

In Django, URL patterns are defined in the urls.py file of each app (and the project’s main urls.py).

```
from django.urls import path
from . import views  # Import views from the
current app

urlpatterns = [
    # Define your URL patterns here
]
```

2. Setting Up URL Patterns for Views

To route requests to different views, you need to map URLs to view functions in your urls.py.

Example of a Basic URL Mapping:

Let's assume we have a view called home in the views.py of an app called myapp.

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Welcome to the
homepage!")
```

```
from django.urls import path
from . import views  # Importing views from the
current app

urlpatterns = [
    path('', views.home, name='home'),  # Maps the
empty URL to the 'home' view
]
```

3. Include URL Patterns from Other Apps
For larger projects, it’s common to include URL patterns from individual apps. In the main urls.py file of the project, you can include the URLs of your apps.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),  # Django's
default admin page
    path('myapp/', include('myapp.urls')),  # Include
URLs from myapp
    path('blog/', include('blog.urls')),  # Include
URLs from blog
]
```

4. Using Dynamic URL Patterns (With Parameters)
Sometimes, you need to capture parts of the URL as parameters and pass them to the view function. Django makes this easy with dynamic URL patterns using path converters.
Example: Capturing URL Parameters
Let’s say you want to display a page for a specific blog post identified by its id. You can capture the id from the URL and pass it to the view function.

```
from django.http import HttpResponse

def post_detail(request, post_id):
    return HttpResponse(f"Viewing post with ID:
{post_id}")
```

```
myapp/urls.py
from django.urls import path
from . import views

urlpatterns = [
    # Define a URL pattern with a dynamic
parameter `post_id`
    path('post/<int:post_id>/', views.post_detail,
name='post_detail'),
]
```

5. Path Converters (Django URL Types)
Django provides several built-in path converters to capture URL parts. Here’s a list of the most commonly used converters:

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

urlpatterns = [
    path('post/<int:post_id>/', views.post_detail,
name='post_detail'),
    path('author/<str:author_slug>/',
views.author_detail, name='author_detail'),
    path('files/<path:file_path>/', views.file_detail,
name='file_detail'),
]
```

6. Reverse URL Lookup (Using reverse() and {% url %})
Django allows you to reference URL patterns in your views or templates using reverse URL lookup.
In Views:
You can use reverse() to dynamically generate a URL.

```
from django.urls import reverse
from django.http import HttpResponseRedirect

def redirect_to_post(request, post_id):
    # Generate the URL for the 'post_detail' view
dynamically
    url = reverse('post_detail', args=[post_id])
    return HttpResponseRedirect(url)
```

Que.2

1. Setting Up Templates in Django

Django looks for HTML templates in specific directories. The typical process for setting up templates involves creating a templates directory within your app and configuring it in the project settings.

Step 1: Create a Templates Directory

- In each app, you can create a templates folder. Inside this folder, create a subfolder with the same name as your app. This helps to organize templates.

Step 2: Configure the Template Directory in settings.py

Ensure that Django knows where to find the templates by adding the following configuration to your project’s settings.py file:

python

2. Rendering Dynamic Content Using Views and Templates

Django allows you to render a template and pass dynamic data (context) to it using views. The context consists of variables that the template can access and use to dynamically generate content.

Step 1: Create a View Function to Render a Template

A view function is responsible for receiving HTTP requests, processing data (if needed), and returning an HTTP response with the rendered HTML template.

Here’s an example of a view that renders a template and passes dynamic content (such as a list of blog posts) to the template:

```
from django.shortcuts import render
from .models import BlogPost

def home(request):
    # Fetching data from the database (e.g., list of blog posts)
    posts = BlogPost.objects.all()

    # Rendering the 'home.html' template with the context (dynamic data)
    return render(request, 'myapp/home.html', {'posts': posts})
```

Step 2: Define the Template to Display the Data
Now that you have passed data to the template, you can use Django’s template language to display it in the HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport"
content="width=device-width, initial-scale=1.0">
   <title>Blog Home</title>
</head>
<body>
   <h1>Welcome to the Blog</h1>

   <h2>Recent Posts</h2>
   <ul>
      {% for post in posts %}
         <li>
            <h3>{{ post.title }}</h3>
            <p>{{ post.content }}</p>
            <small>Posted on: {{ post.created_at }}
</small>
         </li>
      {% empty %}
         <li>No blog posts available.</li>
      {% endfor %}
   </ul>
</body>
</html>
```

3. Linking the View with a URL Pattern
Next, you need to map a URL to the view that will render this template.
Step 1: Define the URL Pattern
In the myapp/urls.py file, map the URL path to the view:

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

urlpatterns = [
    path('', views.home, name='home'),  # Maps the
root URL to the home view
]
```

Step 2: Include App URLs in Project-Level URLs
In the project_name/urls.py file, include your app's URLs so that the root URL can be properly resolved.
project_name/urls.py

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),  # Include URLs
from myapp
]
```

4. Using Template Inheritance
Django templates support template inheritance, which allows you to define a base template with common structure (like headers, footers) and then extend it in other templates.
Step 1: Create a Base Template
myapp/templates/myapp/base.html
html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>{% block title %}My Website{% endblock %}</title>
</head>
<body>
    <header>
        <h1>My Website</h1>
        <nav>
            <a href="/">Home</a>
            <a href="/about/">About</a>
        </nav>
    </header>

    <main>
        {% block content %}{% endblock %}
    </main>

    <footer>
        <p>&copy; 2025 My Website</p>
    </footer>
</body>
</html>
```

Step 2: Extend the Base Template
myapp/templates/myapp/home.html

```
{% extends 'myapp/base.html' %}

{% block title %}Blog Home{% endblock %}

{% block content %}
    <h2>Recent Posts</h2>
    <ul>
        {% for post in posts %}
            <li>
                <h3>{{ post.title }}</h3>
                <p>{{ post.content }}</p>
                <small>Posted on: {{ post.created_at }}
</small>
            </li>
        {% empty %}
            <li>No blog posts available.</li>
        {% endfor %}
    </ul>
{% endblock %}
```

10. Form Validation using JavaScript

1. Basic Structure of a Form
Let’s start by creating a simple HTML form for collecting user information, such as name, email, and password.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-
scale=1.0">
   <title>Form Validation</title>
</head>
<body>
   <h2>Sign Up</h2>
   <form id="signupForm">
      <label for="name">Name:</label>
      <input type="text" id="name" name="name">
      <span id="nameError" style="color:red;"></span><br><br>

      <label for="email">Email:</label>
      <input type="email" id="email" name="email">
      <span id="emailError" style="color:red;"></span><br><br>

      <label for="password">Password:</label>
      <input type="password" id="password" name="password">
      <span id="passwordError" style="color:red;"></span><br><br>

      <button type="submit">Submit</button>
   </form>

   <script src="formValidation.js"></script>
</body>
</html>
```

