DJANGO INTERVIEW QUESTIONS (Beginner + Intermediate)

- Prepared by Eswar Chandra Yadlapalli

1) What is Django?

Answer:
Django is a **high-level, open-source Python web framework** used to build secure, scalable, and maintainable web applications.
It provides a **toolkit of all components** required for web development such as URL routing, templates, ORM, authentication, admin panel, form handling, sessions, security features, etc.

Django encourages **rapid development**, meaning developers can build production-ready applications very fast because many features are pre-built.

It follows **MVT (Model–View–Template)** design pattern, which cleanly separates business logic, UI, and database structure.
In many parts, it also resembles the MVC architecture followed by other frameworks.

Interview Follow-Up Questions Derived From the Above Answer

1.1) What is a web application?

A web application is a software application that runs on a web server and is accessed through a browser (Chrome, Firefox).
Example: Facebook, Gmail, Amazon.

A web application involves:

- Frontend (HTML, CSS, JS)
- Backend (Python, Django, Database)
- Communication through HTTP/HTTPS

1.2) What is a framework? Why do we need it?

A framework is a pre-built structure or set of tools that simplifies development.
Django gives:

- URL router
- Template engine
- Database ORM
- Authentication system

Without framework → developer must build everything manually.

1.3) What are design patterns?

Design patterns are **standard solutions** to commonly occurring software design problems. They provide structured ways to design systems so code becomes reusable, flexible, and maintainable.

Examples:

- Singleton Pattern
- Factory Pattern
- Observer Pattern
- MVC / MVT Pattern

1.4) What is MVC? What is MVT? How are they different?

MVC (Model–View–Controller):
Used by many frameworks like Spring, Rails, Laravel.
Controller controls the flow.

MVT (Model–View–Template):
Used by Django.
Here, Django acts as the Controller automatically, so developer only writes Model, View & Template.

Difference Summary:

MVC	MVT
Developer writes Controller	Django handles controller internally
View shows output	Template shows output
Used in Java, PHP, Ruby	Used in Python Django

1.5) Why is Django called a "high-level" framework?

Because it handles:

- Security
- Sessions
- Database ORM
- User authentication
- Admin dashboard
- Middleware
- Caching
- Templating

All automatically → developer focuses only on business logic.

1.6) Why is Django secure? Explain Django security features.

Some major security features:

- CSRF protection
- XSS protection
- SQL Injection prevention through ORM
- Clickjacking protection
- Password hashing
- Session security

Django provides secure middleware out-of-the-box.

1.7) Why is Django used for rapid development?

Because:

- ORM reduces SQL writing
- Admin panel is auto-created
- Built-in user authentication
- Pre-defined directory structure
- Template engine ready
- Fast deployment options

✔ 2) What is a Django Project and Django App?

Answer:

A **Project** is the complete web application — a container that holds all configurations, settings, and apps.
Example: `amazon.com` full website → 1 project.

An **App** is a modular part of a project designed to handle a specific functionality.
Examples:

- login app
- payment app
- search app
- cart app

One project → many apps.

❓ Follow-Up Questions From This Answer

2.1) Why do we divide a project into multiple apps?

To maintain:

- Modularity
- Reusability
- Maintainability
- Clear separation of features

2.2) Can we reuse Django apps in other projects?

Yes, Django apps are fully reusable.
For example, your login app can be used in another project without changes.

2.3) What happens if we don't configure an app in INSTALLED_APPS?

Django will not recognize:

- Models
- Templates
- Signals
- Admin settings
- Migrations

App behaves like a normal folder, not a Django module.

✔ 3) Explain the Django Request-Response Cycle.

Answer:

When a user opens a URL:

1. Browser sends request → Django server

2. Django checks urls.py

3. URL matches → View function/class executes

4. View interacts with Model if needed

5. View sends data to Template

6. Template renders HTML

7. Response is sent back to browser

❓ Follow-Up Questions

3.1) What happens if URL is not found?

Django throws **404 error**.

3.2) What happens if view returns nothing?

Django throws **500 Server Error**.

3.3) What is HttpResponse?

A basic method to send text output.

✔ 4) What is manage.py and why do we use it?

Answer:

`manage.py` is the command-line utility file automatically created by Django.
It allows us to run important project commands like:

- runserver
- migrate
- createsuperuser
- startapp
- shell

It automatically sets Django settings environment before executing anything.

❓ Follow-Up Questions

4.1) What is the difference between `django-admin` and `manage.py`?

- `django-admin` → general Django tool
- `manage.py` → project-specific tool (knows project settings)

✔ 5) What is settings.py? Explain important settings.

Answer:

`settings.py` is the **configuration file** for the whole project.
It contains:

- Database settings
- Installed apps
- Middleware
- Templates directory
- Static files
- Security keys
- Allowed hosts

❓ Follow-Up Questions

5.1) What is SECRET_KEY?

Used for encryption, sessions, CSRF, cookies.

5.2) What is DEBUG?

Enables debugging mode; should be False in production.

5.3) What is ALLOWED_HOSTS?

Security setting to allow only specific domains.

✔ 6) What is the role of Templates in Django?

Answer:

Templates define **HTML UI** of your web page.
Django template engine supports:

- Dynamic values rendering
- Conditional logic
- Loops
- Template inheritance
- Filters

❓ Follow-Up Questions

6.1) What is template inheritance?

Allows base.html + child templates → reduces repeated HTML.

6.2) Why do we keep templates inside each app?

For modular project structure.

Q1) What is a Model in Django?

✅ Answer (Interviewer-Convincing):

A **Model** in Django is a **Python class** that represents a **database table**.
Django uses its own ORM (Object Relational Mapper) so you don't write SQL manually.
Each attribute defined in the model becomes a **column**, and each instance becomes a **row**.

🔑 Key Features

- Automatically creates tables using migrations
- Eliminates manual SQL queries
- Supports relationships (One-To-One, One-To-Many, Many-To-Many)
- Ensures database consistency and validation

🧪 Simple Real-Time Example

You are building a **Notes App**.
You create a model:

```
class Note(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
```

Django creates a SQL table behind the scenes:

| id | title | content | created_at |

This is the beauty of ORM.

🔁 Follow-Up Questions

1.1) What happens internally when you create a model?

- Django registers the model
- Creates a migration file
- Converts migration → SQL
- Executes SQL → creates table

1.2) Why is ORM better than raw SQL?

- Avoids SQL injection
- Write Python instead of SQL

- DB independent (SQLite, PostgreSQL, MySQL…)
- Faster development

1.3) Can two models communicate?

Yes, using:

- `ForeignKey`
- `OneToOneField`
- `ManyToManyField`

Q2) What is a View in Django?

Answer:

A **View** is a Python function or class that:

- Accepts request data
- Processes logic
- Communicates with models
- Returns a response (HTML, JSON, redirect, file, API result)

Real-time Example

User enters `/home` → view returns HTML.

```
def home(request):
    return render(request, "home.html")
```

Follow-Up Questions

2.1) Types of Views?

- Function-Based Views (FBV)
- Class-Based Views (CBV)

2.2) Which is better?

- FBV → Simple logic
- CBV → Reusable, scalable (ListView, DetailView)

2.3) What is render()?

A helper function that returns HTML response.

Q3) What is a Template? What is its role?

Answer:

Templates are **HTML files** used to build the UI of your website.
Django's template engine allows:

- Insert dynamic data using `{{ }}`
- Write loops and conditions using `{% %}`
- Use template inheritance (base.html → child.html)
- Reuse frontend blocks
- Display data from views/models

Real-Time Example

`home.html`:

```
<h1>Welcome {{ user.username }}</h1>
```

Follow-Up Questions

3.1) What is template inheritance?

A concept where `base.html` contains repeated UI
(logo, navbar, footer)
and each page extends it.

3.2) Why keep templates inside each app?

- Easy to manage
- App-level modularity
- Avoids naming conflicts

3.3) What is the default template engine?

Django Template Engine

Q4) What is the “Control” in Django? (MVC vs MVT)

✅ Answer:

Django follows **MVT architecture**.

MVC Term	Django Term	Responsible
Model	Model	Database
View	Template	UI
Controller	View (+ Django internal engine)	Logic

So, Django takes care of most controller parts — URL routing, request handling, middleware.

🔥 Real-Time Example

When URL hits your project:

- Django routes to correct view
- Executes middleware
- Applies authentication
- Processes context
- Returns response

🔁 Follow-Ups

4.1) Why did Django choose MVT instead of MVC?

Because Django auto-handles controller tasks internally.

4.2) What is the advantage of MVT?

Clear separation:
UI = Template

Logic = View
DB = Model

Q5) How to create a Virtual Environment?

Commands (Windows):

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

Commands (Mac/Linux):

```
python3 -m venv venv
source venv/bin/activate
```

Follow-Up Questions

5.1) Why use virtual environment?

- Package isolation
- Different project versions
- Avoid system package conflicts

5.2) How to deactivate?

```
deactivate
```

Q6) What files are created after creating a venv? Explain their usage.

Inside `venv/`:

File/Folder	Usage
Scripts/	executables (activate, pip, python)
Lib/	installed packages
pyvenv.cfg	environment configuration

Q7) How to create a Django app?

Command:

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

Q8) What files are created inside Django App? Explain each

File	Purpose
models.py	Database tables
views.py	Request handling logic
urls.py	URL routing (manual creation)
admin.py	Admin panel integration
apps.py	App configuration
tests.py	Unit tests
migrations/	Database changes

Q9) What is the default database of Django? Explain in detail.

Answer:

Django uses **SQLite3** as the default database.

Why SQLite?

- Zero configuration
- File-based (db.sqlite3)
- Perfect for development
- Lightweight and fast

Real-time benefit:

Even without installing MySQL/PostgreSQL, you immediately start development.

Q10) What is a URL? Explain its components.

Example:

```
https://example.com:443/products?id=10
```

Part	Meaning
protocol	https
domain/base address	example.com
port	443
path	/products
query string	id=10

Q11) VS Code Extensions Required for Django

- Python
- Django Snippets
- Pylance
- IntelliSense
- GitLens
- Prettier (optional)

Q12) URL Flow Diagram

```
Browser Request
      ↓
project/urls.py
      ↓
app/urls.py
      ↓
views.py
      ↓
model (if needed)
      ↓
template.html
      ↓
Response to Browser
```

Q13) What is Version Control System? Types?

Answer:

A VCS tracks code changes over time.

Types:

- **Centralized** – SVN
- **Distributed** – Git

Q14) Explain Git, GitHub, Git Repository with Real-Time Examples

Git

Local version control installed on laptop.

GitHub

Cloud hosting for Git repositories.

Repository

A project folder where Git tracks changes.

Real-Time Example (Office Scenario)

You write code → commit → push to GitHub → teammate pulls → updates → pushes.

Q15) Basic Git Commands

```
git init
git clone
git add .
git commit -m "msg"
git push
git pull
git status
```

Q16) Types of URL Mapping in Django

- **Specific URL** – direct path
- **Dynamic URL** – path with parameters

Q17) Types of Template Directories

- App-level templates
- Project-level templates

Q15) What is a Django Project vs Django App? Explain with real-time example.

✅ Answer (Detailed):

A **Django Project** is the **entire application configuration**, containing settings, main URL routing, database config, installed apps, authentication, middleware, etc.

A **Django App** is a **small module inside the project** that performs a specific function.

✅ Real-time Example

Imagine **Flipkart** = Django Project

Inside Flipkart:

Function	Django App
Accounts/Login	accounts app
Products page	products app
Cart	cart app
Orders	orders app
Payments	payments app

Project = container
Apps = building blocks

✅ Follow-up Questions

15.1) Can a project have multiple apps?

Yes, every real project has 5–20 apps.

15.2) Can an app belong to multiple projects?

Yes — reusable apps like authentication, blog plugins, captcha modules.

Q16) Explain the request–response cycle in Django.

✅ Answer (Multi-step Explanation):

1User enters URL → Browser sends HTTP Request
2Django receives it via **WSGI/ASGI**
3Request passes through **Middleware** layers
4Django checks **project/urls.py**
5It forwards to **app/urls.py**
6Mapped view is executed
7 View interacts with Model if needed
8 View returns Template or Response
9 Response moves back through Middleware
 Browser receives final output

✅ Real-time Example

User enters:

```
/products/12
```

Flow:

Browser → Django → urls → products app → view → DB → template → browser

---

✅ Follow-up Questions

16.1) What is Middleware?

A layer that processes request/response globally (Authentication, CSRF, Sessions).

16.2) What is WSGI / ASGI?

- WSGI → synchronous (older version)
- ASGI → asynchronous (Fast, WebSockets)

Q17) What is manage.py? Why is it important?

✅ Answer:

`manage.py` is a utility script that provides commands to manage your Django project.

Responsibilities

- Start server
- Apply migrations
- Create apps
- Manage database
- Create superuser
- Collect static files

🧪 Real-time Example

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

🔁 Follow-ups

17.1) What happens if manage.py is deleted?

You lose access to Django commands.

17.2) Where does settings.py get loaded from?

`manage.py` loads settings using:

```
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "project.settings")
```

Q18) What is settings.py? Explain important sections.

