Django

Here is a **complete list of Django topics** from **basic to advanced**, organized step-by-step. This will help you learn Django systematically.

✅ 1. Introduction to Django

- What is Django?
- Features of Django
- Django vs Flask
- MVC vs MVT architecture
- Installing Django (with virtual environment)

✅ 2. Creating Your First Django Project

- `django-admin` vs `manage.py`
- Creating a project: `django-admin startproject`
- Django project structure
- Running the development server
- Understanding `settings.py`, `urls.py`, `wsgi.py`, `asgi.py`

✅ 3. Django Apps

- Creating an app: `python manage.py startapp`
- Project vs App
- Adding app to `INSTALLED_APPS`
- Directory structure of a Django app

✅ 4. URL Routing

- `urls.py` in project and app
- Including app URLs in project

- Dynamic URLs (e.g., `/post/<int:id>/`)
- Named URL patterns

✅ 5. Views

- Function-based views
- Returning `HttpResponse`, `JsonResponse`
- `render()` for HTML templates
- URL to view mapping

✅ 6. Templates

- `templates` folder and structure
- Template tags: `{{ }}` and `{% %}`
- Template inheritance with `{% block %}`
- Passing data from views to templates
- Static files (CSS, JS, images)

✅ 7. Models and Database

- Defining models in `models.py`
- Fields: `CharField`, `IntegerField`, `DateField`, etc.
- `makemigrations` and `migrate`
- Admin site basics (`python manage.py createsuperuser`)
- Registering models in `admin.py`

✅ 8. Django ORM (Object Relational Mapper)

- CRUD operations
  - Create: `Model.objects.create()`
  - Read: `Model.objects.all()`, `filter()`, `get()`
  - Update: `save()`

    - Delete: `delete()`
- QuerySet methods: `count()`, `order_by()`, `exclude()`, etc.

✅ 9. Forms in Django

- Django Form class
- Rendering and validating forms
- Handling GET and POST in views
- ModelForm: Creating forms from models

✅ 10. User Authentication

- User model and `UserCreationForm`
- Login, logout, register
- LoginRequiredMixin
- Permissions and groups
- Custom user model (optional)

✅ 11. Messages Framework

- Adding messages (e.g., success, error)
- Displaying messages in templates
- Using `messages` in views

✅ 12. Middleware

- What is middleware?
- Built-in middleware
- Creating custom middleware

✅ 13. Django Admin (Advanced)

- Customizing admin panel
- Adding filters, search fields
- Inline models
- Custom admin actions

✅ 14. Static and Media Files

- Serving static files in development
- `STATICFILES_DIRS`, `STATIC_ROOT`
- Handling file uploads (media files)
- `MEDIA_URL`, `MEDIA_ROOT`

✅ 15. Class-Based Views (CBVs)

- Introduction to CBVs
- `ListView`, `DetailView`, `CreateView`, `UpdateView`, `DeleteView`
- Overriding CBV methods
- Mixins

✅ 16. Django REST Framework (Optional - For APIs)

- What is DRF?
- Serializers
- API Views and ViewSets
- Routers
- Authentication and permissions

✅ 17. Testing in Django

- Unit tests with `TestCase`
- Client for testing views
- Testing models and forms

✅ 18. Deployment

- Settings for production
- Using Gunicorn and Nginx
- Deploying to Heroku / Railway / PythonAnywhere
- Using `.env` files and `django-environ`

✅ 19. Security in Django

- CSRF protection
- XSS, SQL injection protection
- Secure password storage
- HTTPS setup

✅ 20. Advanced Topics (Optional)

- Signals
- Caching
- Celery (for background tasks)
- Channels (for WebSockets)
- Internationalization (i18n)

Would you like me to start explaining these topics **step by step** in simple terms with code examples?

Great! Let's begin your Django journey step by step. We'll start with **Topic 1: Introduction to Django**.