2. JavaScript for Form Validation

Now, let’s add JavaScript to handle validation. We’ll check if:

- Name is not empty.
- Email is in a valid email format.
- Password meets the minimum length requirement.

```
// Select the form and input fields
const form = document.getElementById('signupForm');
const nameInput = document.getElementById('name');
const emailInput = document.getElementById('email');
const passwordInput = document.getElementById('password');

// Select error message elements
const nameError = document.getElementById('nameError');
const emailError = document.getElementById('emailError');
const passwordError = document.getElementById('passwordError');

// Form submit event
form.addEventListener('submit', function(event) {
  let isValid = true;  // Assume the form is valid initially

  // Clear previous error messages
  nameError.textContent = '';
  emailError.textContent = '';
  passwordError.textContent = '';

  // Validate name (non-empty)
  if (nameInput.value.trim() === '') {
    nameError.textContent = 'Name is required';
    isValid = false;
  }

  // Validate email (check if it's a valid email format)
  const emailPattern = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
  if (!emailPattern.test(emailInput.value)) {
    emailError.textContent = 'Please enter a valid email';
    isValid = false;
  }

  // Validate password (at least 6 characters)
  if (passwordInput.value.length < 6) {
    passwordError.textContent = 'Password must be at least 6 characters long';
    isValid = false;
  }

  // If form is not valid, prevent submission
  if (!isValid) {
    event.preventDefault();
  }
});
```

-

Explanation of JavaScript Code:

1. Event Listener on Form Submit:
    - We attach an event listener to the form’s submit event. When the form is submitted, it triggers the validation process.
2. Validation Logic:
    - Name Validation: We check if the name input is empty. If it is, an error message is displayed.
    - Email Validation: We use a regular expression to check if the email follows the standard email format (e.g., user@example.com).
    - Password Validation: We check if the password length is at least 6 characters. If not, an error message is shown.
3. Prevent Form Submission:
    - If any of the fields fail validation, we set isValid to false and use event.preventDefault() to stop the form from submitting.

3. Improving User Experience with Immediate Feedback

It’s often a good idea to give users immediate feedback while they’re typing, rather than waiting until they submit the form. You can achieve this by adding event listeners on individual input fields to validate each field as the user types.

Add Immediate Feedback for Real-Time Validation

Modify the JavaScript to validate each field when the user finishes typing:

```
// Validate name as the user types
nameInput.addEventListener('input', function() {
   if (nameInput.value.trim() === '') {
      nameError.textContent = 'Name is required';
   } else {
      nameError.textContent = '';
   }
});

// Validate email as the user types
emailInput.addEventListener('input', function() {
   const emailPattern = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
   if (!emailPattern.test(emailInput.value)) {
      emailError.textContent = 'Please enter a valid email';
   } else {
      emailError.textContent = '';
   }
});

// Validate password as the user types
passwordInput.addEventListener('input', function() {
   if (passwordInput.value.length < 6) {
      passwordError.textContent = 'Password must be at least 6
characters long';
   } else {
      passwordError.textContent = '';
   }
});
```

4. Additional Validation Features

You can add more advanced validations, such as:

- Matching Passwords: If you have a "Confirm Password" field, you can compare the two passwords to make sure they match.
- Custom Patterns: You can validate phone numbers, addresses, etc., with regular expressions.
- Required Fields: You can mark certain fields as "required" and ensure they are not empty.

```
<label for="confirmPassword">Confirm Password:</label>
<input type="password" id="confirmPassword"
name="confirmPassword">
<span id="confirmPasswordError" style="color:red;"></span>
<br><br>
```

11. Django Database Connectivity (MySQL or SQLite)

Que.1
To connect Django to a database, you’ll need to configure the DATABASES setting in Django’s settings.py file. Django supports multiple databases, including SQLite (which is the default) and MySQL.

1. Connecting Django to SQLite (Default Database)
By default, Django is configured to use SQLite, a lightweight relational database. It requires no installation or configuration other than what is provided in the settings.py file.
Step 1: Check the Default SQLite Configuration
Open your settings.py file and look for the DATABASES setting:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',  # Use SQLite
        'NAME': BASE_DIR / 'db.sqlite3',  # SQLite database
file
    }
}
```

Step 2: Migrate Database
Once you’ve configured the database, you can create the necessary database tables by running Django's migrations.

```
python manage.py migrate
```

2. Connecting Django to MySQL Database
If you want to use MySQL instead of SQLite, you'll need to do some additional configuration, including installing the MySQL client for Python.
Step 1: Install MySQL Client
Before configuring Django to use MySQL, you need to install the MySQL client library. Run the following command:

```
pip install mysqlclient
```

```
pip install pymysql
```

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

Step 2: Update DATABASES in settings.py
Next, you need to modify the DATABASES setting in settings.py to use MySQL.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',  # Use MySQL backend
        'NAME': 'your_database_name',          # Name of the MySQL
database
        'USER': 'your_database_user',          # MySQL username
        'PASSWORD': 'your_database_password',  # MySQL password
        'HOST': 'localhost',                   # Database host (use 'localhost'
or IP address)
        'PORT': '3306',                        # MySQL default port
    }
}
```

Step 3: Create the Database in MySQL
Before running migrations, make sure the database exists in MySQL. You can create the database using MySQL commands:

```
CREATE DATABASE your_database_name;
```

Step 4: Migrate Database
After configuring the database, run the following migration command to create the necessary tables in MySQL:

```
python manage.py migrate
```

3. Troubleshooting Common Issues

Issue: MySQL Database Not Found

- Make sure the database you specified in settings.py exists in MySQL. You can create a database manually via MySQL or use Django’s migrate command if it’s missing.

Issue: MySQL Client Not Installed

- Ensure that you've installed mysqlclient or PyMySQL. The installation process might be tricky on some systems, especially Windows. If you encounter installation issues, consult the official Django documentation for the latest instructions or consider using Docker to containerize your MySQL setup.

Issue: Connection Errors

- Double-check the HOST, USER, PASSWORD, and PORT fields in settings.py to ensure they are correct.
- If MySQL is running on a remote server, ensure that your MySQL server allows connections from your IP address (check firewall settings).

Que.2

1. Models in Django ORM

In Django, each model represents a table in the database. A model is a Python class that subclasses django.db.models.Model. The class attributes represent the table columns.