🔑 Key Sections

- INSTALLED_APPS → registered apps
- DATABASES → DB connection
- MIDDLEWARE → request filters
- TEMPLATES → HTML configuration
- STATIC_URL → CSS/JS setup
- AUTH_PASSWORD_VALIDATORS → security rules

- TIME_ZONE & LANGUAGE_CODE → localization

Real-time example

When you add a new app:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'myapp',
]
```

If not added → View/Model not registered → Error.

Follow-ups

18.1) Can we have multiple settings.py files?

Yes (dev, staging, production).

18.2) What is DEBUG=True?

Shows detailed error messages (never use in production).

Q19) What are Installed Apps in Django?

Answer:

`INSTALLED_APPS` is a list of all apps Django should load when the project starts.

Includes:

- Built-in apps (admin, auth, sessions)
- Custom apps
- Third-party apps (rest_framework, crispy_forms)

Real-time Example

When you install DRF:

```
pip install djangorestframework
```

Then add:

```
INSTALLED_APPS = ['rest_framework']
```

Now Django loads it.

Follow-up

19.1) What if app is not added?

- Models wont' migrate
- Templates not loaded
- URLs may break

Q20) What are Migrations in Django?

Answer:

Migrations are Django's way of:

- Tracking DB changes
- Converting Python models → SQL
- Version controlling database schema

Commands:

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

Example

You add a new field:

```
age = models.IntegerField()
```

Django creates migration and updates database.

Follow-up

20.1) Where are migrations stored?

Inside each app folder: `migrations/0001_initial.py`

20.2) What if you delete the migration folder?

DB won't sync → errors.

Q21) What is URL Dispatcher in Django?

Answer:

Django uses **URL dispatcher** to match requested URLs with corresponding views using patterns.

Example in `urls.py`:

```
path('home/', views.home)
```

Follow-up

21.1) Types of URL patterns?

- Static URLs (`home/`)
- Dynamic URLs (`product/<int:id>/`)
- Regex URLs (older Django versions)

Q22) What is HttpResponse? When do we use it instead of render()?

Answer:

`HttpResponse` returns plain text or raw data.

Example:

```
return HttpResponse("Hello")
```

Use when:

- You need to return API text
- Return file
- Return JSON (old way)

`render()` is used when sending HTML templates.

Q23) What is the difference between Static files and Media files?

Static Files

- CSS
- JS
- Images (site icons)

Media Files

- User-uploaded content (profile photos, documents)

Real-time:

Flipkart → product images = media
Flipkart CSS → static

Q24) What is Django Admin? Why is it powerful?

Answer:

Django Admin is a fully automated backend UI for managing database records without writing code.

Features:

- CRUD operations
- Filters
- Search
- Permissions
- Custom dashboards

Follow-Up

24.1) How do you register a model in admin?

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

Q25) Explain CSRF Token in Django. Why is it needed?

Answer:

CSRF (Cross-Site Request Forgery) token protects POST forms from malicious attacks.

Every POST form must include:

```
{% csrf_token %}
```

Without CSRF → attacker can submit forms on your behalf.

Q26) What is the role of Apps.py?

Stores app configuration.

Example:

```
class NotesConfig(AppConfig):
    name = 'notes'
```

Q27) What is the difference between Django and Flask?

Django	Flask
Full framework	Micro framework
Batteries included	Minimal
Admin panel	No admin
ORM available	ORM optional
MVT	No fixed architecture

Q28) What is MTV in Django?

Model – Template – View
Django's design pattern.

Q29) What is `__str__()` in Model?

Defines how model objects appear in admin/debug.

Q30) What is the difference between GET and POST?

GET	POST

	GET	POST
Reads data	Sends data	
Parameters in URL	Hidden	
Bookmarkable	Not bookmarkable	
Less secure	More secure	

Q31) What is the difference between GET and POST methods?

✅ Answer (Detailed):

`GET` and `POST` are two HTTP request methods used to communicate between browser ↔ server.

🔹 GET Method

- Data is sent in the URL
- Not secure
- Limited data length
- Used for reading/fetching data

Example:

```
/search?keyword=laptop
```

🔹 POST Method

- Data sent in request body
- Secure
- Used for sensitive forms
- Used for insert/update/delete

Example:
Submitting a login form.

❓ Follow-up Questions

31.1) Where do you use GET in real time?

- Search bar
- Filter dropdown
- Pagination

31.2) Where do you use POST?

- Login
- Signup
- Payment
- File upload

31.3) Can GET send sensitive data?

❌ No. Passwords, OTP, payments should never use GET.

Q32) What is urlpatterns in Django?

Answer:

`urlpatterns` is a list of URL patterns that map incoming URLs to views.

Example:

```
urlpatterns = [
    path("", views.home),
    path("login/", views.login_view),
]
```

Django reads this list **top to bottom**.

Follow-up

32.1) What if two URLs match the same pattern?

Django uses the **first match**.

32.2) Where should urlpatterns exist?

- project/urls.py
- app/urls.py

Q33) What is `include()` in Django URLs?

Answer:

`include()` allows creating separate URL files for each app and linking them into main project.

Example:

```
path("accounts/", include("accounts.urls"))
```

This keeps code modular.

Follow-up

33.1) Why do we use include()?

- Cleaner project
- Easy debugging
- Multiple teams can work independently

Q34) What is a Template Directory? How many types?

Types:

1. **Project-level templates folder**
2. **App-level templates folder**

Example Structure

```
project/
    templates/   ← project level
app/
    templates/appname/   ← app level
```

Follow-up

34.1) Which one is recommended?

App-level templates → better modularity.

34.2) Why do we write `templates/appname/file.html`?

To avoid name conflicts.

Q35) What is the difference between project-level and app-level templates?

Project-Level	App-Level
Centralized location	App-specific UI
Good for shared layouts	Modular
Can be messy in big apps	Best practice

Q36) What is Django Shell?

Answer:

A Python shell with Django context loaded.

Command:

```
python manage.py shell
```

Use to:

- Test queries
- Create objects
- Debug ORM

Follow-Up

36.1) Example: Create object in shell

```
from notes.models import Note
Note.objects.create(title="Test", content="Hello")
```

Q37) What is QuerySet in Django?

Answer:

A QuerySet is a collection of objects retrieved from database.

Examples:

```
Note.objects.all()
Note.objects.filter(user=1)
Note.objects.get(id=10)
```

Follow-up

37.1) Is QuerySet lazy or eager?

Lazy — executes only when needed.

Q38) What is ORM? Why does Django use ORM?

Answer:

ORM converts Python code into SQL queries.

Advantages:

- No SQL required
- Secure
- DB independent
- Faster development

Follow-up

38.1) Is ORM faster than SQL?

No — SQL is faster, ORM is more convenient.

Q39) What is the difference between render() and redirect()?

render()	redirect()
Returns HTML template	Sends browser to another URL
Used for UI	Used after POST, login, save

Example:

```
return redirect("home")
```

Q40) What is Django Path Converters?

Example:

```
path("product/<int:id>/", views.product)
```

Supported types:

- int
- str
- slug
- uuid
- path

Follow-up

40.1) What is slug?

SEO-friendly URL text:

```
python-basics-tutorial
```

Q41) What is Django Admin? How to customize it?

We already explained basics earlier.
Now advanced:

Customization:

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

Q42) What is settings.py ALLOWED_HOSTS?

Controls which domains your site can serve.

Example:

```
ALLOWED_HOSTS = ['127.0.0.1', 'localhost', 'mydomain.com']
```

Used for security.

Q43) What are Static Files in Django? How to configure?

Setup:

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [BASE_DIR / 'static']
```

Used for:

- CSS
- JS
- Images

Q44) What is Media File Handling?

Setup:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

Used for:

- Profile pictures
- Uploaded files

Q45) What is `__str__()` in Model? Why must we always define it?

Answer:

Improves readability in admin and shell.

Example:

```
def __str__(self):
    return self.title
```

Q46) What is superuser? How to create one?

Command:

```
python manage.py createsuperuser
```

Used to access admin panel.

Q47) What are Middleware in Django?

Middleware is a function executed on every request & response.

Examples:

- Authentication
- Session management
- CSRF protection
- Security

Follow-up

47.1) Can we create custom middleware?

Yes.

Q48) What is the purpose of `apps.py`?

Holds app configuration.

Q49) What is Django Project Structure? Files meaning.

- manage.py → command manager
- project folder → main configuration
- settings.py → configurations
- urls.py → routing
- wsgi.py/asgi.py → deployment
- app folders → features

Q50) What is Django's Philosophy?

- DRY: Don't Repeat Yourself
- Rapid Development
- Security First
- Clean Design
- Batteries Included

What is a Template in Django?

✅ Answer (Full Explanation)

A **Template** in Django is a text file (usually HTML) that defines the **structure and layout of the UI** for your web pages. It is a blueprint used to render dynamic data sent by Django Views. Templates allow separation of **presentation (UI)** and **business logic**, making your project modular, maintainable, and readable.

Key Points:

- Templates are usually stored in the `templates/` folder inside an app or project.
- They use the **Django Template Language (DTL)** to dynamically render data.
- They support:
  - Variables: `{{ variable_name }}`
  - Tags: `{% if %} ... {% endif %}`, `{% for %} ... {% endfor %}`
  - Filters: `{{ value|filter_name }}`
  - Template inheritance

Real-World Example:
Imagine you are building a **Blog App**:

- `home.html` shows the latest posts.
- `post_detail.html` shows details of a selected post.
  Instead of writing repeated HTML (header, footer, navbar), templates allow you to **reuse common layout**, and only dynamically change the content.

✅ Follow-up Questions

1. Where are templates stored in a Django project?
   - Answer: Either **app-level templates** (`app_name/templates/app_name/`) or **project-level templates** (`project/templates/`).
2. What is the difference between **template variable** and **context variable**?
   - Template variables (`{{ var }}`) are placeholders in the HTML.
   - Context variables are Python data passed from the view to template.
3. Can templates contain Python code directly?
   - ❌ No. Templates are intentionally designed**not to execute Python code**, ensuring separation of logic & presentation.

✅ Cross-Questions

1. Why does Django restrict Python code in templates?

- Security, readability, maintainability, and MVC/MVT separation.

2. Can you include JavaScript or CSS dynamically in templates?
   - Yes, using static files and `{% static %}` template tag.

Intermediate Level

- Templates can extend **base templates** to avoid repetitive HTML (Template Inheritance).
- Templates can load **static files** (CSS/JS/Images) using `{% load static %}`.

Advanced Level

- Templates can include custom **template tags** and **filters** to handle complex logic.
- Example: A filter to format date `{{ post.published_at|date:"F j, Y" }}`

Q2) What is Template Inheritance in Django?

Answer (Full Explanation)

Template Inheritance allows you to create a **base template** with common structure (header, navbar, footer) and then extend it in child templates to override specific sections.

Benefits:

- Reduces repeated HTML code
- Centralized control over UI
- Modular & maintainable templates

How it works:

1. Create a **base template**: `base.html`

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}My Site{% endblock %}</title>
    <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>
<body>
    <header>{% block header %}Header{% endblock %}</header>
    <main>{% block content %}{% endblock %}</main>
    <footer>{% block footer %}Footer{% endblock %}</footer>
</body>
</html>
```

2. Create a **child template**: `home.html`

```
{% extends "base.html" %}

{% block title %}Home Page{% endblock %}
{% block content %}
    <h1>Welcome to My Blog</h1>
{% endblock %}
```

Real-World Example:

- An e-commerce website: `base.html` contains the **header, navbar, footer, and common scripts**.
- Each app (products, cart, checkout) extends `base.html` and only overrides **specific content**.

❓ Follow-Up Questions

1. What are `{% block %}` and `{% extends %}`?
    - `{% block %}` defines replaceable sections in the base template.
    - `{% extends %}` tells Django to use another template as the base.
2. Can a template extend multiple templates?
    - ❌ No. Django allows extending**only one base template**.
3. What happens if you forget `{% block %}` in base template?
    - Child template cannot override missing sections.

❓ Cross-Questions

1. How is template inheritance better than copying HTML everywhere?
    - DRY principle → maintainability → less error-prone.
2. How can you include another template inside a template without inheritance?
    - Use `{% include 'template_name.html' %}`

❓ Intermediate Level

- Advanced inheritance: You can **nest multiple levels**.
- Example: `base.html` → `dashboard_base.html` → `user_profile.html`

Advanced Level

- Dynamic template inheritance using variables:

```
{% extends base_template %}
```

- Useful when rendering different layouts based on **user roles** (Admin vs Customer).

Q3) How to pass context from View to Template?

Answer

- **Context** is a Python dictionary sent from the **View** to the template.

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

def home(request):
    context = {
        'user': 'Eswar',
        'notifications': 5
    }
    return render(request, 'home.html', context)
```

- **Template Usage:**

```
<h1>Welcome {{ user }}</h1>
<p>You have {{ notifications }} notifications</p>
```

Real-World Example:

- In a **dashboard app**, you might pass:
  - `user` → username
  - `tasks` → list of pending tasks
  - `cart_items` → number of items in shopping cart

Follow-Up Questions

1. What if context key does not exist?
   - Template renders **empty string** (no error).
2. Can context contain objects, lists, or querysets?
   - Yes, Django templates can iterate over querysets and objects.
3. How to pass **dynamic data** to multiple templates without repeating code?
   - Use **context processors**.

❓ Cross-Questions

1. Difference between **context** and **session**?
    - Context → temporary, per request
    - Session → persistent, across requests
2. Difference between `render()` and `HttpResponse()`?
    - `render()` → combines template + context
    - `HttpResponse()` → raw response (text/HTML)

Q4) How to load static files in templates?

❓ Answer

1. Load static files:

```
{% load static %}
<link rel="stylesheet" href="{% static 'css/style.css' %}">
<script src="{% static 'js/main.js' %}"></script>
<img src="{% static 'images/logo.png' %}" alt="Logo">
```

2. Django Settings:

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

Real-World Example:

- Your website's CSS, JS, and images are stored in `/static/` and accessed via `{% static %}` tag.
- If user requests `/home/`, template renders HTML + CSS + JS properly.

❓ Follow-Up Questions

1. What is the difference between **STATICFILES_DIRS** and **STATIC_ROOT**?
    - `STATICFILES_DIRS` → development directories
    - `STATIC_ROOT` → directory for `collectstatic` (production)
2. Can media files be loaded like static files?
    - □ No, media files need`MEDIA_URL` and `MEDIA_ROOT`.
3. How to avoid broken static files in production?
    - Run `python manage.py collectstatic` and configure server.

❓ Cross-Questions

1. Difference between static files and media files?
2. How does Django handle caching of static files?

What are Template Tags in Django?

✅ Answer (Full Explanation)

Template tags are **special constructs** in Django templates enclosed within `{% %}`. They allow you to **control logic, loops, conditionals, and include other templates** dynamically in HTML.

Two types:

1. **Basic Tags** — built-in simple logic
    - `{% if %}`, `{% for %}`, `{% comment %}`, `{% include %}`
2. **Custom Tags** — created by developers for **complex functionality**

Real-World Example:

- In a **Blog website**, you can display posts dynamically:

```
{% for post in posts %}
  <h2>{{ post.title }}</h2>
  <p>{{ post.content }}</p>
{% empty %}
  <p>No posts found!</p>
{% endfor %}
```

- `{% empty %}` ensures **graceful fallback** if `posts` is empty.

❓ Follow-Up Questions

1. Difference between `{% include %}` and `{% extends %}`?
    - `include` → embed another template anywhere
    - `extends` → inherit a base template structure
2. Can we use loops inside loops?
    - ✅ Yes,**nested `{% for %}` loops** are allowed
3. What is `{% csrf_token %}`?
    - Adds CSRF protection in forms to prevent cross-site request forgery attacks.

❓ Cross-Questions

1. Why Django restricts logic in templates?
    - Security, separation of concerns (MVC/MVT), maintainability.
2. How to debug template variables if data is missing?
    - Use `{% debug %}` tag (prints context variables)

❓ Intermediate Level

- Advanced loops: `forloop.counter`, `forloop.first`, `forloop.last`

```
{% for item in items %}
  {{ forloop.counter }}. {{ item.name }}
{% endfor %}
```

❓ Advanced Level

- Custom template tags for **complex operations** (e.g., dynamic menu rendering)

```
# templatetags/custom_tags.py
from django import template
register = template.Library()

@register.simple_tag
def multiply(a, b):
    return a * b
```

- Usage:

```
{% load custom_tags %}
<p>Result: {% multiply 5 6 %}</p>
```

Q2) What are Template Filters in Django?

❓ Answer (Full Explanation)

Template filters are used to **modify variables** before rendering them in templates. Filters are applied using the `|` symbol.

Syntax:

```
{{ variable|filter_name }}
```

Common Built-in Filters:

- `{{ name|upper }}` → converts text to uppercase
- `{{ name|lower }}` → converts text to lowercase
- `{{ date|date:"F j, Y" }}` → formats date
- `{{ value|length }}` → returns length of a list/string
- `{{ text|truncatechars:50 }}` → truncate to 50 characters

Real-World Example:

```
<p>{{ post.content|truncatechars:100 }}</p>
```

- Displays only **first 100 characters** of a blog post.

❓ Follow-Up Questions

1. Can we chain multiple filters?
    - ✅ Yes, `{{ name|upper|truncatechars:10 }}`
2. How to create a **custom filter**?

```
# templatetags/custom_filters.py
from django import template
register = template.Library()

@register.filter
def add_prefix(value, prefix):
    return f"{prefix}-{value}"
```

- Usage:

```
{% load custom_filters %}
<p>{{ username|add_prefix:"user" }}</p>
```

3. Can filters accept arguments?
    - ✅ Some filters accept arguments, e.g., `truncatechars:50`

❓ Cross-Questions

1. Difference between template tags and filters?
    - Tags → control flow & logic
    - Filters → modify variables/data
2. How to debug a filter output?
    - Print value in template or use `{% debug %}`

⍰ Intermediate Level

- Use filters in **loops & conditionals**:

```
{% for post in posts %}
  {{ post.title|upper }}
{% endfor %}
```

⍰ Advanced Level

- Advanced custom filters can interact with **database, settings, or complex logic**.
- Example: Filter to **mask email addresses** in templates

```
@register.filter
def mask_email(email):
    parts = email.split("@")
    return parts[0][:2] + "***@" + parts[1]
```

- Usage:

```
<p>{{ user.email|mask_email }}</p>
```

Q3) Real-Time Template Practice

Scenario: Build a small Blog UI

Step 1 — base.html

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}My Blog{% endblock %}</title>
    <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>
<body>
    <header>
        <h1>My Blog</h1>
        {% block navbar %}{% endblock %}
    </header>
    <main>
        {% block content %}{% endblock %}
    </main>
    <footer>
        {% block footer %}Copyright 2025{% endblock %}
    </footer>
</body>
</html>
```

Step 2 — home.html

```
{% extends "base.html" %}
{% block title %}Home{% endblock %}
{% block content %}
<h2>Latest Posts</h2>
{% for post in posts %}
  <div>
    <h3>{{ post.title|upper }}</h3>
    <p>{{ post.content|truncatechars:100 }}</p>
  </div>
{% empty %}
  <p>No posts available!</p>
{% endfor %}
{% endblock %}
```

Step 3 — View

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

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

Step 4 — Real-time Practice Tips

- Add **static CSS & JS** for styling
- Create **another child template** for post details
- Use **filters for date formatting & truncation**
- Use `{% include %}` for **reusable components** (like cards or post summary)

❓ Follow-Up Questions

1. How do you handle missing static files in templates?
2. Can `{% include %}` templates have their own blocks?
    - ❌ Blocks in included templates cannot be overridden

❓ Cross-Questions

1. How to render **dynamic navigation menu** based on user role?
    - Use **template tags** + **context variables**
2. Difference between `{% block %}` and `{% include %}`?
    - Block → overridable section in base template
    - Include → static inclusion of reusable template

What are Django Forms?

✅ Answer (Full Explanation)

Django **forms** are **Python classes** that allow you to handle **HTML forms** easily. They take care of:

- Rendering HTML form elements
- Validating user input
- Converting form data to Python types
- Handling errors gracefully

Two main types of forms:

1. **Django Forms (`forms.Form`)** — manually define fields

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=50)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)
```

2. **Model Forms (`forms.ModelForm`)** — automatically map to Django models

```
from django.forms import ModelForm
from .models import Post

class PostForm(ModelForm):
    class Meta:
        model = Post
        fields = ['title', 'content']
```

Real-World Example:

- Contact forms, signup forms, login forms, feedback forms.

❓ Follow-Up Questions

1. Difference between `forms.Form` and `forms.ModelForm`?
    - `Form` → independent, no model required
    - `ModelForm` → tightly coupled with a model, automatically handles CRUD
2. How do you handle form validation?
    - Use `is_valid()` method. Errors are automatically available via `form.errors`

❓ Cross-Questions

1. How do you add **custom validation**?

```
def clean_name(self):
    name = self.cleaned_data['name']
    if not name.isalpha():
        raise forms.ValidationError("Name should contain only letters")
    return name
```

2. Difference between `cleaned_data` and `initial`?
    - `cleaned_data` → validated user input
    - `initial` → default form values

Q2) How Django Handles GET and POST Requests in Forms?

✅ Answer (Full Explanation)

- **GET** request:
    - Sends data via **URL parameters** (`?key=value`)
    - Used for search forms or fetching data
    - Data is visible in URL → not secure for passwords
- **POST** request:
    - Sends data **inside HTTP body**
    - Used for submitting sensitive data (signup, login)
    - More secure for password/hidden fields

Example View Handling Both:

```
from django.shortcuts import render
from .forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Process form data
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']
            return render(request, 'success.html', {'name': name})
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})
```

❓ Follow-Up Questions

1. How do you retrieve GET data in Django?
    - `request.GET.get('param_name')`
2. How do you retrieve POST data in Django?
    - `request.POST.get('param_name')`
    - Or use `form.cleaned_data` if using Django Forms

❓ Cross-Questions

1. Can you use GET request to submit sensitive data?
    - ❌ Not recommended— visible in URL, insecure
2. Difference between `request.POST` and `form.cleaned_data`?
    - `request.POST` → raw input, not validated
    - `form.cleaned_data` → validated, cleaned Python types

Q3) What is CSRF Protection in Django?

❓ Answer (Full Explanation)

CSRF (Cross-Site Request Forgery):

- Attack where a **malicious website tricks user's browser** into submitting unwanted requests.
- Django protects forms automatically by adding a **CSRF token**.

How it works:

- Include `{% csrf_token %}` in every POST form

```
<form method="POST">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Submit</button>
</form>
```

- Token is validated server-side on form submission

Real-World Example:

- Any login, signup, or payment form uses CSRF protection to prevent attacks

---

❓ Follow-Up Questions

1. What happens if CSRF token is missing?
    - ❌ Django raises `403 Forbidden` error
2. How to disable CSRF for testing?

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

@csrf_exempt
def test_view(request):
    return HttpResponse("CSRF disabled")
```

- ❌ Not recommended in production

❓ Cross-Questions

1. Difference between CSRF and XSS?
    - CSRF → tricks user into submitting requests
    - XSS → injects malicious scripts into page
2. Can AJAX requests be CSRF-protected?
    - ✅ Yes, send CSRF token in headers:

```
xhr.setRequestHeader("X-CSRFToken", csrftoken);
```

Q4) Real-Time Form Practice

Scenario: Build **Contact Form UI**

Step 1 — forms.py

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=50)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)
```

Step 2 — views.py

```
from django.shortcuts import render
from .forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
```

```
            return render(request, 'success.html', {'name':
form.cleaned_data['name']})
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})
```

Step 3 — contact.html

```
{% load static %}
<html>
<head>
    <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>
<body>
    <h1>Contact Us</h1>
    <form method="POST">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

Step 4 — success.html

```
<h2>Thank you, {{ name }}! Your message has been received.</h2>
```

Step 5 — Practice Tips

- Test **GET vs POST** by printing `request.GET` and `request.POST`
- Add **custom validations** for email or phone number
- Style form using **Bootstrap**
- Add **success messages** or redirect using `HttpResponseRedirect`

❓ Follow-Up Questions

1. How do you preserve form data after invalid submission?
    - Form automatically repopulates fields when passed back:
2. `return render(request, 'contact.html', {'form': form})`
3. Difference between `HttpResponse` and `render` in forms?
    - `HttpResponse` → returns raw response
    - `render` → renders template with context

❓ Cross-Questions

1. How to handle multiple forms in a single template?

    - Give **unique names**, create multiple form instances, validate individually
2. How to handle file uploads with forms?
    - Use `forms.FileField` and add `enctype="multipart/form-data"` in form tag

How to style Django forms in templates?

✅ Answer (Full Explanation)

By default, Django renders forms with basic HTML. For professional UIs:

1. **Add CSS classes to widgets**

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=50,
widget=forms.TextInput(attrs={'class':'form-control'}))
    email = forms.EmailField(widget=forms.EmailInput(attrs={'class':'form-
control'}))
    message = forms.CharField(widget=forms.Textarea(attrs={'class':'form-
control'}))
```

2. **Use `as_p`, `as_table`, `as_ul`** in template

```
{{ form.as_p }} <!-- default rendering -->
```

3. **Integrate Bootstrap**

```
<form method="POST" class="p-4 bg-light">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Real-World Example:

- Bootstrap-styled login forms, admin dashboards, feedback forms

---

✅ Follow-Up Questions

1. Difference between styling in **forms.py** vs **templates**?
    - `forms.py` → reusable CSS class for widget
    - `template` → per-page styling
2. How to override **form label style**?

```
forms.CharField(label='Full Name',
widget=forms.TextInput(attrs={'class':'form-control'}))
```

❓ Cross-Questions

1. How to render **field by field** instead of `{{ form.as_p }}`?

```
<label>Name:</label>{{ form.name }}
<label>Email:</label>{{ form.email }}
```

2. How to add **placeholder text**?

```
widget=forms.TextInput(attrs={'placeholder':'Enter your name'})
```

Q2) Custom Validation in Django Forms

✅ Answer (Full Explanation)

- Django provides **built-in validators** (e.g., `EmailField`, `MaxLengthValidator`)
- **Custom validation** is done using `clean_<field>()` or `clean()` methods

Field-level validation:

```
def clean_email(self):
    email = self.cleaned_data['email']
    if not email.endswith('@gmail.com'):
        raise forms.ValidationError('Only Gmail addresses allowed')
    return email
```

Form-level validation (cross-field):

```
def clean(self):
    cleaned_data = super().clean()
    password = cleaned_data.get('password')
    confirm = cleaned_data.get('confirm_password')
    if password != confirm:
        raise forms.ValidationError("Passwords must match")
```

Real-World Example:

- Password confirmation during signup
- Age validation in registration forms

❓ Follow-Up Questions

1. What is the difference between `ValidationError` and `forms.errors`?
    - `ValidationError` → raised in Python code

- `form.errors` → collected automatically and displayed in template

2. Can you use **validators from Django validators module**?

```
from django.core.validators import MaxValueValidator, MinLengthValidator
age = forms.IntegerField(validators=[MaxValueValidator(60)])
```

❓ Cross-Questions

1. How to validate **multiple fields together**?
   - Use `clean()` → form-level validation
2. How to display **custom error messages in templates**?

```
{% for field in form %}
    {{ field.label_tag }} {{ field }}
    {% if field.errors %}
        <div class="text-danger">{{ field.errors }}</div>
    {% endif %}
{% endfor %}
```

Q3) Handling Form Errors in Django

✅ Answer (Full Explanation)

1. **Automatic error handling:** Django forms automatically populate `form.errors`

```
{{ form.non_field_errors }} <!-- Form-level errors -->
{{ form.field_name.errors }} <!-- Field-level errors -->
```

2. **Real-Time Error Display:** Use Bootstrap alerts or inline error messages

```
<div class="alert alert-danger">
    {{ form.non_field_errors }}
</div>
```

3. **Preserving User Input:** When a form is invalid, Django re-populates the fields automatically

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

Real-World Example:

- Signup form with password mismatch errors
- Login form with invalid credentials

❓ Follow-Up Questions

1. How to customize **non-field errors**?

```
raise forms.ValidationError("Custom form-level error")
```

2. How to show **first error only** for UX improvement?

```
{{ form.errors.as_text|slice:":1" }}
```

❓ Cross-Questions

1. Can we log form errors for debugging?

```
if not form.is_valid():
    print(form.errors)
```

2. How to **highlight error fields** in template?

```
{% if field.errors %} <div class="border border-danger">{{ field }}</div> {%
endif %}
```

Q4) Real-Time UI Enhancements

Techniques to improve form UX:

1. **Placeholders & tooltips**

```
widget=forms.TextInput(attrs={'placeholder':'Enter email'})
```

2. **Input masks** (phone numbers)

- Use JS libraries like **Inputmask.js**

3. **Dynamic form validation with JS**

- Validate email format or password strength on the client-side

4. **Bootstrap Feedback**

```
<input type="text" class="form-control is-invalid">
<div class="invalid-feedback">Please enter a valid email</div>
```

5. **AJAX Form Submission**

- Submit form without page reload

```
$.ajax({
    url: '/contact/',
    type: 'POST',
    data: $('#contact-form').serialize(),
    success: function(response){ alert('Form submitted!'); }
});
```

❓ Follow-Up Questions

1. Difference between **server-side validation** and **client-side validation**?
    - Server-side → secure, mandatory
    - Client-side → improves UX, optional
2. How to make **forms responsive**?
    - Use Bootstrap classes like `col-md-6`, `form-control`, `form-row`

❓ Cross-Questions

1. Can AJAX submission still be CSRF-protected?
    - ✅ Yes, include token in header
2. How to **reset form after successful submission**?