Example of a simple model:

```
from django.db import models

class Student(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    email = models.EmailField()
    birth_date = models.DateField()

    def __str__(self):
        return f'{self.first_name} {self.last_name}'
```

2. Creating Database Records (Inserting Data)

You can create and insert new records into the database using the model's .save() method.

```
# Create a new student object
student = Student(first_name="John", last_name="Doe",
email="john.doe@example.com", birth_date="2000-01-01")

# Save the object to the database
student.save()
```

3. Querying the Database (Selecting Data)

Django ORM provides several methods for retrieving records. The basic query method is Model.objects.all().

```
# Get all student records
students = Student.objects.all()
for student in students:
    print(student.first_name, student.last_name)
```

4. Updating Database Records

To update a record, first retrieve it, modify the fields, and then save it again.

```
python
# Update a student's email
student = Student.objects.get(id=1)
student.email = "new.email@example.com"
student.save()
```

5. Deleting Records

To delete a record, retrieve it and call .delete() on the object.

```
# Delete a specific student
student = Student.objects.get(id=1)
student.delete()
```

6. Aggregations and Annotations

Django ORM supports aggregation, which is useful for calculating summaries like counts, averages, sums, etc.

7. Related Data (Foreign Keys, Many-to-Many Relationships)
Django ORM also supports relationships between models using ForeignKey, ManyToManyField, and OneToOneField.
Example: Foreign Key
Let’s say you have a Course model related to Student:

```
class Course(models.Model):
    name = models.CharField(max_length=100)

class Enrollment(models.Model):
    student = models.ForeignKey(Student,
on_delete=models.CASCADE)
    course = models.ForeignKey(Course,
on_delete=models.CASCADE)

# Get all courses a student is enrolled in
student = Student.objects.get(id=1)
courses = student.enrollment_set.all()
for course in courses:
    print(course.name)
```

8. Raw SQL Queries
If the Django ORM cannot handle a complex query, you can execute raw SQL queries using django.db.connection.

```
from django.db import connection

def run_custom_query():
    with connection.cursor() as cursor:
        cursor.execute("SELECT * FROM app_student WHERE
last_name = %s", ['Doe'])
        result = cursor.fetchall()
    return result
```

12. ORM and QuerySets

1. What is a QuerySet?
A QuerySet is a collection of database queries that are used to fetch and manipulate data. When you perform any database operation in Django (like filtering, ordering, etc.), you get a QuerySet object.
A QuerySet is lazy, meaning it doesn't hit the database until you actually evaluate it. This is a crucial feature for optimizing database queries.

```
from myapp.models import Student

# Fetch all Student objects
students = Student.objects.all()
```

2. Lazy Evaluation of QuerySets
As mentioned, QuerySets are evaluated lazily. This means that a QuerySet does not execute the query until it is actually needed (i.e., when it is evaluated).
For example:

```
students = Student.objects.filter(last_name="Doe")  #
QuerySet is created, but no query is run yet
students_list = list(students)  # Now the query is
executed when we evaluate the QuerySet
```

3. Querying with QuerySets
You can perform various database operations using QuerySets like filtering, ordering, and retrieving data.
Here are some examples of common operations:
Fetching All Records

```
students = Student.objects.all()  # Fetch all records from
the Student table
```

Filtering Records

```
# Get students with last name 'Doe'
students = Student.objects.filter(last_name="Doe")

# Get students whose first name starts with 'J'
students =
Student.objects.filter(first_name__startswith="J")
```

4. QuerySet Methods for Manipulating Data

You can chain multiple methods together on a QuerySet to refine your queries and get more specific data. Here are some commonly used methods:

.exclude()

Returns a QuerySet that excludes results based on the specified conditions.

```
# Get all students whose last name is not 'Doe'
students = Student.objects.exclude(last_name="Doe")
```

.order_by()

Orders the results based on one or more fields.

```
# Get all students ordered by first name (ascending)
students = Student.objects.all().order_by('first_name')

# Get all students ordered by first name (ascending) and
last name (descending)
students = Student.objects.all().order_by('first_name', '-
last_name')
```

.distinct()
Removes duplicate records from the QuerySet.

```
# Get distinct first names from the students
students = Student.objects.values('first_name').distinct()
```

5. QuerySet Evaluation: Methods that Force Evaluation
Several methods force the evaluation of a QuerySet. They trigger the execution of the database query and return the result.

.all()
Returns all records from the model.

```
students = Student.objects.all()  # Evaluates the QuerySet and fetches all students
```

.count()
Returns the number of records in the QuerySet.

```
count = Student.objects.filter(last_name="Doe").count()
```

.first() / .last()
Returns the first or last record of the QuerySet.

```
first_student = Student.objects.all().first()
last_student = Student.objects.all().last()
```

.get()
Returns a single object. If the query returns more than one result, it raises a MultipleObjectsReturned exception.

```
student = Student.objects.get(id=1)
```

.exists()
Checks whether a QuerySet has any results.

```
has_students =
Student.objects.filter(last_name="Doe").exists()
```

6. QuerySet and Aggregation
Django provides powerful aggregation tools for performing calculations like counts, sums, averages, etc., on your QuerySets.
Using aggregate() for Aggregations

```
from django.db.models import Count, Avg

# Count the number of students
total_students = Student.objects.aggregate(Count('id'))

# Calculate the average age of students
average_age = Student.objects.aggregate(Avg('age'))
```

7. Relationship Queries
Django ORM also supports querying related models using ForeignKey, ManyToManyField, and OneToOneField.
Example: ForeignKey Relationship

```
# Assume we have a Course model related to Student
through ForeignKey
course = Course.objects.get(id=1)
students_in_course = course.student_set.all()
```

8. Modifying Data with QuerySets
Django allows you to update and delete records directly using QuerySets.
Updating Records

```
# Update a student's last name
Student.objects.filter(id=1).update(last_name="NewLast
Name")

# Delete a specific student
student = Student.objects.get(id=1)
student.delete()

# Delete students with a specific last name
Student.objects.filter(last_name="Doe").delete()
```

9. Raw SQL Queries
If you need to execute complex queries that can't be easily expressed using Django’s ORM, you can execute raw SQL queries using django.db.connection.

```
from django.db import connection

def run_custom_query():
    with connection.cursor() as cursor:
        cursor.execute("SELECT * FROM myapp_student
WHERE last_name = %s", ['Doe'])
        result = cursor.fetchall()
    return result
```

10. Performance Considerations
    - Lazy Evaluation: Since QuerySets are lazy, Django ORM optimizes queries, fetching data only when needed.
    - Database Hits: Be mindful of database hits. If you chain too many queries or run queries in loops, it may lead to multiple database queries. Use methods like select_related() and prefetch_related() to optimize database access.