```
$('#contact-form')[0].reset();
```

Q5) Day 2 — Full Interview Q&A Coverage

By the end of **DAY-2**, you should be ready for:

Foundational Questions

- Difference between Template and Form
- What is CSRF? Why is it needed?
- GET vs POST requests

Intermediate Questions

- Custom field validation & cross-field validation
- Styling forms using `attrs` and Bootstrap
- Handling form errors gracefully

Advanced Questions

- AJAX form submission & CSRF integration
- Client-side vs server-side validation

- Real-time UI enhancements (placeholders, input masks, inline feedback)
- Multiple forms in a single template
- Preserving form data after errors

Real-World Examples Covered:

- Contact Us Form
- Signup/Login Forms with validation & error handling
- Payment forms with CSRF and AJAX submission

Q1) What is a Model in Django?

✅ Answer (Full explanation)

A **Model** is a Python class that *declares the shape and behaviour of data* your app will store — effectively a blueprint for a database table. Each model attribute maps to a database column, and each model instance maps to a row. Django's ORM (Object-Relational Mapper) translates model operations into SQL so you can work with Python objects instead of SQL queries.

Key responsibilities

- Define fields and their types (CharField, IntegerField, DateTimeField, etc.)
- Declare relationships (ForeignKey, OneToOneField, ManyToManyField)
- Provide model methods for business logic (e.g., `get_full_name()`)
- Optionally control metadata via inner `Meta` class (ordering, table name, unique constraints)

🔹 Real-world example

Notes app:

```
from django.db import models
from django.contrib.auth import get_user_model

User = get_user_model()

class Note(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='notes')
    title = models.CharField(max_length=200)
    content = models.TextField()
    is_pinned = models.BooleanField(default=False)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"{self.title} ({self.user.username})"

    class Meta:
        ordering = ['-created_at']
```

❓ Follow-up questions (with answers)

- **Q:** What does `on_delete=models.CASCADE` do?
  A: When the related `User` is deleted, all their `Note` objects are also deleted.
- **Q:** Why use `get_user_model()` instead of `auth.User`?
  A: It supports custom user models and is recommended for reusable apps.
- **Q:** What is `related_name` for?
  A: It gives a reverse-access attribute (`user.notes`) to query related objects.

❓ Cross-questions

- How would you add a slug for SEO-friendly URLs? (Add `slug = models.SlugField(unique=True)` and populate it in `save()` or via signals.)
- When should you use `TextField` vs `CharField`? (`TextField` for large free text; `CharField` for short, indexed strings.)

Level notes

- **Beginner:** Models map to DB tables, fields to columns.
- **Intermediate:** Use `Meta` for ordering, unique constraints, indexes.
- **Advanced:** Add custom managers, querysets, model signals, and database indexes for performance.

Q2) How do you create a model file and what are best practices?

❓ Answer

Create models in `models.py` of your app. Keep models focused: each model represents a single concept (e.g., `Note`, `Tag`, `Category`). Use small helper methods on models for encapsulated behavior. For complex projects, split models by module (e.g., `models/__init__.py`, `models/note.py`) and import in `models/__init__`.

Best practices

- Keep model concerns focused (single responsibility).
- Use `choices` for constrained values.
- Add `__str__()` for readable admin entries.
- Add `get_absolute_url()` for canonical links.
- Add indexes with `Meta.indexes` for frequently queried fields.
- Avoid importing heavy modules at top-level if not needed.

❓ Real-world example

Add a `Tag` model and ManyToMany:

```
class Tag(models.Model):
    name = models.CharField(max_length=30, unique=True)

class Note(models.Model):
    # ...fields...
    tags = models.ManyToManyField(Tag, blank=True, related_name='notes')
```

❓ Follow-ups

- **Q:** When and why split models into multiple files?
  A: When `models.py` becomes large; for readability and maintainability.
- **Q:** How to add database-level unique constraints spanning fields?
  A: Use `unique_together` (deprecated) or `constraints = [models.UniqueConstraint(...)]` in `Meta`.

Q3) What are migrations in Django?

✅ Answer (Full explanation)

Migrations are **versioned changes** to the database schema derived from model changes. They act like "diffs" for your database: Django creates migration files describing what SQL to run, and `migrate` applies them. Migrations let you evolve the schema safely across environments (dev, staging, production).

Migration workflow

1. Change models (add/modify/delete fields/models).
2. Run `python manage.py makemigrations` → creates migration files under `app/migrations/`.
3. Run `python manage.py migrate` → executes SQL against DB to bring schema in sync.

Migrations also support data migrations (Python code that mutates data during a schema change).

❓ Follow-up questions

- **Q:** What is the difference between `makemigrations` and `migrate`?
  A: `makemigrations` creates the migration *files* based on model changes. `migrate` applies those migration files to the database.
- **Q:** What is an initial migration?
  A: Typically `0001_initial.py`, it creates the initial table schema for new models.
- **Q:** What are data migrations?
  A: Migrations that include Python code (in `operations = [migrations.RunPython(...)]`) to transform or populate data during schema changes.

▯ Common commands

- `python manage.py makemigrations appname`
- `python manage.py migrate`
- `python manage.py showmigrations` — shows applied/pending migrations
- `python manage.py sqlmigrate appname 0001` — shows SQL for a migration

Cross-questions

- How to revert a migration? (`python manage.py migrate appname <previous_migration>`)
- What is `--fake` migration? (Marks migration as applied without running SQL). Use with care for manual DB changes.

Level notes

- **Beginner:** Run makemigrations/migrate after model changes.
- **Intermediate:** Examine `migrations/` files; write small data migrations.
- **Advanced:** Manage complex schema changes, squashed migrations, multi-DB migrations, and branching histories.

Q4) Build a basic Note model (step-by-step) — practical

▯ Steps & code

1. In `notes/models.py`:

```
from django.db import models
from django.contrib.auth import get_user_model

User = get_user_model()

class Note(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='notes')
    title = models.CharField(max_length=200)
    content = models.TextField()
    is_archived = models.BooleanField(default=False)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title
```

2. Register in `notes/admin.py`:

```
from django.contrib import admin
```

```
from .models import Note

@admin.register(Note)
class NoteAdmin(admin.ModelAdmin):
    list_display = ('title','user','created_at','is_archived')
    list_filter = ('is_archived',)
    search_fields = ('title','content')
```

3. Create migrations & apply:

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

4. Shell quick test:

```
python manage.py shell
>>> from django.contrib.auth import get_user_model
>>> User = get_user_model()
>>> user = User.objects.first()
>>> from notes.models import Note
>>> Note.objects.create(user=user, title="Todo", content="Refactor models")
```

❓ Follow-ups

- **Q:** How to add full-text search for `content`?
  A: Use PostgreSQL full-text search (`SearchVector`), or integrate third-party systems (Elasticsearch).
- **Q:** How to paginate notes in views?
  A: Use Django's `Paginator` class or class-based `ListView` with `paginate_by`.

Q5) What are ForeignKey, OneToOneField, ManyToManyField? When to use each?

❓ Answer

- **ForeignKey** — one-to-many; use when many child rows relate to one parent (e.g., many `Note` objects belong to one `User`).
- **OneToOneField** — one-to-one; use for profile tables: each `User` has one `UserProfile`.
- **ManyToManyField** — many-to-many; use for tags, categories where multiple notes can have multiple tags.

Example

```
class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(blank=True)

class Tag(models.Model):
```

```
    name = models.CharField(max_length=30, unique=True)

class Note(models.Model):
    tags = models.ManyToManyField(Tag, blank=True)
```

Follow-ups

- **Q:** How are ManyToMany relations stored?
  A: In an auto-generated join table (unless you provide a `through` model).
- **Q:** How to query ManyToMany efficiently?
  A: Use `prefetch_related('tags')` to avoid N+1 queries.

Q6) What are model managers & custom QuerySets?

Answer

The default manager `objects` returns a `QuerySet`. Customize behavior by adding custom managers or QuerySet methods for reusable query logic.

Example

```
class NoteQuerySet(models.QuerySet):
    def pinned(self):
        return self.filter(is_pinned=True)

class Note(models.Model):
    # fields...
    objects = NoteQuerySet.as_manager()
```

Now `Note.objects.pinned()` returns pinned notes.

Follow-ups

- **Q:** When to use managers vs querysets?
  A: Use QuerySet for chainable filters; use Manager for custom entry points or altering default queryset.

Q7) Migrations gotchas & advanced topics

Common issues & solutions

- **Conflicting migrations:** Happens when two branches create migrations; resolve by creating a merge migration or manually editing.

- **Renaming fields:** Use `RenameField` operation in migration or let Django detect rename with `preserve_default` flags; otherwise data loss risk.
- **Removing a field with data:** Consider data migration to move or archive data before dropping a column.
- **Squashing migrations:** `python manage.py squashmigrations appname start end` — combines many migrations into one for performance; test carefully.
- **Fake migrations:** `python manage.py migrate --fake appname zero` — mark migrations applied/unapplied without running SQL (used cautiously when DB schema was changed manually).

❓ Follow-ups

- **Q:** How to generate a data migration?
  A: `python manage.py makemigrations --empty appname` then edit migration with `migrations.RunPython(my_data_migration_fn)`.
- **Q:** How to handle large tables during schema change?
  A: Use non-blocking operations where DB supports them (Postgres concurrent indexes), run schema changes in maintenance windows, or use background jobs to migrate data gradually.

Q8) What is Jinja and how does it relate to Django templates? (Jinja tags)

✅ Answer (Full explanation)

Jinja2 is an alternative templating engine to Django's built-in Template Language (DTL). Jinja is used by Flask by default and is known for speed and a syntax similar to DTL. Django supports Jinja2 as an optional template backend if you want Jinja features (e.g., richer sandbox, faster rendering).

Key differences

- **Syntax**: Similar tags (`{{ }}`, `{% %}`) but some built-ins differ.
- **Filters & tests**: Jinja has a rich set and syntax for tests (`is defined`, `is iterable`).
- **Autoescaping**: Both support autoescape but behavior may differ.
- **Template inheritance & macros**: Jinja supports `macro` (like functions) which DTL lacks.

Using Jinja in Django
Add Jinja2 backend in `TEMPLATES` in `settings.py`:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.jinja2.Jinja2',
        'DIRS': [BASE_DIR / 'jinja_templates'],
        'APP_DIRS': False,
```

```
        'OPTIONS': {
            'environment': 'myproject.jinja2.environment',
        },
    },
    # keep Django templates backend too if needed
]
```

Then create `myproject/jinja2.py` to configure environment (filters, globals).

❓ Follow-ups

- **Q:** Should you use Jinja or DTL in Django?
  A: Use DTL for standard Django features (easy integration with `TemplateResponse`, template tags). Use Jinja if you need macros, performance, or prefer its syntax — but know differences (e.g., template tag ecosystem).
- **Q:** Are Django template tags available in Jinja?
  A: No — Django built-in tags (like `{% url %}`) aren't available by default; you must provide equivalents or register functions/filters.

❓ Cross-questions

- How to pass Django context processors to Jinja? (Configure environment and add desired context manually or use helper functions.)
- How to migrate a project from DTL to Jinja? (Audit templates, replace tags/filters, test thoroughly.)

Q9) Interview-style rapid Q&A (concise answers — use to rehearse)

- **Q:** What command creates migration files?
  A: `python manage.py makemigrations`
- **Q:** What command applies migrations?
  A: `python manage.py migrate`
- **Q:** Where migration files are stored?
  A: `your_app/migrations/`
- **Q:** How do you inspect SQL for a migration?
  A: `python manage.py sqlmigrate app_name migration_number`
- **Q:** How to roll back the last migration?
  A: `python manage.py migrate appname <previous_migration>` (or use `zero` to unapply all for app)
- **Q:** What is a data migration?
  A: A migration that runs Python code to transform or populate data, using `migrations.RunPython`.
- **Q:** Why use `auto_now_add` vs `auto_now`?
  A: `auto_now_add` sets timestamp when created; `auto_now` updates timestamp every save.

Q10) Practical mini-tasks (homework / practice)

1. Implement `Note` with `Tag` (ManyToMany) and admin customisation (filters + search).
2. Add a `slug` field to `Note`, auto-fill it on save (use `slugify`) and create migrations.
3. Create a data migration that backfills `slug` for existing notes.
4. Write a custom manager `Note.objects.active()` that filters non-archived notes. Add tests for the manager.
5. Convert a simple DTL template to Jinja and note the differences (e.g., `{% url %}` replacement).

Q1) What is Django Admin and why is it useful?

✅ Answer (full explanation)

Django Admin is a built-in, auto-generated administrative interface that lets developers and site administrators **create, read, update, and delete (CRUD)** model data without writing UI code. It introspects your `models.py` and provides a web UI for managing objects, with paging, filters, search, forms, and permissions.

Why useful

- Rapid data management for development and small-scale operations
- Instant admin UI after registering models — huge time saver
- Built-in user & permission system integrated with Django auth
- Extensible: customize forms, list displays, inlines, actions, and templates

Real-world example
For a Notes app you can register `Note` and `Tag`. The admin provides an easy way for content editors to add, edit, or archive notes without a front-end UI.

❓ Follow-up questions (with answers)

1.1) How do you enable admin for a new project?

- Ensure `django.contrib.admin` is in `INSTALLED_APPS`, run `python manage.py migrate`, create a superuser with `createsuperuser`, then visit `/admin/`.

1.2) Is admin meant for end-users?

- No — it's an internal tool for staff. For production user workflows build custom views/permissions.

1.3) Is admin secure by default?

- Reasonably: it uses auth, CSRF protection, and permission checks. But you must secure the endpoint (restrict access, use HTTPS, consider IP whitelisting).

Cross-questions

- How would you hide admin from public discovery? (Change URL from `/admin/` to something less predictable; use web server restrictions.)
- Why not use admin as the public CMS for high-traffic websites? (Performance, UX limitations, and fine-grained workflow requirements.)

Q2) How to register a model with admin?

Answer (full explanation)

At minimum, import the model and call `admin.site.register(Model)`. Better practice: use a `ModelAdmin` class to customize the admin behavior.

Example (simple):

```
# notes/admin.py
from django.contrib import admin
from .models import Note

admin.site.register(Note)
```

Example (recommended with ModelAdmin):

```
@admin.register(Note)
class NoteAdmin(admin.ModelAdmin):
    list_display = ('title', 'user', 'is_archived', 'created_at')
    search_fields = ('title','content')
```

Real-world note: Prefer `@admin.register()` decorator — clearer and groups registration with configuration.

Follow-up questions

2.1) Where to put admin code?

- `app/admin.py`. For complex apps split into `admin/` package and import in `app/admin.py`.

2.2) What does `list_display` do?

- Controls columns shown on the changelist page (index of objects).

2.3) Can you unregister a model?

- Yes: `admin.site.unregister(MyModel)` — useful to replace default registration.

❓ Cross-questions

- What if two apps have model name conflicts? (Use app label in search/filter or register with distinct admin names.)
- How do you make a model read-only in admin? (Override `has_add_permission`, `has_change_permission`, `has_delete_permission` to return False and/or set fields as `readonly_fields`.)

Q3) Customizing list view: `list_display`, `list_filter`, `search_fields`, `ordering`

❓ Answer (full explanation)

`ModelAdmin` provides options to tune the changelist:

- `list_display` — tuple of fields or callables to show as columns.
- `list_filter` — fields for sidebar filters (booleans, choices, foreign keys, DateField with options).
- `search_fields` — tuple of field names to search (use `'^'`, `'='`, `'@'` prefixes for performance).
- `ordering` — default ordering on the changelist page.

Example:

```
class NoteAdmin(admin.ModelAdmin):
    list_display = ('title','user','is_archived','created_at')
    list_filter = ('is_archived','created_at','user')
    search_fields = ('title','content','user__username')
    ordering = ('-created_at',)
```

Real-world benefit: Admin users can quickly narrow content (e.g., show only archived notes from last week from a specific user).

❓ Follow-up questions

3.1) What prefixes can you use in `search_fields`?

- `'^field'` — startswith (fast if indexed)
- `'=field'` — exact match
- `'@field'` — full text (Postgres specific)
  Default is `icontains` (case-insensitive contains).

3.2) How to add a custom column (callable) in `list_display`?

- Define a method on `ModelAdmin` or on model, add `short_description` to label it, and include it in `list_display`.

❓ Cross-questions

- How do you avoid N+1 queries in admin list view? (Use `list_select_related` for foreign keys and `prefetch_related` by overriding `get_queryset`.)
- How to filter by related model fields in `list_filter`? (Use `list_filter` with related field names or `RelatedOnlyFieldListFilter`.)

Q4) Inline models (editing related objects on same page)

❓ Answer (full explanation)

Inlines let you edit related objects on the parent model's change page. Useful for one-to-many relationships (e.g., Notes with Comments).