12. ORM and QuerySets

Que1.
1. Django Forms
Forms in Django are used to handle user input from web pages. Django provides a form class (django.forms.Form) to define and handle forms.
Basic Structure of a Django Form
A Form in Django is defined as a Python class, and each field corresponds to an input element in the HTML form.

2. Handling Forms in Views
Once you define a form, you need to handle it in your views. Typically, forms are handled in POST requests, where the user submits the form data.
Displaying and Validating Forms in Views
In your views, you can create an instance of the form, validate it, and save the data if necessary.

3. Form Field Types and Validation
Django provides a variety of form fields with built-in validation:

- CharField: For text input (e.g., names, titles).
- EmailField: For email addresses.
- IntegerField: For integers.
- DateField: For dates.
- ChoiceField: For selecting from a list of choices.

Each field can also have validators that ensure data integrity. For example, you can set a field to be required or to have a specific length.

4. Django Authentication
Authentication in Django refers to identifying users, verifying their credentials, and providing access to protected parts of your application.
User Authentication Flow
1. Login: Verifying if a user exists with the correct password.
2. Logout: Logging out the user from the application.
3. User Registration: Allowing new users to create an account.

5. Django User Model
Django comes with a built-in User model for managing users. This model includes fields like:
- username
- password
- email
- first_name
- last_name

You can interact with the User model via django.contrib.auth.models.User.

6. Login and Logout Views
Django provides login and logout functionality through built-in views.
Login View
You can use Django’s built-in LoginView to manage user login.

7. User Registration

You can create a user registration form using Django’s built-in UserCreationForm.

User Creation Form

Django provides a pre-built form for creating a user: django.contrib.auth.forms.UserCreationForm.

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

def register_view(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()  # Create the user
            return redirect('login')
    else:
        form = UserCreationForm()

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

8. Protecting Views with Login Required
You can protect views that should only be accessible to authenticated users using the @login_required decorator.

```
from django.contrib.auth.decorators import
login_required

@login_required
def protected_view(request):
    return render(request, 'protected.html')
```

9. Password Management
Django provides easy ways to manage user passwords:

- Set Password: user.set_password('new_password')
- Check Password: user.check_password('password')

You can also use password reset views for users to reset their passwords if they forget.

Que.2

1. Set Up the Project and App

First, ensure you have a Django project and an app where you'll implement the authentication system.

```
# Create a Django project
django-admin startproject myproject

# Navigate to the project directory
cd myproject

# Create a new Django app
python manage.py startapp accounts
```

Now, add your accounts app to the INSTALLED_APPS in your settings.py:

```
# myproject/settings.py

INSTALLED_APPS = [
    # other apps
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'accounts',
]
```

2. User Registration (Sign-Up)
Django provides a built-in form called UserCreationForm that simplifies user registration.
Creating a Registration Form View
In your accounts/views.py, create a view to handle user registration:

```
# accounts/views.py
from django.shortcuts import render, redirect
from django.contrib.auth.forms import
UserCreationForm
from django.contrib import messages

def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()  # Save the new user
            messages.success(request, 'Account created
successfully')
            return redirect('login')  # Redirect to login page
    else:
        form = UserCreationForm()

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

```
<!-- accounts/templates/accounts/register.html -->
{% block content %}
  <h2>Register</h2>
  <form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Register</button>
  </form>
{% endblock %}
```

URL Routing for Registration

In your accounts/urls.py, create a URL route for the registration view:

```
# accounts/urls.py
from django.urls import path
from . import views

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

Add this URL to your main project's urls.py:

```
# myproject/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('accounts.urls')),
]
```

3. User Login

Django provides a built-in LoginView that handles user login.

Using Django’s Built-In LoginView

In your urls.py, add a route for the login view:

```
# accounts/urls.py
from django.contrib.auth.views import LoginView

urlpatterns = [
    path('login/', LoginView.as_view(), name='login'),
]
```

This view will look for a template called registration/login.html by default. Create this template in your templates/registration/login.html:

```
<!-- accounts/templates/registration/login.html -->
{% block content %}
  <h2>Login</h2>
  <form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Login</button>
  </form>
{% endblock %}
```

5. Password Management
Django provides built-in views for password management, including password reset and password change functionalities.
Password Reset (Forgot Password)
To allow users to reset their password, you can use the built-in PasswordResetView.

```
# accounts/urls.py
from django.contrib.auth import views as auth_views

urlpatterns = [
    # other paths
    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'),
]
```

1. Template for Password Reset:

Django will look for the following templates in the templates/registration directory:

- password_reset_form.html — The form to input an email.
- password_reset_done.html — A message telling the user that the password reset link was sent.
- password_reset_confirm.html — The form to enter a new password after clicking the link.
- password_reset_complete.html — A success message after the password has been reset.

Password Change

For logged-in users who want to change their password, you can use the built-in PasswordChangeView.

1. URL Configuration:

```
# accounts/urls.py
from django.contrib.auth import views as auth_views

urlpatterns = [
    # other paths
    path('password_change/',
auth_views.PasswordChangeView.as_view(),
name='password_change'),
    path('password_change/done/',
auth_views.PasswordChangeDoneView.as_view(),
name='password_change_done'),
]
```

1. Template for Password Change:
2. You’ll need to create a template for the password change form:

```
<!--
accounts/templates/registration/password_change_form.
html -->
{% block content %}
  <h2>Change Password</h2>
  <form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Change Password</button>
  </form>
{% endblock %}
```

6. Protect Views Using Login Required

You can use the @login_required decorator to protect views that should only be accessible by logged-in users.

```
# accounts/views.py
from django.contrib.auth.decorators import
login_required

@login_required
def profile_view(request):
    return render(request, 'accounts/profile.html')
```

```
# accounts/urls.py
from django.urls import path
from . import views
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('register/', views.register, name='register'),
    path('login/', auth_views.LoginView.as_view(),
name='login'),
    path('logout/', auth_views.LogoutView.as_view(),
name='logout'),
    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'),
    path('password_change/',
auth_views.PasswordChangeView.as_view(),
name='password_change'),
    path('password_change/done/',
auth_views.PasswordChangeDoneView.as_view(),
name='password_change_done'),
]
```

14. CRUD Operations using AJAX

1. Introduction to AJAX
AJAX enables web pages to communicate with the server and retrieve data without refreshing the page. With AJAX, JavaScript can send HTTP requests (like GET, POST) to a server, which processes them and returns a response. The JavaScript on the page then processes the response, allowing dynamic updates to the page content.