Example:

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

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

@admin.register(Note)
class NoteAdmin(admin.ModelAdmin):
    inlines = [CommentInline]
```

Tabular vs Stacked: `TabularInline` shows compact rows; `StackedInline` stacks full forms.

❓ Follow-up questions

4.1) When to use inlines?

- When related objects are small and should be edited while editing parent.

4.2) What about performance?

- Too many inline objects can slow page load; limit with `max_num` or avoid inlines for large child sets.

❓ Cross-questions

- How to validate inline forms? (Use `InlineFormSet` custom validation by overriding `get_formset`.)
- How to prevent adding inlines for non-staff? (Check permissions in `has_add_permission` on the Inline class.)

Q5) Admin actions & bulk operations

❓ Answer (full explanation)

Admin actions are functions applied to selected items on the changelist (e.g., mark selected notes archived). They operate server-side and can be customized.

Example:

```
def make_archived(modeladmin, request, queryset):
    queryset.update(is_archived=True)
make_archived.short_description = 'Mark selected notes as archived'

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

Real-world usage: Bulk publish/unpublish, export selected rows, or trigger background tasks for selected items.

❓ Follow-up questions

5.1) How to add confirmation step to action?

- Actions can return an `HttpResponse` with a custom confirmation template; refer to Django docs for action intermediate confirmations.

5.2) Can actions be asynchronous?

- Yes — call tasks (e.g., Celery) in the action to offload heavy work.

❓ Cross-questions

- What if action needs form input? (Define an action that redirects to an intermediate admin view or use custom admin view to collect parameters.)
- How to restrict actions to certain users? (Check `request.user.has_perm()` inside action and raise `PermissionDenied` if unauthorized.)

Q6) Admin forms: customizing form widgets, validation, readonly fields

✅ Answer (full explanation)

Control form rendering and validation in admin by:

- `form` — supply a custom `ModelForm` with validators and custom widgets
- `readonly_fields` — fields shown but not editable
- `fieldsets` — layout grouping and ordering of fields
- `formfield_overrides` — set widgets for field types globally

Example:

```
from django import forms

class NoteForm(forms.ModelForm):
    class Meta:
        model = Note
        fields = '__all__'
        widgets = {
            'content': forms.Textarea(attrs={'rows':4,'cols':60}),
        }

class NoteAdmin(admin.ModelAdmin):
    form = NoteForm
    readonly_fields = ('created_at',)
    fieldsets = (
        (None, {'fields':('title','user')}),
        ('Details', {'fields':('content','is_archived')}),
    )
```

❓ Follow-up questions

6.1) How to add model validation that shows as form error in admin?

- Implement `clean()` on the `ModelForm` or override model `clean()` and ensure `full_clean()` is called.

6.2) How to add help text to admin fields?

- Use `help_text` in model field definition or set widget `attrs`.

❓ Cross-questions

- How to internationalize admin labels and help texts? (Use `verbose_name`, `verbose_name_plural`, and `ugettext_lazy` for translations.)
- How to add dynamic choices to a field in admin? (Override `formfield_for_choice_field` or set choices in the form's `__init__`.)

Q7) Admin permissions & security

❓ Answer (full explanation)

Permissions are tied to models: `add`, `change`, `delete`, `view`. Admin checks these permissions by default. You can override permission methods (`has_add_permission`, `has_change_permission`, `has_delete_permission`) to implement custom rules.

Best practices

- Only give staff status to trusted users.
- Use per-model permissions and Django groups to group privileges.
- Protect admin endpoint with HTTPS and firewall or VPN when possible.
- Consider 2FA for admin accounts and monitor login attempts.

Example:

```
class NoteAdmin(admin.ModelAdmin):
    def has_delete_permission(self, request, obj=None):
        return request.user.is_superuser
```

❓ Follow-up questions

7.1) How to create a custom permission?

- Define `permissions = [('can_publish','Can publish notes')]` in model `Meta`. Use `user.has_perm('app_label.can_publish')`.

7.2) How to audit admin actions?

- Use `django.contrib.admin.models.LogEntry` or third-party audit apps to log changes.

❓ Cross-questions

- How to prevent cross-site scripting in admin? (Admin auto-escapes output; avoid marking strings safe; sanitize any HTML rendered.)
- How to limit admin by IP? (Use middleware or web server rules to restrict access to admin URL.)

Q8) Admin performance & scaling

✅ Answer (full explanation)

Admin can become slow with large datasets. Techniques to improve:

- Add indexes on frequently filtered/searched fields.
- Use `list_select_related` and `get_queryset` to prefetch related objects.
- Limit `list_display` and use pagination (default 100).
- Use `raw_id_fields` for foreign keys with large related tables to show a lookup widget instead of full dropdown.
- Avoid expensive property methods in `list_display` — precompute or use annotations.

Example optimizations:

```
class NoteAdmin(admin.ModelAdmin):
    list_display = ('title','user','created_at')
    list_select_related = ('user',)
    raw_id_fields = ('user',)
    def get_queryset(self, request):
        qs = super().get_queryset(request)
        return qs.select_related('user').prefetch_related('tags')
```

❓ Follow-up questions

8.1) Why use `raw_id_fields`?

- Prevent huge dropdowns and load only IDs with a search popup — much faster and scalable.

8.2) How to measure admin query performance?

- Use Django debug toolbar in dev or log SQL with `connection.queries` to inspect slow queries.

Cross-questions

- When should you replace admin with a custom dashboard? (When you need custom workflows, complex UIs, heavy data operations, or public-facing content management.)

Q9) CRUD: Python (ORM) vs raw SQL — comparison & examples

Answer (full explanation)

Create / Read / Update / Delete can be done via Django ORM (Pythonic) or raw SQL. ORM is safer, portable, and integrates with Django features (migrations, signals). Raw SQL gives ultimate control and can be faster for complex queries.

Examples

Create

- ORM:

```
note = Note.objects.create(user=user, title='Hi', content='...')
```

- SQL:

```
INSERT INTO notes_note (user_id, title, content, created_at) VALUES
(1,'Hi','...', '2025-11-30');
```

Read

- ORM:

```
notes = Note.objects.filter(user=user, is_archived=False).order_by('-
created_at')
```

- SQL:

```
SELECT * FROM notes_note WHERE user_id=1 AND is_archived=false ORDER BY
created_at DESC;
```

Update

- ORM:

```
Note.objects.filter(pk=1).update(title='New Title')
n = Note.objects.get(pk=1)
n.title = 'New Title'
n.save()
```

- SQL:

```
UPDATE notes_note SET title='New Title' WHERE id=1;
```

Delete

- ORM:

```
Note.objects.filter(pk=1).delete()
```

- SQL:

```
DELETE FROM notes_note WHERE id=1;
```

Pros & cons

- **ORM pros**: Prevents SQL injection, DB independence, readable, integrates with Django caching/migrations/signals.
- **ORM cons**: Sometimes less efficient for complex joins/aggregations; may generate suboptimal SQL if misused.
- **Raw SQL pros**: Full control, can use DB-specific features & optimized queries.
- **Raw SQL cons**: Prone to injection if not careful, less portable, bypasses ORM caching & model methods.

How to run raw SQL safely in Django

```
from django.db import connection
with connection.cursor() as cursor:
    cursor.execute("SELECT * FROM notes_note WHERE id=%s", [note_id])
    row = cursor.fetchone()
```

Use parameterized queries to avoid SQL injection.

❓ Follow-up questions

9.1) When to prefer raw SQL?

- Complex analytics, window functions, CTEs, or DB-specific performance optimizations.

9.2) How to combine ORM and raw SQL?

- Use `extra()` (deprecated) or `RawSQL`, `raw()` QuerySet for SELECTs, or `connection.cursor()` for non-SELECT.

❓ Cross-questions

- How do transactions work in ORM vs raw SQL? (Django's ORM supports transactions via `transaction.atomic()`; raw SQL must respect same transaction scopes using `atomic` or manual connection control.)
- How to test code that uses raw SQL? (Use test DB and assert expected rows; prefer to wrap SQL in helper layers for testability.)

Q10) Interview rapid Q&A (practice flashcards)

- **Q:** How to register `Note` with custom admin list view?
  A: Use `@admin.register(Note)` and `class NoteAdmin(admin.ModelAdmin)` with `list_display`.
- **Q:** How to add a search box for `title` and `content`?
  A: `search_fields = ('title','content')`
- **Q:** How to show related `user.username` in `list_display`?
  A: Add method `def user_name(self,obj): return obj.user.username` and include `user_name` in `list_display` (or use `user__username` in `search_fields`).
- **Q:** How to restrict delete to superusers?
  A: Override `has_delete_permission` in `ModelAdmin` and return `request.user.is_superuser`.
- **Q:** How to optimize admin queries for FK fields?
  A: Use `list_select_related = ('user',)` or override `get_queryset` with `select_related`.

Practical mini-tasks (homework)

1. Register `Note` and `Tag` models; customize `NoteAdmin` with `list_display`, `list_filter`, `search_fields`, and `raw_id_fields` for `user`.

2. Add an admin action “Export selected notes to CSV”. Implement it so it streams CSV as response.
3. Create `CommentInline` for `Note` and limit inline rows with `max_num=5`.
4. Add a custom admin view to show analytics (e.g., notes per day) and link it from admin index.
5.

1 What is a Django Form?

Answer:

A Django Form is a Python class used to **receive, validate, and clean user input** before saving it into the database.
It helps ensure data is **safe, structured, and validated** before reaching views or models.

Real-Time Example:

Imagine a **job application form** where users enter name, email, experience.
Django Forms validate:

- email must be valid
- name cannot be empty
- experience must be a number

Follow-up:

Q: Why not use raw HTML forms instead of Django Forms?
A: HTML forms provide no security/validation. Django Forms provide built-in validation, error display, CSRF protection, and cleaning of malicious data.

Cross-question:

Q: If I submit a form without validation, what risk occurs?
A: Injection attacks, database corruption, XSS, and incorrect data entry.

---

2 Difference between Form and ModelForm?

Answer:

Django Form	ModelForm
Independent from models	Connected directly to a model
Manual field creation	Auto-generates fields
Manual save() logic	form.save() creates/updates DB rows
Good for non-database input	Good for CRUD

Real-Time Example:

Contact Form (Form) → not stored in DB
Employee Form (ModelForm) → stored in Employee model

Follow-up:

Q: When should you choose Form instead of ModelForm?
A: When data is not meant for the database (ex: Feedback form, Search form).

Cross-question:

Q: Can we add custom validation in ModelForm?
A: Yes, using `clean()` or `clean_fieldname()` methods.

□ 3□ What is CSRF? Why does Django use it?

Answer:

CSRF → **Cross-Site Request Forgery**
It protects a website from malicious users who try to submit forms **without permission**.

Django adds a unique CSRF token in every form to ensure requests come from trusted users only.

Real-Time Example:

Without CSRF, an attacker could secretly submit a **bank transfer form** on behalf of a logged-in user.

Follow-up:

Q: What happens if CSRF token is missing?
A: Django throws: **403 CSRF Verification Failed**.

Cross-question:

Q: Do APIs need CSRF?
A: Usually no → APIs use tokens, JWT, OAuth.

□ 4□ How do you create a form in Django (Theory + Code)?

Answer:

A basic Form inherits from `forms.Form`.

Code:

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=50)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)
```

Real-Time Example:

Used for customer support "Contact Us" page.

Follow-up:

Q: How to add custom validation?

```
def clean_name(self):
    name = self.cleaned_data['name']
    if not name.isalpha():
        raise forms.ValidationError("Only alphabets allowed")
    return name
```

5️⃣ How to create a ModelForm (Theory + Code)?

Answer:

ModelForm auto-creates form fields based on a model.

```
from django import forms
from .models import Employee

class EmployeeForm(forms.ModelForm):
    class Meta:
        model = Employee
        fields = ['name', 'role', 'salary']
```

Real-Time Example:

Used in admin dashboards to create employees.

Follow-up:

Q: How to exclude fields?

```
exclude = ['created_at']
```

6️⃣ Explain Create, Edit, Delete Form Operations in Django.

Answer:

CRUD operations use ModelForms to perform database actions.

Create (POST request)

```
def add_employee(request):
    if request.method == 'POST':
        form = EmployeeForm(request.POST)
        if form.is_valid():
            form.save()
    else:
        form = EmployeeForm()
```

Edit (Fetch existing object)

```
def edit_employee(request, id):
    employee = Employee.objects.get(id=id)
    form = EmployeeForm(request.POST or None, instance=employee)
    if form.is_valid():
        form.save()
```

Delete

```
def delete_employee(request, id):
    employee = Employee.objects.get(id=id)
    employee.delete()
```

Follow-up:

Q: What is the purpose of `instance=` in edit forms?
A: It binds the form to an existing database row.

Cross-question:

Q: Can ModelForm update only selected fields?
A: Yes → specify fields in the Meta class.

□ 7□ What is Form Validation? Explain clean() and clean_fieldname().

Answer:

Validation ensures data is correct.

per-field validation

```
def clean_email(self):
    email = self.cleaned_data['email']
    if "gmail.com" not in email:
        raise ValidationError("Must be Gmail")
```

form-level validation

```
def clean(self):
    cleaned = super().clean()
    a = cleaned.get("a")
    b = cleaned.get("b")
    if a == b:
        raise ValidationError("A and B cannot be same")
```

Real-Time Example:

Check passwords match → Signup form.

□ 8□ What are widgets in Django Forms?

Answer:

Widgets convert Python data types → HTML input elements.

Examples:

- `forms.Textarea`
- `forms.PasswordInput`
- `forms.CheckboxInput`

Code:

```
message = forms.CharField(widget=forms.Textarea(attrs={'rows':4}))
```

Follow-up:

Q: Can we customize form field design?
A: Yes, using widget attributes (class, placeholder, id).

□ 9□ Explain form.as_p(), form.as_table(), form.as_ul()

Answer:

These render form fields quickly with minimal HTML.

Method	Output
as_p	`<p>` tags
as_table	`<tr><td>`
as_ul	`<li>` items

Follow-up:

Q: Real-world use?
A: Rare in production → developers prefer custom HTML for styling.

□ □ What is the difference between POST and GET in forms?

Answer:

GET	POST
data visible in URL	data hidden
used for search	used for create/update
not for sensitive data	recommended for forms

Real Example:

Search → GET
Login → POST

What is CRUD in Django? Why is it important?

Answer:

CRUD stands for **Create, Read, Update, Delete** — the 4 fundamental operations used in almost all web applications.
In Django, CRUD is implemented through:

- **Models** → database structure
- **Views** → CRUD logic
- **Templates** → user interface
- **URLs** → routing
- **Forms / ModelForms** → data validation

Without CRUD, a web app cannot manage dynamic data — like notes, tasks, users, blogs, products.

Real-time example:

Your **Notes App** allows:

- Create a note
- List all notes
- View a specific note
- Edit a note
- Delete a note

Follow-up:

Q: Can CRUD exist without using ModelForms?
A: Yes. You can use raw HTML forms, Django Forms, or DRF serializers depending on your use case.

Cross-question:

Q: Which part of CRUD is most error-prone?
A: Update and Delete — because they modify/destroy existing data and require careful validation & permissions.

□ 2□□ Explain the “List View” in Django.

Answer:

A List View displays **multiple rows** of data from a model.
It is used when you want to show:

- Notes list
- Products list
- Employee list
- Blog posts list

In Django, listing is done using:

- `Model.objects.all()`
- Passing the queryset to the template
- Rendering in a loop

Code Example:

```
def notes_list(request):
    notes = Note.objects.filter(user=request.user)
```

```
    return render(request, "notes/note_list.html", {"notes": notes})
```

Real-time example:

WhatsApp chat list → shows all chats for a user.

Follow-up:

Q: Why should we filter by user?
A: To ensure users see **only their own data**, not others'.

Cross-question:

Q: What happens if `.all()` is used instead of filtering by user?
A: Data leakage → security issue.

□ 3□ What is a "Detail View" in Django?

Answer:

Detail View shows **a single instance** of a model based on a unique identifier (like id or slug).

Example:

```
def note_detail(request, id):
    note = Note.objects.get(id=id, user=request.user)
    return render(request, "notes/note_detail.html", {"note": note})
```

Real-time example:

Clicking a note → opens full note content.

Follow-up:

Q: What if ID does not exist?
A: Use `get_object_or_404()` to avoid server errors.

Cross-question:

Q: Can detail view use slug instead of ID?
A: Yes, slugs improve SEO and readable URLs.

□ 4□□ Explain Update/Edit View in Django.

Answer:

Update View modifies existing database entries.
It uses **ModelForm + instance parameter** to pre-fill old data.

Example:

```
def note_edit(request, id):
    note = get_object_or_404(Note, id=id, user=request.user)
    form = NoteForm(request.POST or None, instance=note)

    if form.is_valid():
        form.save()
        return redirect("notes_list")

    return render(request, "notes/note_form.html", {"form": form})
```

Real-time example:

Editing your saved note in Google Keep.

Follow-up:

Q: What does `instance=note` do?
A: It binds the form to an existing object, enabling updates instead of creating a new entry.

Cross-question:

Q: What if we don't pass instance?
A: A new note will be created instead of updating → duplicate records.

□ 5□□ Explain Delete View in Django.

Answer:

Delete View removes a database row.
It should always require confirmation to avoid accidental deletion.

Example:

```
def note_delete(request, id):
    note = get_object_or_404(Note, id=id, user=request.user)
    if request.method == "POST":
        note.delete()
```

```
        return redirect("notes_list")
    return render(request, "notes/note_delete_confirm.html", {"note": note})
```

Real-time example:

Clicking trash icon → asks “Are you sure?” → deletes note.

Follow-up:

Q: Why delete via POST, not GET?
A: GET should not modify data → security best practice.

Cross-question:

Q: What if delete is triggered by GET request?
A: CSRF attacks become possible.

□ 6□ Explain the full CRUD flow for a Notes App.

Beginner Explanation:

- Create → Add new note
- Read → Show list of notes
- Update → Edit existing note
- Delete → Remove note

Intermediate Explanation:

Each CRUD operation uses:

- URL → route
- View → logic
- Template → UI
- Form → validation
- Model → data storage

Advanced Explanation:

A production-ready Notes CRUD must include:

- User authentication → only logged-in users

- CSRF protection
- Authorization → users cannot edit others' notes
- Pagination in List View
- 404 handling for invalid IDs
- Search + filters
- Timestamps → created_at, updated_at
- Soft-delete (optional)
- Transaction safety

---

7 What built-in generic views does Django provide for CRUD?

Answer:

Django provides **Class-Based Views (CBVs)** for CRUD:

Operation	Generic View
List	`ListView`
Detail	`DetailView`
Create	`CreateView`
Update	`UpdateView`
Delete	`DeleteView`

Example:

```
class NoteListView(ListView):
    model = Note
    template_name = "notes/note_list.html"

class NoteCreateView(CreateView):
    model = Note
    form_class = NoteForm
    success_url = "/notes/"
```

Follow-up:

Q: Why use CBVs instead of FBVs?
A: Less code, more structure, reusable.

Cross-question:

Q: Which is easier for beginners?
A: Function-Based Views because flow is easier to understand.

□ 8□ What is `get_object_or_404()` and why is it used?

Answer:

A shortcut that returns object if found, else raises **404 Not Found**.

```
note = get_object_or_404(Note, id=id, user=request.user)
```

Real-time example:

Trying to visit `/note/999999/` for a non-existing note.

Follow-up:

Q: Why not use try–except instead?
A: get_object_or_404() is cleaner, more readable, and handles error pages automatically.

Cross-question:

Q: Does it increase performance?
A: No — just improves code quality and error handling.

□ 9□ How do you implement pagination in List View?

Answer:

Pagination breaks long lists into pages.

Example:

```
from django.core.paginator import Paginator

notes = Note.objects.all()
paginator = Paginator(notes, 5)
page = request.GET.get("page")
```

```
notes_page = paginator.get_page(page)
```

Real-time example:

Gmail shows 50 emails per page.

Follow-up:

Q: What if user enters an invalid page number?
A: Django returns last valid page.

Cross-question:

Q: Which is more efficient — pagination or infinite scroll?
A: Pagination → less server load, better for SEO.

□ □ How to handle security in CRUD?

Answer:

CRUD must be protected from:

- Unauthorized edits
- CSRF
- Direct object access
- Incorrect IDs

Must implement:

✔ `@login_required`
✔ filter by user
✔ POST-only deletion
✔ CSRF tokens

Real-time example:

Prevent user A from editing user B's notes.

Follow-up:

Q: What decorator ensures user authentication?
A: `@login_required`.

Cross-question:

Q: What is “object-level authorization”?
A: Checking that the object belongs to the logged-in user.

⭐ DAY 7 — NOTES APP COMPLETION (FULL INTERVIEW PACK)

Covers:
✔ Finishing UI
✔ Validations (Frontend + Backend)
✔ Deployment preparation
✔ URL + View + Template integration
✔ CRUD polish
✔ Git workflow
✔ Production settings basics

This is a **very important** day because you finish your **first Django end-to-end working project**, expected in every interview.

🔹 1️⃣ What steps are involved in completing a Django project UI?

Beginner Explanation:

UI means the visual part the user interacts with: forms, lists, buttons, pages.

Key UI Tasks:

- Clean base layout
- Template inheritance
- Bootstrap integration
- Responsive forms
- Messages (success/error)
- Navigation bar
- Footer
- Buttons (Add, Edit, Delete)

Intermediate Explanation:

A good UI must be:

- Mobile responsive
- Reusable
- Lightweight
- Template-optimized

Advanced Explanation:

- Use layout blocks (base.html)
- Use custom tags/filters
- Minify CSS/JS for production
- Avoid repeating HTML using components

Real-time Example:

A Notes app similar to *Google Keep Lite*.

Follow-up Q:

What is the importance of base.html?
→ Saves time, improves maintainability.

Cross-question:

How do you avoid code duplication in Django templates?
→ Template inheritance + includes.

---

2 How do you add basic validations in Django?

A) Backend Validations (Django Forms / ModelForms)

Examples:
✔ Required fields
✔ Max/min length
✔ Email validation
✔ Unique constraint
✔ Custom clean() methods

Example (clean method)

```
def clean_title(self):
    title = self.cleaned_data.get("title")
    if len(title) < 3:
        raise forms.ValidationError("Title must be at least 3 characters")
    return title
```

B) Frontend Validations (HTML/JS)

✔ required
✔ minlength
✔ maxlength
✔ pattern

Real-time Example:

A user trying to save an empty note → show error message.

Follow-up Q:

Why do we need backend validation even if frontend exists?
→ Because frontend can be bypassed.

Cross-question:

How do you show form errors in template?

```
{{ form.non_field_errors }}
{{ form.field_name.errors }}
```

□ 3□□ Explain the flow of a CRUD Notes App (Final Polish).

Create → Save → Redirect → Show success message

List → Pagination → Search

Edit → Pre-filled form

Delete → Confirm delete page

Entire Flow:

1. URL hits

2. View picks logic
3. Form validates
4. Model saves into DB
5. Template displays result

Real-time Example:

Google Keep → Add → Save → Displays instantly.

Follow-up Q:

How do you avoid duplicate notes?
→ Add unique constraints or custom clean().

Cross-question:

How do you handle unauthenticated user trying to add a note?
→ LoginRequiredMixin.

□ 4□ How do you prepare a Django project for deployment?

(Critical interview topic)

Beginner:

- Collect static files
- Install dependencies
- Use production server (gunicorn/uvicorn)

Intermediate:

- DEBUG = False
- Allowed hosts
- Whitenoise for static files
- Configure media files
- Environment variables

Advanced:

- Reverse proxy (Nginx)
- SSL (HTTPS)

- Cloud DB (RDS/PostgreSQL)
- Logging & monitoring
- Caching (Redis)
- Load balancer (ALB)

---

5 What changes are required when DEBUG=False?

Explanation:

Django does NOT serve static files in production mode.

Required:

```
DEBUG = False
ALLOWED_HOSTS = ['*']
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

Then run:

```
python manage.py collectstatic
```

Real-time Example:

Deploying on Render/EC2/digitalocean → static CSS not loading → solution: Whitenoise.

Follow-up Q:

What happens if ALLOWED_HOSTS is wrong?
→ "Bad Request 400" error.

Cross-question:

What is the risk of keeping DEBUG=True in production?
→ Sensitive data leakage.

---

6 What is Whitenoise and why do we need it?

Answer:

Whitenoise helps Django serve static files *without* needing Nginx.

Install:

```
pip install whitenoise
```

settings.py:

```
MIDDLEWARE.insert(1, "whitenoise.middleware.WhiteNoiseMiddleware")
STATICFILES_STORAGE =
"whitenoise.storage.CompressedManifestStaticFilesStorage"
```

Follow-up Q:

Why not rely on Django's default static serving?
→ It is only for development.

Cross-question:

Can Whitenoise serve media files?
→ No.

□ 7□ What is a requirements.txt file and how do you generate it?

Explanation:

It contains all installed Python packages for deployment.

Command:

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

Follow-up Q:

Why is this important?
→ Server installs same dependencies.

□ 8□ What is .env and why do we use environment variables?

Explanation:

Sensitive data must not be inside settings.py.

Examples to store in .env:

- DB password
- SECRET_KEY
- API Keys

Tools:

- python-dotenv
- django-environ

Real-time Example:

AWS RDS credentials stored in environment.

Follow-up Q:

Why shouldn't we push .env to GitHub?
→ Security breach.

□ 9□ □ Explain Git workflow when finalizing a project.

Steps:

1. Create branch
2. Commit UI changes
3. Merge to main
4. Tag release version
5. Push to GitHub
6. Deploy with CI/CD

Follow-up Q:

Why use tags?
→ Rollbacks + release tracking.

Cross-question:

What is a pull request?
→ Code review before merging.

□ □ Day-7 Real-Time Deliverables (Final Project State)

Your Notes App must have:

✔ Add Note
✔ Edit Note
✔ Delete Note
✔ List Notes
✔ View Details
✔ Search (optional bonus)
✔ Pagination (optional)
✔ Clean UI (Bootstrap)
✔ Form validations
✔ Django messages
✔ Login protection (if authentication added)
✔ GitHub pushed code
✔ Ready for deployment

This matches real-time interview expectation for **first Django project**.

What is URL routing in Django? Why is it required?

Beginner Explanation

URL routing is the process by which Django maps a browser request
`http://example.com/home/`
to a Python function (view) that should handle that request.

Think of URLs as "address routes" to the correct view.

Intermediate Explanation

Django decouples URLs from code.
This means you can rearrange URLs **without changing your code logic**, improving maintainability.

URLs → Views → Templates is a fundamental flow.

Advanced Explanation

URL routing supports:

- Named routes
- Dynamic routes
- Regex (path converters)
- Namespacing for large applications
- Reverse URL lookups → no hardcoding URLs

Real-Time Example

Amazon product URL:

```
https://amazon.in/product/1234/
```

Django handles this like:

```
path("product/<int:id>/", views.product_detail)
```

Follow-Up Questions

1. What is reverse URL resolution?
2. Why should URLs never be hardcoded?
3. What happens if two URLs match the same pattern?

Cross-Questions

1. If two apps have the same URL name, how do you avoid conflict?
   → *Use namespace.*

□ 2□□ What is include() and why do we use it?

Beginner Explanation

`include()` links another URL file inside project URLs.

Intermediate Explanation

Without include(), a large project's urls.py would become extremely long and unmanageable.

Advanced Explanation

include() helps with:

- App-level routing
- Modular design
- Plug & play apps
- Reusable apps (can be shared across projects)

Real-Time Example

In a real company, different teams maintain different apps:

- payments app
- auth app
- orders app

Each team can maintain its own urls.py.

Follow-Up Questions

1. Can include() accept parameters?
2. Can you include URL patterns dynamically?

Cross-Questions

1. What if include() is not used and everything is written in project urls.py?
   → Hard to scale, merge conflicts, difficult maintenance.

---

3 What is app_name and why is it important?

Beginner Explanation

`app_name` is used inside `urls.py` of an app to define its namespace.

Intermediate Explanation

It prevents URL name conflicts between multiple apps.

Example:
Both *blog* app and *shop* app may have:

```
name="detail"
```

Without namespaces → conflict.

Advanced Explanation

Namespaces allow:

- Reverse lookups like `"blog:detail"`
- Multiple instances of the same app
- Multi-tenant deployments
- API versioning (v1, v2)

Real-Time Example

A company has two apps:

- users
- adminpanel

Both have a URL named "dashboard".
Using app_name, you use:

```
users:dashboard
adminpanel:dashboard
```

Follow-Up Questions

1. What error do you get if URL names clash?
   → NoReverseMatch.

Cross-Questions

1. Is app_name required?
   → No, but heavily recommended for all real-time projects.

□ 4□ How do you organize URLs in large Django projects?

Beginner Explanation

Split URLs by app.

Intermediate Explanation

Use the following structure:

```
project/
 |-- project/urls.py
 |-- accounts/urls.py
 |-- dashboard/urls.py
 |-- notes/urls.py
 |-- api/v1/urls.py
```

Advanced Explanation

Enterprise projects use:
✔ Nested includes
✔ Versioned APIs
✔ Admin panels
✔ Feature-based folders
✔ Router-based URLs (DRF)

Real-Time Example

A fintech company may organize URLs like:

```
/auth/login/
/auth/logout/
/payments/initiate/
/payments/status/
/transactions/history/
/admin/overview/
```

Follow-up Questions

1. How do you structure URLs for microservices?
2. How do you add versioning in URLs?

Cross-Questions

1. What is the best practice when URL patterns become too long?
   → Split into multiple files or modules.

□ 5□ What is the difference between path(), re_path(), and include()?

Beginner Explanation

- **path()** → simple URL patterns
- **re_path()** → regex-based URLs
- **include()** → nested URLs

Intermediate Explanation

Use `re_path` for advanced patterns like slugs, UUIDs, date-based routing.

Advanced Explanation

In REST APIs, you often structure URLs with:

```
re_path(r"^api/v1/products/(?P<id>\d+)/$", view)
```

Real-Time Example

A blog app supporting:

- year
- month
- slug
  can use regex.