2. Setting Up Your Django Project
Make sure your Django project is set up with an app. For this tutorial, let's assume you have an app named ajaxapp. If not, you can create one with:

```
python manage.py startapp ajaxapp
```

```
# settings.py
INSTALLED_APPS = [
    # other apps
    'ajaxapp',
]
```

3. Using AJAX for Asynchronous Requests
AJAX requests are typically made using JavaScript. Here, we will cover a simple example of how to use AJAX to send data to the server and update the webpage dynamically.
Example: Fetching Data Using AJAX (GET Request)

1. Define a View to Handle the Request
In views.py, create a view that will return data when requested.

```
# ajaxapp/views.py
from django.http import JsonResponse
from .models import Note
from .forms import NoteForm

def add_note(request):
    if request.method == 'POST' and request.is_ajax():
        form = NoteForm(request.POST)
        if form.is_valid():
            note = form.save()
            return JsonResponse({
                'id': note.id,
                'title': note.title,
                'content': note.content
            })
        else:
            return JsonResponse({'error': 'Invalid data'},
status=400)
```

2. Create the Note Form
In forms.py, create a form for adding notes.

```
# ajaxapp/forms.py
from django import forms
from .models import Note

class NoteForm(forms.ModelForm):
    class Meta:
        model = Note
        fields = ['title', 'content']
```

3. Create URL Route for Adding Notes

In urls.py, map the URL for adding a note.

```
# ajaxapp/urls.py
from . import views

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

4. JavaScript to Send Data (POST Request)

In the HTML template, add a form for adding new notes and use AJAX to send the data to the server.

```html
<!-- ajaxapp/templates/ajaxapp/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>AJAX Example</title>
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
</head>
<body>
  <h1>Notes</h1>

  <!-- Form for adding a new note -->
  <input type="text" id="title" placeholder="Note Title">
  <textarea id="content" placeholder="Note Content"></textarea>
  <button id="add-note">Add Note</button>

  <ul id="notes-list"></ul>

  <script>
    // Fetch notes when the page is loaded
    $(document).ready(function() {
      $.ajax({
        url: '/ajax/get_notes/',
        method: 'GET',
        success: function(response) {
          $('#notes-list').empty();
          response.notes.forEach(function(note) {
            $('#notes-list').append(`
              <li>
                <strong>${note.title}</strong><br>
                ${note.content}
              </li>
            `);
          });
        }
      });
    });

    // Add new note using AJAX
    $('#add-note').click(function() {
      const title = $('#title').val();
      const content = $('#content').val();

      $.ajax({
        url: '/ajax/add_note/',
        method: 'POST',
        data: {
          'title': title,
          'content': content,
          'csrfmiddlewaretoken': '{{ csrf_token }}',
        },
        success: function(response) {
          $('#notes-list').append(`
            <li>
              <strong>${response.title}</strong><br>
              ${response.content}
            </li>
          `);
          $('#title').val('');
          $('#content').val('');
        },
        error: function(xhr, status, error) {
          console.error("Error adding note:", error);
        }
      });
    });
  </script>
</body>
</html>
```

15. Customizing the Django Admin Panel

1. Customizing the Model Admin Class
The most common way to customize the Django admin panel is by overriding the ModelAdmin class for a specific model. You can define how the model should be displayed and how it behaves in the admin interface.

a) Customizing the List Display
By default, Django's admin will display all the fields of a model in the list view. You can specify which fields should appear by s# admin.py

```
from django.contrib import admin
from .models import Note

class NoteAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'created_at', 'updated_at')

admin.site.register(Note, NoteAdmin)
```

etting the list_display attribute.

b) Adding Filters
You can add filters to the list view to make it easier to filter records based on specific fields.

```
class NoteAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'created_at', 'updated_at')
    list_filter = ('created_at',)

admin.site.register(Note, NoteAdmin)
```

c) Search Functionality
You can enable a search box for easy searching of specific fields in the list view using the search_fields attribute.

```
class NoteAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'created_at')
    search_fields = ('title', 'content')

admin.site.register(Note, NoteAdmin)
```

2. Customizing Forms in the Admin Panel
You can customize the forms used for adding or editing models in the admin panel by using ModelForm. This allows you to modify the layout and validation of form fields.
a) Using ModelForm to Customize Fields
Create a custom form by subclassing forms.ModelForm and linking it to your ModelAdmin.

```
# forms.py
from django import forms
from .models import Note

class NoteForm(forms.ModelForm):
    class Meta:
        model = Note
        fields = ['title', 'content']

    # Add custom validation
    def clean_title(self):
        title = self.cleaned_data.get('title')
        if 'Django' not in title:
            raise forms.ValidationError('Title must contain "Django"')
        return title
```

b) Customizing Fieldsets
You can group form fields in the admin interface using the fieldsets attribute to define how they are arranged.

```
class NoteAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': ('title', 'content')
        }),
        ('Date Information', {
            'fields': ('created_at', 'updated_at'),
            'classes': ('collapse',)
        }),
    )

admin.site.register(Note, NoteAdmin)
```

3. Customizing the Admin Interface with Inlines
If you have related models, you can use InlineModelAdmin (e.g., TabularInline or StackedInline) to manage related objects within the same page.

```
from django.contrib import admin
from .models import Comment, Note

class CommentInline(admin.TabularInline):
    model = Comment
    extra = 1  # How many empty forms to display

class NoteAdmin(admin.ModelAdmin):
    inlines = [CommentInline]

admin.site.register(Note, NoteAdmin)
```

b) Using StackedInline
StackedInline displays related models in a stacked (vertically aligned) format.

```
class CommentInline(admin.StackedInline):
    model = Comment
    extra = 1

class NoteAdmin(admin.ModelAdmin):
    inlines = [CommentInline]

admin.site.register(Note, NoteAdmin)
```

4. Customizing List Filters
You can create custom filters in the admin panel to allow for more complex filtering logic beyond the default fields.
a) Using list_filter with Custom Filters
Django allows you to create custom filters for models in the admin interface.

```
from django.contrib import admin
from django.utils.translation import gettext_lazy as _
from .models import Note

class CustomDateFilter(admin.SimpleListFilter):
    title = _('Created this year')
    parameter_name = 'created_this_year'

    def lookups(self, request, model_admin):
        return (
            ('yes', _('Yes')),
            ('no', _('No')),
        )

    def queryset(self, request, queryset):
        if self.value() == 'yes':
            return queryset.filter(created_at__year=2025)
        return queryset

class NoteAdmin(admin.ModelAdmin):
    list_filter = (CustomDateFilter,)

admin.site.register(Note, NoteAdmin)
```

5. Customizing Admin Templates
If you want full control over how the admin interface looks, you can override the default admin templates.
a) Overriding Templates
Django allows you to override the default admin templates. For instance, if you want to change how the change form of a model is rendered, you can override the change_form.html template.

1. Create a folder templates/admin in your project directory.
2. Copy the default template from django/contrib/admin/templates/admin/change_form.html.
3. Modify it as needed.

This approach gives you complete control over the HTML and CSS of the admin panel.
b) Custom Admin CSS/JS
You can also add custom CSS or JavaScript to the admin interface by using the Media class inside your ModelAdmin class.