Follow-up Questions

1. When should you avoid regex routing?
   → When path converters are enough.

Cross-Questions

1. Which is faster: path() or re_path()?
   → path() is faster (regex is slower).

□ 6□ What are URL Path Converters?

Types:

- `<int:id>`
- `<str:slug>`
- `<slug:slug>`
- `<uuid:id>`
- `<path:subpath>`

Real-Time Example

```
path("profile/<uuid:user_id>/", views.profile)
```

Used in microservices and distributed systems.

Follow-up Questions

1. Difference between slug and str?
2. Why use UUID instead of int?

Cross-Questions

1. Can you create your own custom path converter?
   → Yes.

□ 7□ What is reverse URL lookup?

Beginner Explanation

Instead of writing URL manually, Django finds it by name.

Example:

```
return redirect("notes:list")
```

Intermediate Explanation

reverse() ensures:

- You can rename URLs safely
- No hardcoding
- Clean code

Advanced Explanation

Used for:
✔ Redirections
✔ Template linking `{% url 'notes:add' %}`
✔ Dynamic URL generation
✔ DRF routers

Follow-up Questions

1. What happens if URL name doesn't exist?
   → NoReverseMatch.

Cross-Questions

1. Why reverse lookup is crucial for large apps?
   → Easier maintenance & refactoring.

□ 8□□ Real-Time: Complete URL Structure for a Large Notes App

```
project/
 └── urls.py
       path("", include("core.urls"))
       path("notes/", include("notes.urls"))
       path("accounts/", include("accounts.urls"))
```

```
    path("api/v1/", include("api.urls"))
```

Notes App (notes/urls.py)

```
app_name = "notes"

urlpatterns = [
    path("", views.notes_list, name="list"),
    path("add/", views.add_note, name="add"),
    path("edit/<int:id>/", views.edit_note, name="edit"),
    path("delete/<int:id>/", views.delete_note, name="delete"),
]
```

9Important Interview Questions for Day-8

1. Why do we use app_name?
2. What is namespacing? Types? (application namespace, instance namespace)
3. What is the difference between `url`, `path`, `re_path`?
4. How does Django resolve URL conflicts?
5. How to structure URLs for enterprise-level apps?
6. How to implement API versioning in URLs?
7. What causes NoReverseMatch exceptions?
8. How to debug URL errors?
9. Why do large Django projects need modular URL design?
10. What tools help visualize URL routes? (django-extensions)

What are Class-Based Views (CBVs) in Django?

Answer:

Class-Based Views are Django views written as **Python classes** instead of functions.
They allow:

- Code reusability
- Cleaner structure
- Inbuilt generic behavior (List, Detail, Create, Update, Delete)
- Method overloading using `get()`, `post()`, `dispatch()`

Real-time Example:
Your project has many CRUD pages.
Using CBVs, you avoid writing repeated logic for:

- Listing objects
- Showing a single object

- Creating new objects
- Editing existing entries
- Deleting data safely

Follow-up Questions:

1 FBV vs CBV?

- FBV → function, simple, readable
- CBV → class, reusable, powerful, but harder for beginners

2 Why do companies prefer CBVs?

- Faster development
- Standard structure
- Easy to extend

Cross-Questions:

- What is the `dispatch()` method?
  → It decides whether to call `get()`, `post()`, `put()`, etc.
- Can CBVs use decorators?
  → Yes, through `method_decorator`.

Q2. What is ListView? Explain in detail.

Answer:

`ListView` is a generic CBV used to display a **list of database objects**.

Key attributes:

Attribute	Meaning
`model`	Which model to fetch
`template_name`	Which HTML to load

Attribute	Meaning
`context_object_name`	Custom variable name for template
`paginate_by`	Pagination
`ordering`	Sort results

Example:

```
class NoteListView(ListView):
    model = Note
    template_name = 'notes/list.html'
    context_object_name = 'notes'
    ordering = ['-created_at']
    paginate_by = 10
```

Real-time Example:

Every dashboard in companies includes a **table view**:
Employees, Payments, Orders, Transactions → all use ListView logic.

Follow-up Questions:

1. How to filter inside ListView?

```
def get_queryset(self):
    return Note.objects.filter(user=self.request.user)
```

2. How to add extra data?

```
def get_context_data(self, **kwargs):
    ctx = super().get_context_data(**kwargs)
    ctx['title'] = "My Notes"
    return ctx
```

Cross-Questions:

- What happens if `template_name` is not given?
  → Django uses default path: `app/model_list.html`.
- How does ListView know which model fields to fetch?
  → From `model` attribute.

□ Q3. What is DetailView? Explain.

Answer:

`DetailView` shows the details of **one specific object**, identified by:

- `pk`
- `slug`

Example:

```
class NoteDetailView(DetailView):
    model = Note
    template_name = 'notes/detail.html'
    context_object_name = 'note'
```

Real-time Example:

Product details page → Amazon product page
Blog single article → Medium post

Follow-up Questions:

1. What if object is not found?
   → Raises `404`.
2. How to change lookup?

```
slug_field = 'title'
slug_url_kwarg = 'title'
```

Cross-Questions:

- How does DetailView fetch the record?
  → Using `get_object()` internally.

□ Q4. Explain CreateView with example.

Answer:

CreateView is used to display a **form** and **save the data**.

Required attributes:

- `model`
- `fields` or `form_class`
- `template_name`
- `success_url`

Example:

```
class NoteCreateView(CreateView):
    model = Note
    fields = ['title', 'content']
    template_name = 'notes/form.html'
    success_url = '/notes/'
```

Real-time Example:

Used in:

- Signup forms
- Add product screen
- Add customer form
- Create blog post

Follow-up Questions:

1. How to attach user to CreateView?

```
def form_valid(self, form):
    form.instance.user = self.request.user
    return super().form_valid(form)
```

2. How to add validations?
   → Use Model.clean() or form validation methods.

Cross-Questions:

- What happens after saving?
  → Redirects using `success_url`.

□ Q5. Explain UpdateView in detail.

Answer:

UpdateView is used to edit an existing object.

Example:

```
class NoteUpdateView(UpdateView):
    model = Note
    fields = ['title', 'content']
    template_name = 'notes/form.html'
    success_url = '/notes/'
```

Real-time Example:

Editing:

- Profile
- Notes
- Blog posts
- Products

Follow-up Questions:

1. How does Django know which object to update?
   → Uses pk from the URL.
2. How to check user permissions?
   → Override `dispatch()`.

Cross-Questions:

- Difference between CreateView vs UpdateView?
  → Create = new object; Update = edit existing.

□ Q6. Explain DeleteView in detail.

Answer:

DeleteView is used to remove an object.

Example:

```
class NoteDeleteView(DeleteView):
    model = Note
    template_name = 'notes/confirm_delete.html'
    success_url = '/notes/'
```

Real-time Example:

- Delete account
- Remove product
- Delete message

Follow-up Questions:

1. Why DeleteView requires confirmation template?
   → To prevent accidental deletes.
2. How to restrict delete only to owner?

```
def dispatch(self, request, *args, **kwargs):
    obj = self.get_object()
    if obj.user != request.user:
        raise PermissionDenied
    return super().dispatch(request, *args, **kwargs)
```

Cross-Questions:

- Why DeleteView performs POST delete instead of GET?
  → GET must not change server state → security.

Q7. What is get_queryset(), get_object(), get_context_data()?

Answer:

Method	Purpose
`get_queryset()`	Modify list data
`get_object()`	Modify detail retrieval
`get_context_data()`	Add extra variables to template

Real-time Example:

You want to show **only the logged-in user's notes**:
→ Override `get_queryset()`.

Follow-up Question:

Why should we override only necessary methods?
→ For clean, maintainable code.

Cross-Questions:

- Difference between mixins vs overriding methods?

□ Q8. What are Mixins in CBVs?

Answer:

Mixins are small reusable classes that add functionality to CBVs.

Common mixins:

- `LoginRequiredMixin`
- `UserPassesTestMixin`
- `PermissionRequiredMixin`

Example:

```
class SecureView(LoginRequiredMixin, ListView):
    model = Note
```

Follow-up Questions:

1. Why Mixins must come before parent class?
   → Due to MRO (Method Resolution Order).
2. What is MRO?
   → Python's order for resolving methods in inheritance.

Cross-Questions:

- What happens if mixin is placed after the parent class?
  → Login check may not work.

□ Q9. Full CRUD Example Using CBVs (Complete Flow)

urls.py

```
urlpatterns = [
    path('', NoteListView.as_view(), name='list'),
    path('<int:pk>/', NoteDetailView.as_view(), name='detail'),
    path('add/', NoteCreateView.as_view(), name='add'),
    path('<int:pk>/edit/', NoteUpdateView.as_view(), name='edit'),
    path('<int:pk>/delete/', NoteDeleteView.as_view(), name='delete'),
]
```

You will be asked this in interviews.

Q1. What is MEDIA_ROOT in Django? Explain clearly.

Answer:

`MEDIA_ROOT` is the **absolute file-system path** where Django stores uploaded files.

Located inside **settings.py**:

```
MEDIA_ROOT = BASE_DIR / 'media'
```

Whenever a user uploads a file using:

- FileField
- ImageField
  Django stores them inside this MEDIA_ROOT directory.

Real-Time Example:

If your app allows users to upload:

- Profile pictures
- Documents
- Attachments
- Product images

→ All these files are physically stored inside `media/`.

Follow-up Questions:

1 Why we never store media inside STATIC folder?
→ Because static files are bundled & cached; uploaded media changes frequently.

2 Can MEDIA_ROOT exist outside the project folder?
→ Yes, many companies store it on **AWS S3** or external storage.

Cross-Questions:

- What happens if MEDIA_ROOT is not configured?
  → File upload fails with a FileSystemStorage error.
- Is MEDIA_ROOT used in production?
  → Only if serving directly, otherwise stored in cloud (S3, GCP, Azure).

□ Q2. What is MEDIA_URL? Why do we need it?

Answer:

`MEDIA_URL` is the **public URL path** used to access uploaded files.

Example:

```
MEDIA_URL = '/media/'
```

If a file is stored at:

```
media/profile/user1.jpg
```

It becomes accessible at:

```
http://localhost:8000/media/profile/user1.jpg
```

Django uses MEDIA_URL only in **development**.

Real-time Example:

On your website, user profile picture shows like:

```
<img src="{{ user.profile.image.url }}">
```

`.url` internally uses MEDIA_URL.

Follow-up Questions:

1□ Why MEDIA_URL must end with a slash?
→ To avoid broken URLs.

2□ Can MEDIA_URL be something else?
→ Yes:

```
MEDIA_URL = '/userfiles/'
```

Cross-Questions:

- Static URL vs Media URL difference?
  → static = CSS/JS
  → media = user uploads

□ Q3. How to configure Django for file uploads?

Answer:

Add this in **settings.py:**

```
MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

Add this in **urls.py** (development only):

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Beginner→Advanced Explanation:

- Django development server (runserver) needs this to **serve uploaded files**.
- In production (Nginx, Apache), this part is replaced by the server.

Follow-up Questions:

1 □ Why do we not use static() in production?
→ Static() is slow and insecure.

2 □ Who serves the media files in production?
→ Web server such as **Nginx**.

Cross-Questions:

- What if static() is missing?
  → Uploaded images will show "broken URL".

□ Q4. Explain FileField in Django with an example.

Answer:

`FileField` is used to upload **any type of file**: PDF, DOCX, ZIP, etc.

Example model:

```
class Document(models.Model):
    title = models.CharField(max_length=100)
    file = models.FileField(upload_to='documents/')
```

upload_to meaning:

Uploaded files stored inside:

```
media/documents/
```

Real-time Example:

Used for uploading:

- Resumes
- Reports
- Payment receipts
- CSV files

Follow-up Questions:

1□ Where is file information stored?
→ In the database path:
`documents/myupload.pdf`

2️⃣ Where is the actual file stored?
→ Inside `media/documents/`.

3️⃣ Can FileField validate file size?
→ Yes → override clean().

Cross-Questions:

- What if two files have the same name?
  → Django auto-renames: `file_1.pdf`, `file_2.pdf`.

🔹 Q5. Explain ImageField with example.

Answer:

`ImageField` is a FileField **with image validation**.

Requires **Pillow library**:

```
pip install Pillow
```

Model:

```
class Profile(models.Model):
    name = models.CharField(max_length=100)
    photo = models.ImageField(upload_to='profiles/')
```

Real-time Examples:

- Profile pictures
- Product images
- Blog thumbnails

Follow-up Questions:

1️⃣ What happens if Pillow is not installed?
→ Django raises "ImportError: No module named PIL".

2️⃣ Can ImageField detect corrupt images?
→ Yes.

Cross-Questions:

- Difference between FileField vs ImageField?
  → ImageField performs validation.

❓ Q6. How to show uploaded file in template?

Answer:

Template:

```
<img src="{{ profile.photo.url }}" alt="">
```

OR

```
<a href="{{ document.file.url }}">Download</a>
```

Real-time Example:

On Facebook, Instagram → images are stored in MEDIA_ROOT and served using MEDIA_URL.

Follow-up Questions:

1️⃣ What is `.url`?
→ Returns full file path including MEDIA_URL.

2️⃣ What is `.path`?
→ Absolute server file path.

□ Q7. How does Django store uploaded files internally?

Answer:

Django uses **FileSystemStorage**, which:

- Creates directories
- Handles naming
- Ensures safe paths
- Writes file in binary format

Cross-Question:

How to override default storage?
→ Use custom storage class.

□ Q8. Explain security risks in file uploads.

Answer:

Risk	Example
Executable files upload	.exe, .sh
Path traversal	../../../etc/passwd
Large file upload	Server crash
Malware images	Corrupt files

Solutions:

- Validate file extensions
- Validate size

- Store files on S3 with private permissions
- Use UUID file names

Follow-up Questions:

1 What happens if user uploads 2GB file?
→ Depends on server configuration and `FILE_UPLOAD_MAX_MEMORY_SIZE`.

Q9. How to validate file size and extension?

Answer:

```
def clean(self):
    file = self.file
    if file.size > 5 * 1024 * 1024:
        raise ValidationError("File too large!")

    if not file.name.endswith('.pdf'):
        raise ValidationError("Only PDF files allowed")
```

Cross-Question:

Where else can validations be applied?
→ Form validation or custom validators.

Q10. How to delete file from disk when deleting model?

```
def delete(self, *args, **kwargs):
    self.file.delete()
    super().delete(*args, **kwargs)
```

Follow-up:

Why Django does not auto-delete files?
→ To prevent unintentional data loss.

□ Q11. How to replace old file when updating?

```
def save(self, *args, **kwargs):
    try:
        old_file = Document.objects.get(pk=self.pk).file
        if old_file != self.file:
            old_file.delete()
    except:
        pass
    super().save(*args, **kwargs)
```

□ Q12. File Upload in Forms (Complete Example)

Model:

```
class Document(models.Model):
    file = models.FileField(upload_to='documents/')
```

Form:

```
class DocumentForm(forms.ModelForm):
    class Meta:
        model = Document
        fields = ['file']
```

View:

```
def upload_file(request):
    if request.method == 'POST':
        form = DocumentForm(request.POST, request.FILES)
        if form.is_valid():
            form.save()
```

```
            return redirect('success')
    else:
        form = DocumentForm()
    return render(request, 'upload.html', {'form': form})
```

Template:

```
<form method="POST" enctype="multipart/form-data">
    {% csrf_token %}
    {{ form }}
    <button>Upload</button>
</form>
```

Follow-up Question:

Why `enctype="multipart/form-data"` required?
→ To send binary file data.

DAY 11 — FULL REVISION (DAY 1–10)

Django Fundamentals + Project Flow + Core Concepts + Git Workflow

---

Q1. Explain Django Project Flow (Full Request–Response Cycle).

Answer: (Multi-line, clear, interview level)

Django follows an **MTV architecture (Model–Template–View)**.

When a user enters a URL in the browser:

1 **Browser sends HTTP request**
↓
2 **Django receives request in wsgi.py/asgi.py**
↓
3 **settings.py loads configurations**
↓
4 **URL dispatcher (project/urls.py)** checks pattern
↓
5 It redirects to the correct **app/urls.py**
↓

6 URL maps to a **view function or class**
↓
7 View interacts with **models (database)**
↓
8 View returns HTML by passing data to **template**
↓
9 Django sends **HTTP response to browser**

Real-time Example:

User clicks **"View Notes"** button:
`/notes/` → project/urls.py → notes/urls.py → NoteListView → queries DB → returns list.html template.

Follow-up Questions:

1. What is Django MTV and how is it different from MVC?
2. Which file handles initial configuration of the project?
3. What is URL dispatcher?

Cross-Questions:

- What happens if URL does not match any patterns?
  → Django returns **404 error**
- Can we have multiple apps with same view names?
  → Yes, as long as URL namespaces are unique.

Q2. Explain Models (Day-3 Revision)

Answer:

A **Model** represents a database table in Django.
Each model class becomes one table.
Each model field becomes a database column.

Example:

```
class Note(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
```

Real-time Example:

You create `Note` model → Django creates `notes_note` table in DB.
Insert/update/delete operations become **ORM queries**.

Follow-up Questions:

- Why Django ORM is better than raw SQL?
- What is migration?
- When do we use ForeignKey?

Cross-Questions:

- Can we write raw SQL in Django?
  → Yes (cursor, RawSQL, extra()).
- What is the use of primary key?
  → Django auto-creates `id`.

Q3. Explain Forms (Day-5 Revision)

Answer:

Forms allow collecting user input safely.

Two types:
✔ Django `Form` → manual fields
✔ `ModelForm` → auto-generates fields from model

✅ Real-time Example:

Creating "Add Note" form:

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

This avoids writing HTML repeatedly.

✅ Follow-up Questions:

- Why ModelForm is preferred?
- How does Django prevent CSRF attacks?
- What is `is_valid()`?

✅ Cross-Questions:

- What happens during validation?
- How to show form errors?

Q4. Explain Class-Based Views (Day-9 Revision)

✅ Answer:

CBVs convert common operations into reusable classes.

Examples:

- **ListView** → show list of objects
- **DetailView** → show single object
- **CreateView** → create new record
- **UpdateView** → update record
- **DeleteView** → delete record

✅ Real-world Example:

Notes App CRUD operations implemented in **4 lines each** using CBVs:

```
class NoteListView(ListView):
    model = Note
```

✅ Follow-up Questions:

- Why CBVs are better than FBVs?
- How to override context data?
- How to add custom logic in CBVs?

✅ Cross-Questions:

- What is mixin in Django?
- What is dispatch() method?

Q5. Explain Static & Media Files (Day-10 Revision)

✅ Static Files

Used for: CSS, JS, images used in UI
Configuration:

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

✅ Media Files

Used for: user-uploaded files (images, PDFs)
Configuration:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

Real-time Example:

User uploads a profile picture → stored in `media/profile/`.

Follow-up Questions:

- Difference between static & media files?
- Why MEDIA_URL must be configured in development?

Cross-Questions:

- Why Django does not serve media in production?
  → Nginx/Apache handles it.

Q6. Explain Git & GitHub Workflow

Answer:

Git → version control system
GitHub → cloud hosting for repositories
Repository → project storage with history

Real-time example:

You developed Note App locally → push to GitHub → share with team.

Follow-up Questions:

- What is branching?
- What is merge conflict and how to solve?
- What is git pull vs git fetch?

❓ Cross-Questions:

- What is `.gitignore`?
- Why commits must be small and meaningful?

Q7. Full CRUD Workflow (Day-6 + Project Revision)

Full cycle:

1. User visits → List view
2. Click Add → Create view
3. Edit → Update view
4. Delete → Delete view

❓ Real-time Example:

In a company:
Invoice App → CRUD for invoices

❓ Follow-up Questions:

- Why POST method is used for delete?
- Can delete be done using GET? (No, unsafe)

❓ Cross-Questions:

- What is optimistic vs pessimistic locking?
- What is transaction.atomic?

Q8. URL Routing & Namespacing (Day-8 Revision)

Answer:

`include()` organizes URLs.
`app_name` avoids conflict.

Real-world Example:

`notes:list` uniquely identifies list page of Notes App.

Follow-up Questions:

- Why use namespacing?
- Can we include same app multiple times?

Cross-Questions:

- What is reverse() method?
- What is lazy reverse?

Q9. Templates & Template Inheritance (Day-2 Revision)

Answer:

Templates help in rendering dynamic HTML.

Template inheritance:

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

Real-time Example:

All pages share:

- navbar
- footer
- theme colors

Follow-up Questions:

- What are Django filters?
- How `{{ variable }}` works?

Cross-Questions:

- What is autoescape?
- How to avoid XSS in templates?

Q10. Migrations (Day-3 Revision)

Answer:

Migrations track database schema changes.

Commands:

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

Follow-up Questions:

- What is migration file?
- Can you roll back migration?

❓ Cross-Questions:

- What happens if alter table manually in DB?

END RESULT — After Day-11 You Should Know:

✔ Project Flow
✔ URL Dispatching
✔ Models & ORM
✔ Forms & Validation
✔ CBVs
✔ Templates
✔ Static/Media
✔ Git Workflow
✔ CRUD End-to-End
✔ Migrations & Database

Django Messages Framework (Interview Q&A)

1️⃣ What is the Django Messages Framework? (Beginner)

Explanation

The Django Messages Framework is a built-in system that allows you to **display one-time notifications** to users after certain actions—such as login, logout, form submission, or CRUD operations.
These messages persist for only **one request-response cycle**, making them ideal for user feedback.

Django stores these messages temporarily using the session or cookie-based storage and automatically clears them after rendering.

Real-Time Example

- When a user logs in → “Login successful”
- When a form fails → “Please correct the errors”
- When a note gets deleted → “Note deleted successfully”

Follow-up Question

Q: Why are messages called "flash messages"?
A: Because they appear only once and disappear automatically after the next page load.

Cross-question

Q: Are Django messages stored in the database?
A: No. They use session or cookie-based temporary storage, not the database.

---

2 Name the default levels in the Django Messages Framework. (Beginner)

Explanation

Django provides **five predefined message levels**, each associated with a priority:

1. **DEBUG**
2. **INFO**
3. **SUCCESS**
4. **WARNING**
5. **ERROR**

These help you classify the importance and visibility of messages.

Real-Time Example

- SUCCESS → "Your profile has been updated."
- ERROR → "Invalid credentials."

Follow-up Question

Q: Can we add custom message levels?
A: Yes, using `MESSAGE_LEVEL` settings.

Cross-question

Q: Which message level appears if no level is specified?
A: INFO.

---

3 How do you implement messages in Django? (Intermediate)

Explanation

You need to follow three steps:

Step 1: Import and add a message in View

```
from django.contrib import messages

def login_user(request):
    messages.success(request, "Login successful!")
    return redirect('home')
```

Step 2: Display messages in Template

```
{% if messages %}
  {% for msg in messages %}
    <div class="alert alert-{{ msg.tags }}">
        {{ msg }}
    </div>
  {% endfor %}
{% endif %}
```

Step 3: Include Bootstrap or custom CSS

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css
">
```

Real-Time Example

In the Notes App, after creating a note:

```
messages.success(request, "Note added successfully!")
```

Follow-up Question

Q: Why do we use `msg.tags` in templates?
A: It converts Django message levels into CSS-friendly class names.

Cross-question

Q: What happens if we forget to include `{% for msg in messages %}`?
A: Messages will be stored but never shown to the user.

---

4 What message storage backends exist in Django? (Intermediate)

Explanation

Django provides **four backends**:

1. **CookieStorage**
2. **SessionStorage** (default)
3. **FallbackStorage** (session → cookie)
4. **Custom Storage** (user-defined)

Default storage uses user sessions, meaning messages are tied to authenticated or anonymous session data.

Real-Time Example

If cookies are disabled:

- FallbackStorage stores messages in cookies.

Follow-up Question

Q: Where do you configure storage backend?
A:

```
MESSAGE_STORAGE = 'django.contrib.messages.storage.session.SessionStorage'
```

Cross-question

Q: Why is session storage preferred?
A: It’s secure, persistent, and not limited by cookie size.

---

5 Why do we connect messages with Bootstrap alerts? (Intermediate)

Explanation

Bootstrap alerts give messages a **professional UI**, using color-coded alert classes like:

- `alert-success`
- `alert-warning`
- `alert-danger`

Django message tags match Bootstrap class names, providing seamless styling.

Real-Time Example

When a note fails validation:

```
messages.error(request, "Title must not be empty!")
```

Bootstrap displays it as a red alert box.

Follow-up Question

Q: Can we override message CSS?
A: Yes, using custom styles or Tailwind classes.

Cross-question

Q: Can we animate messages?
A: Yes, using JavaScript (fade-out after few seconds).

---

6 How does Django internally store and clear messages? (Advanced)

Explanation

Internally:

1. Messages are added to `_messages` attribute of the request.
2. They get stored in session/cookies via middleware: `django.contrib.messages.middleware.MessageMiddleware`
3. After template rendering, they are auto-cleared.

This provides two guarantees:

- Messages persist exactly **one response cycle**
- No manual cleanup needed

Real-Time Example

Redirect flow:

```
POST → add message → redirect → read message → clear
```

Follow-up Question

Q: Why does Django use middleware for messages?
A: To inject a messages storage backend into every request object.

Cross-question

Q: What happens if MessageMiddleware is missing?
A: Messages will not function; Django will throw an attribute error.

---

7 Difference: messages.success() vs messages.add_message() (Advanced)

Explanation

`success()` is a **shortcut method** for readability.
`add_message()` is **low-level and customizable**.

Example:

```
messages.add_message(request, messages.SUCCESS, "Operation completed")
```

Real-Time Example

Dynamic message level:

```
level = messages.ERROR if form.errors else messages.SUCCESS
messages.add_message(request, level, "Result updated")
```

Follow-up Question

Q: Which should you use in large projects?
A: Shortcut methods for readability unless dynamic levels are required.

Cross-question

Q: Can messages be translated?
A: Yes, Django supports i18n for message text.

DAY 14 — Middleware Introduction

1 What is Middleware in Django?

Definition:
Middleware in Django is a **lightweight, low-level plugin system** that processes requests and responses globally, before they reach your views or after they leave your views. Think of it as a pipeline where each middleware can inspect, modify, or act upon requests/responses.

Key points:

- It's **executed for every request** to your Django app.
- Handles **pre-processing** (before view) and **post-processing** (after view).
- Common uses: authentication, logging, security headers, custom error handling.

Real-time example:

- When a user visits your website, a `SecurityMiddleware` ensures HTTPS is used before the request reaches your view.
- Another example: `AuthenticationMiddleware` checks if the user is logged in and attaches `request.user` to the request object.

---

2 How to Create Custom Middleware

Step-by-step:

1. **Create middleware file**: `myproject/middleware.py`

```
# middleware.py

class SimpleLoggingMiddleware:
    """
    Custom middleware to log requests and responses.
    """
    def __init__(self, get_response):
        self.get_response = get_response
        print("Middleware initialized")  # Runs once when server starts

    def __call__(self, request):
        # Pre-processing: before view
        print(f"Incoming request path: {request.path}")

        response = self.get_response(request)

        # Post-processing: after view
        print(f"Outgoing response status: {response.status_code}")

        return response
```

2. **Add to settings.py**

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'myproject.middleware.SimpleLoggingMiddleware',  # <-- Custom
]
```

3. **Test:**

- Visit any URL; check console logs for request path and response status.

---

3 Request/Response Hook Logic

Middleware has two primary hooks:

Hook Type	Method	Runs When	Example Use Case
Pre-request	`__call__` before `get_response`	Before view function	Logging, authentication check, modifying headers

Hook Type	Method	Runs When	Example Use Case
Post-response	`__call__` after `get_response`	After view function	Modifying response, adding headers, logging response

Advanced:
Middleware can also implement these optional hooks:

- `process_view(request, view_func, view_args, view_kwargs)` → Modify behavior before view.
- `process_exception(request, exception)` → Handle exceptions globally.
- `process_template_response(request, response)` → Modify template responses.

---

4 Interview Q&A

Beginner

- **Q:** What is middleware in Django?
  A: Middleware is a framework of hooks into Django's request/response processing. It lets you globally process requests and responses.
- **Follow-up:** Give examples of built-in middleware in Django.
  Answer: `AuthenticationMiddleware`, `SecurityMiddleware`, `SessionMiddleware`.

Intermediate

- **Q:** How do you create custom middleware?
  A: Create a class with `__init__` and `__call__` methods, implement request/response hooks, and add it to `MIDDLEWARE` list.
- **Cross-question:** Why is `__init__` used in middleware?
  A: It runs **once** at server startup, suitable for initialization.

Advanced

- **Q:** Explain the difference between `__call__` and `process_view` middleware methods.
  A:
  - `__call__`: Handles pre/post request globally.
  - `process_view`: Runs **just before the view function**, can inspect or modify view args.
- **Follow-up scenario:** Implement middleware that blocks requests from a specific IP.

```
class BlockIPMiddleware:
    BLOCKED_IPS = ['192.168.1.10']
```

```
def __init__(self, get_response):
    self.get_response = get_response

def __call__(self, request):
    if request.META.get('REMOTE_ADDR') in self.BLOCKED_IPS:
        from django.http import HttpResponseForbidden
        return HttpResponseForbidden("Access Denied")
    return self.get_response(request)
```

- **Cross-question:** How does middleware order in `settings.py` affect execution?
  Answer: Middleware executes **top → bottom for request**, **bottom → top for response**.

---

5 Practice / Real-time Scenario

Scenario:
You are building a SaaS platform. You need middleware to:

1. Log user activity.
2. Inject custom security headers.
3. Track API response time.

Task: Implement one middleware that does **all three** in a single class. Test with different URLs.