```
class NoteAdmin(admin.ModelAdmin):
    class Media:
        css = {
            'all': ('css/custom_admin.css',)
        }
        js = ('js/custom_admin.js',)

admin.site.register(Note, NoteAdmin)
```

6. Custom Admin Actions
You can define custom admin actions to apply certain operations to multiple objects at once.
a) Defining Custom Actions

```
from django.contrib import admin
from .models import Note

def mark_notes_as_read(modeladmin, request, queryset):
    queryset.update(status='read')

mark_notes_as_read.short_description = 'Mark selected notes as read'

class NoteAdmin(admin.ModelAdmin):
    actions = [mark_notes_as_read]

admin.site.register(Note, NoteAdmin)
```

```
from django.contrib import admin
from .models import Note

def mark_notes_as_read(modeladmin, request, queryset):
    queryset.update(status='read')

mark_notes_as_read.short_description = 'Mark selected notes as read'

class NoteAdmin(admin.ModelAdmin):
    actions = [mark_notes_as_read]

admin.site.register(Note, NoteAdmin)
```

7. Using Third-Party Packages

There are several third-party packages that help you further customize the Django admin. Some popular ones include:

- django-grappelli: Provides a more modern and user-friendly interface for the Django admin.
- django-admin-interface: Adds extra styling and features to the Django admin panel.
- django-model-utils: Adds additional features to Django models and the admin interface.

16. Payment Integration Using Paytm

1. Overview of Payment Gateway Integration
Payment gateways, such as Paytm, provide a secure way for users to make payments via credit cards, debit cards, wallets, and net banking. When integrating a payment gateway, the process generally involves:

1. Creating an account with the payment gateway provider (Paytm in this case).
2. Configuring the payment gateway in your Django project by setting up API keys, credentials, and payment URLs.
3. Setting up the backend logic to generate payment requests and process responses.
4. Handling the frontend to allow users to securely submit their payment details.

2. Steps to Integrate Paytm into Django
Below are the main steps to integrate Paytm as a payment gateway in your Django project.
Step 1: Create a Paytm Merchant Account
To integrate Paytm as a payment gateway, you must first sign up for a Paytm Merchant Account.

- Visit the Paytm Developer Console: Paytm Developer
- Create a new merchant account and get access to Merchant ID, Merchant Key, and other credentials required for integration.
- You will also get access to the Sandbox Environment (for testing) and Live Environment (for production).

Step 2: Install Paytm Python SDK
Paytm provides a Python SDK that simplifies the integration process. You can install the SDK using pip:
bash

Step 3: Set Up the Paytm API Credentials
You will need to store your Paytm credentials in your Django settings file (settings.py). Add the following:

```
# settings.py

PAYTM_MERCHANT_ID = 'your_merchant_id'
PAYTM_MERCHANT_KEY = 'your_merchant_key'
PAYTM_WEBSITE = 'your_website'
PAYTM_CHANNEL_ID = 'your_channel_id'
PAYTM_INDUSTRY_TYPE = 'your_industry_type'

PAYTM_CALLBACK_URL =
'http://yourdomain.com/payment/callback/'  # Callback URL
after payment
```

Step 4: Create Payment Views and URL Patterns
Now, create the views for handling the payment process (request generation and payment status verification) in your Django project.
a) Create a Payment Request View
This view will create a payment request, generate a checksum, and redirect the user to Paytm’s payment gateway.

```
# views.py
from django.shortcuts import render, redirect
from django.conf import settings
import random
from paytmchecksum import PaytmChecksum

def payment_gateway(request):
    if request.method == 'POST':
        # Fetch payment details from the form or session
        order_id = str(random.randint(100000, 999999))  #
Example: generating order ID
        customer_email = request.POST['email']
        amount = request.POST['amount']

        # Payment details
        params = {
            'MID': settings.PAYTM_MERCHANT_ID,
            'ORDER_ID': order_id,
            'CUST_ID': customer_email,
            'INDUSTRY_TYPE_ID': settings.PAYTM_INDUSTRY_TYPE,
            'CHANNEL_ID': settings.PAYTM_CHANNEL_ID,
            'TXN_AMOUNT': amount,
            'WEBSITE': settings.PAYTM_WEBSITE,
            'CALLBACK_URL': settings.PAYTM_CALLBACK_URL,
        }

        # Generate checksum
        checksum = PaytmChecksum.generate_checksum(params,
settings.PAYTM_MERCHANT_KEY)
        params['CHECKSUMHASH'] = checksum

        # Create payment URL
        paytm_url = 'https://securegw-stage.paytm.in/order/process'
        return render(request, 'paytm/payment_redirect.html',
{'params': params, 'paytm_url': paytm_url})

    return render(request, 'paytm/payment_form.html')
```

b) Create Payment Form (Frontend)
In the template payment_form.html, create a form where the user enters payment details.

```
<!-- payment_form.html -->
<form method="POST" action="{% url 'payment_gateway' %}">
   {% csrf_token %}
   <input type="email" name="email" placeholder="Your Email"
required>
   <input type="text" name="amount" placeholder="Amount to
Pay" required>
   <button type="submit">Proceed to Pay</button>
</form>
```

c) Create Payment Redirect Template
This template will redirect the user to the Paytm payment gateway with the necessary details and checksum.

```
<!-- payment_redirect.html -->
<form method="POST" action="{{ paytm_url }}"
name="paytm_form">
   {% for key, value in params.items %}
      <input type="hidden" name="{{ key }}" value="{{ value }}">
   {% endfor %}
   <button type="submit">Pay Now</button>
</form>

<script type="text/javascript">
   document.paytm_form.submit();
</script>
```

Step 5: Handle Payment Response (Callback URL)
After the payment is processed, Paytm will send a callback to the URL you specified in the CALLBACK_URL. You need to handle this callback and verify the payment status.

```
# views.py
from django.http import JsonResponse
from django.conf import settings
from paytmchecksum import PaytmChecksum

def payment_callback(request):
    paytm_params = request.POST

    # Verify checksum
    checksum = paytm_params.get('CHECKSUMHASH')
    is_valid_checksum =
PaytmChecksum.verify_checksum(paytm_params,
settings.PAYTM_MERCHANT_KEY, checksum)

    if is_valid_checksum:
        # Process payment success
        if paytm_params['RESPCODE'] == '01':  # Payment success
            order_id = paytm_params['ORDERID']
            txn_id = paytm_params['TXNID']
            # Update your database to mark the order as successful
            return JsonResponse({'status': 'success', 'txn_id': txn_id,
'order_id': order_id})
        else:
            # Payment failed
            return JsonResponse({'status': 'failure'})
    else:
        return JsonResponse({'status': 'checksum verification
failed'})
```

Step 6: Configure the URL for Payment Callback
Add the following URL pattern for the callback URL in your urls.py:

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

urlpatterns = [
    path('payment/', views.payment_gateway,
name='payment_gateway'),
    path('payment/callback/', views.payment_callback,
name='payment_callback'),
]
```

3. Test Payment Integration
Before moving to the production environment, you must test the integration using Paytm’s Sandbox Environment.

1. Use Sandbox credentials provided by Paytm for testing purposes.
2. Ensure that the CALLBACK_URL is set correctly and the payment processing flow works without issues in the testing environment.
3. Once you’re confident that everything works, switch to Live credentials and update the CALLBACK_URL to the production URL.

4. Security Considerations
When integrating a payment gateway like Paytm into your Django project, ensure the following security measures:

- Use SSL: Make sure your website is served over HTTPS to encrypt sensitive payment data.
- Checksum Verification: Always verify the checksum provided by Paytm to ensure the integrity of the payment response.
- CSRF Protection: Protect forms and APIs with Django’s built-in CSRF protection.
- Sensitive Information: Never expose your Paytm credentials (merchant ID, key) in the frontend. Always keep them in settings.py or environment variables.

17. GitHub Project Deployment

Step 1: Initialize Git in Your Django Project
If you haven't already initialized Git in your project, you'll need to do that first.

```
cd /path/to/your/django/project
git init
git status
```

Step 2: Create a .gitignore File
You’ll want to avoid pushing certain files or directories to GitHub (like database files, virtual environments, or Django settings containing secrets). This is done by adding these files to a .gitignore file.

1. Create a .gitignore file in the root of your project.
2. You can use a template for Django projects. Here's a typical .gitignore for Django:

```
*.pyc
__pycache__
db.sqlite3
/venv/
*.log
*.pot
*.py[cod]
.env
.vscode
*.bak
*.swp
media/
static/
```

Add the .gitignore file to Git:

```
git add .gitignore
```

Step 3: Add Project Files to Git
Add all the files you want to push to GitHub (excluding those ignored by .gitignore):

```
git add .
```

Step 4: Commit the Changes
Commit the changes to Git with a meaningful commit message:

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

Step 5: Create a Repository on GitHub

1. Log in to GitHub and go to the GitHub website.
2. Create a new repository by clicking the "+" icon at the top right of the page and selecting "New repository".
3. Fill in the repository details, such as:
    - Repository name
    - Description (optional)
    - Choose whether the repository should be public or private
    - Initialize it without a README (since you already have one locally)
4. Click Create repository.

Step 6: Connect Your Local Repository to GitHub
After creating the repository, GitHub will provide you with the URL of the repository (either HTTPS or SSH).

1. Add the remote URL to your local Git repository:

```
git remote add origin
https://github.com/yourusername/yourrepository.git
```

Verify the remote URL:

```
git remote -v
```

Step 7: Push the Changes to GitHub
Finally, push your code to GitHub:

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

Step 8: Verify the Push
Go back to your GitHub repository in a web browser, and you should see all your project files have been uploaded successfully.

Step 9: Set Up Git for Future Commits
For future commits and pushes, you don’t need to repeat the above steps. Here’s the workflow:

1. Make changes to your project.
2. Add files to Git staging area:

```
git add .
git commit -m "Describe your changes"
git push origin main
```

18. Live Project Deployment (PythonAnywhere)

Step 1: Prepare Your Django Project
Before deploying, make sure your Django project is ready for production.
1. Settings Configuration for Production
In your Django settings.py, make sure to configure the following for production:
DEBUG = False

Set ALLOWED_HOSTS to allow your domain or server's IP address:
ALLOWED_HOSTS = ['your-username.pythonanywhere.com']

- Set up the database (if using a database like MySQL, make sure to configure it in DATABASES).
- Static Files: Django needs to be configured to handle static files in production (e.g., CSS, JavaScript).
- In settings.py, make sure you have:

```
STATIC_URL = '/static/'

# Add this line for production
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

Media Files: Similarly, set up media file handling:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

2. Create a requirements.txt File
Generate a requirements.txt file to list all your dependencies. This helps in installing the required packages on the server.
In your project directory, run:

```
pip freeze > requirements.txt
```

Step 2: Sign Up and Set Up a PythonAnywhere Account

1. Create an Account: Sign up at PythonAnywhere.
2. Login: After signing up, log in to your PythonAnywhere account.

Step 3: Create a New Web App on PythonAnywhere

1. Create a New Web App:
   - From your PythonAnywhere dashboard, go to the "Web" tab.
   - Click on the "Add a new web app" button.
   - Choose the Python version (ensure it's the version you are using in your project).
   - Select "Manual configuration" (since Django is a manual setup).
2. Set Up the Web App: PythonAnywhere will ask for your web app's source code location. You'll later upload your code to the server.

Step 4: Upload Your Django Project

1. Using Git (Preferred Method):
   - If your project is in a Git repository (e.g., on GitHub), you can clone the project directly to PythonAnywhere.
   - In the "Files" tab on PythonAnywhere, navigate to the directory where you want to place your project.
   - Open a Bash console and run:

```
git clone https://github.com/yourusername/yourproject.git
```

Step 5: Install Dependencies on PythonAnywhere

1. Open a Bash Console on PythonAnywhere.
2. Navigate to your project directory.
3. Create a virtual environment (if you don't already have one):

```
python3.8 -m venv myenv
source myenv/bin/activate
pip install -r requirements.txt
```

Step 6: Set Up Database (if applicable)
If your project uses a database (like SQLite, MySQL, or PostgreSQL), you'll need to configure it on PythonAnywhere.

1. Create and Configure a Database:
   - PythonAnywhere provides MySQL databases (or you can use SQLite if you prefer).
   - If using MySQL, go to the "Databases" tab and create a new database.
   - Configure the DATABASES setting in settings.py with your MySQL credentials (provided by PythonAnywhere).

Step 7: Collect Static Files
Django needs to collect static files (e.g., images, CSS, JavaScript) and place them in a folder where they can be served in production.

```
python manage.py collectstatic
```

Step 8: Configure the Web App

1. Configure the WSGI File:
   - In the "Web" tab on PythonAnywhere, you'll find an option to edit your WSGI file.
   - Set the WSGI path to point to your project’s wsgi.py file.

```
import os
import sys

path = '/home/yourusername/yourproject'
if path not in sys.path:
    sys.path.insert(0, path)

os.environ['DJANGO_SETTINGS_MODULE'] = 'yourproject.settings'
from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

Step 9: Test and Launch the Project

- Test the Project: After completing all the configurations, you should be able to visit your web app’s URL (e.g., yourusername.pythonanywhere.com) and see your Django project running.
- Monitor Logs:
- Check the "Error log" and "Access log" in the "Web" tab for any errors or issues during the deployment process.

Step 10: Additional Configuration

1. Set up email configuration: If your app sends emails, configure an email backend (e.g., Gmail, SendGrid) in your settings.py.
2. Security settings: Make sure to configure security settings like SSL for HTTPS (you may need a custom domain for this), and Django's CSRF settings.

19. Social Authentication

To set up social login options (like Google, Facebook, and GitHub) in Django using OAuth2, we will use the django-allauth package. It simplifies the process of integrating social logins by handling OAuth2 authentication flow and providing pre-configured authentication backends for many popular social providers.

Step 1: Install Dependencies
First, install the django-allauth package:
pip install django-allauth

Step 2: Configure Authentication Backends
You need to configure Django's authentication backends to include the django-allauth backend. Update the AUTHENTICATION_BACKENDS setting in your settings.py:

```
# settings.py

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

Step 3: Configure Social Account Settings in settings.py
You will need to configure keys for each social login provider (Google, Facebook, GitHub). Create a SOCIAL_AUTH_* key for each provider.
For Google OAuth2:

1. Go to the Google Developer Console.
2. Create a new project.
3. Enable Google+ API or Google Identity Services.
4. Create OAuth 2.0 credentials and get your Client ID and Client Secret.
5. Add the redirect URI:

```
http://localhost:8000/accounts/google/login/callback/  # For local dev
https://yourdomain.com/accounts/google/login/callback/  # For production
```

Update the settings in settings.py:

```
# settings.py

SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = '<your-google-client-id>'
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = '<your-google-client-secret>'
```

Step 4: Add URLs for Social Authentication
Now, you need to include the URLs for social login and authentication. In your urls.py, include the allauth URLs.

```
# urls.py

from django.urls import path, include

urlpatterns = [
    # Include allauth URLs for login, signup, social authentication, etc.
    path('accounts/', include('allauth.urls')),
]
```

Step 5: Run Migrations
Run Django migrations to create the necessary database tables for the authentication system:

```
python manage.py migrate
```

Step 6: Create Templates for Social Authentication
django-allauth provides default templates for login and signup, but you can override them to customize the UI.
For example, to display social login buttons (like Google, Facebook, and GitHub) on the login page, you can modify the login.html template.

1. Create a custom login.html file in templates/account/ if you want to customize it:

```
<!-- templates/account/login.html -->

{% load socialaccount %}
{% providers_media_js %}
<h2>Login</h2>

<!-- Google login button -->
{% provider_login_url 'google' as google_login_url %}
<a href="{{ google_login_url }}">Login with Google</a>

<!-- Facebook login button -->
{% provider_login_url 'facebook' as facebook_login_url %}
<a href="{{ facebook_login_url }}">Login with Facebook</a>

<!-- GitHub login button -->
{% provider_login_url 'github' as github_login_url %}
<a href="{{ github_login_url }}">Login with GitHub</a>
```

Step 7: Test the Social Login

1. Run the Django development server:

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

Step 8: Deploy to Production

After testing the social login locally, deploy your project to a production environment (e.g., Heroku, AWS, PythonAnywhere).

1. Update Redirect URIs on the provider's OAuth application page to point to the live production URLs (e.g., https://yourdomain.com/accounts/google/login/callback/).
2. Set up HTTPS (SSL) for your production site, as most social providers require it for OAuth2 authentication.
3. Set up environment variables for sensitive keys (e.g., Google Client ID/Secret, Facebook App ID/Secret, GitHub Client ID/Secret) in your production environment for better security.

20. Google Maps API

Integrating Google Maps API into a Django project allows you to display interactive maps, geolocation data, markers, and much more. In this guide, we will cover the steps to integrate the Google Maps API in Django for various functionalities such as displaying maps, adding markers, and enabling location-based features.

Step 1: Get Your Google Maps API Key

Before using Google Maps in your Django project, you need to obtain an API key.

1. Go to the Google Cloud Console.
2. Create a new project or select an existing one.
3. Enable the Google Maps JavaScript API and any other relevant APIs (e.g., Geocoding API, Places API, etc.) by navigating to APIs & Services > Library.
4. Go to APIs & Services > Credentials and create a new API key. Google will generate an API key that you will use to authenticate your application.

Step 2: Install Required Packages

While the Google Maps API can be used directly with JavaScript in the frontend, you might want to install some additional Django packages if you're handling geolocation or geocoding (e.g., converting addresses to coordinates and vice versa).

- django-google-maps can be used for integrating maps and geocoding in your Django project.
- geopy is another option for geocoding and reverse geocoding operations in Python.

```
pip install geopy
```

Step 3: Set Up Google Maps API in Templates
You will need to include Google Maps' JavaScript API in your template to interact with the maps. You can add it to your HTML templates.

1. Include Google Maps API: Add this <script> tag in your base template (e.g., base.html).

```
<!-- Add this script tag inside your <head> or just before the closing </body> tag -->
<script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&callback=initMap" async defer></script>
```

1. Create a JavaScript Function to Initialize the Map:

Here's a basic example of how to initialize the map with a marker:

```
<!-- In your template, e.g., maps.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Google Maps</title>
    <style>
        #map {
            height: 500px;
            width: 100%;
        }
    </style>
</head>
<body>
    <h1>Google Maps Integration</h1>
    <div id="map"></div>

    <script>
        let map;

        // Function to initialize the map
        function initMap() {
            const location = { lat: 37.7749, lng: -122.4194 };  // Example: San Francisco
coordinates

            map = new google.maps.Map(document.getElementById('map'), {
                center: location,
                zoom: 12,
            });

            // Add a marker at the location
            const marker = new google.maps.Marker({
                position: location,
                map: map,
                title: "San Francisco",
            });
        }

        // Call the initMap function once the API is loaded
        window.initMap = initMap;
    </script>
</body>
</html>
```