🧭 1. Introduction to Django

🔷 What is Django?

Django is a **high-level Python web framework** that allows you to build **secure, scalable, and maintainable web applications quickly**.

It follows the **"batteries-included"** philosophy — meaning it comes with many built-in tools like:

- User authentication
- Admin panel
- URL routing
- Form handling
- ORM (Object-Relational Mapping)

🔷 Why Use Django? (Features)

Feature	Description
✅ Fast Development	Create apps quickly with less code
✅ Secure	Protects against common attacks like SQL injection, CSRF
✅ Scalable	Used by big websites (Instagram, Pinterest)
✅ Versatile	Suitable for all types of web applications
✅ Built-in Admin	Auto-generated admin dashboard for models
✅ DRY Principle	"Don't Repeat Yourself" - reusable code

🔷 Django Architecture: MVT

Django uses **MVT architecture**, which stands for:

Component	Description
Model	Defines your database structure
View	Controls logic & processes requests
Template	Handles front-end HTML presentation

Example:

- User visits `/products/`
- View handles the logic to get product list
- Model fetches product data from DB
- Template shows the products in HTML

🔷 Django vs Flask (Comparison)

Django	Flask
Full-featured	Minimal (micro framework)
Built-in admin panel	No admin by default
ORM included	Needs extension
Great for big projects	Great for small apps

🔷 Installing Django (Step-by-Step)

✅ Step 1: Create a virtual environment (recommended)

python -m venv myenv

Activate it:

Windows:
myenv\Scripts\activate

-

Mac/Linux:
source myenv/bin/activate

-

✅ Step 2: Install Django

pip install django

✅ Step 3: Check Django version

django-admin --version

Would you like to move on to **Topic 2: Creating Your First Django Project**?

Great! Let's move to **Topic 2: Creating Your First Django Project** 🚀

🛠️ 2. Creating Your First Django Project

🔹 Step 1: Create a Django Project

📌 **Command:**
django-admin startproject myproject

This creates a folder named `myproject` with the following structure:

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

📁 Project Folder Structure Explanation

File/Folder	Purpose
`manage.py`	Command-line tool for managing the project (run server, migrations, etc.)
`myproject/`	Inner folder that contains actual project code
`__init__.py`	Tells Python this is a package
`settings.py`	Project settings (DB, apps, middleware, etc.)
`urls.py`	URL declarations
`asgi.py`	ASGI entry point (for async support)
`wsgi.py`	WSGI entry point (used by most production servers)

🔹 Step 2: Run the Development Server

Change into the project directory and run:

```
cd myproject
python manage.py runserver
```

You'll see:

Starting development server at http://127.0.0.1:8000/

Open that URL in your browser — you'll see the **Django welcome page** 🎉

🔹 Step 3: Understanding `manage.py`

This file is used to run Django commands:

Command	Description
`runserver`	Starts the development server
`startapp`	Creates a new Django app
`makemigrations`	Prepares DB changes
`migrate`	Applies DB changes
`createsuperuser`	Creates admin user

✅ Summary

You've now:

- Installed Django
- Created your first project
- Understood the project structure
- Ran the server and saw the welcome page

Shall we move to **Topic 3: Django Apps** next?

Let's continue with **Topic 3: Django Apps** 🧩

🧱 3. Django Apps

In Django, a **project** is made up of **apps**.

🔹 An **app** is a module that does one specific task — like handling blog posts, users, or products.

🔹 Why Apps?

- Apps make Django projects modular and reusable.
- You can plug the same app into different projects.

🛠️ Step-by-Step: Creating a Django App

📌 Command:
python manage.py startapp myapp

It creates a folder named `myapp/` like this:

```
myapp/
├── __init__.py
├── admin.py
├── apps.py
├── migrations/
│   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

📁 App File Structure Explained

File	Purpose
`models.py`	Define database models here
`views.py`	Logic for handling requests
`admin.py`	Register models in admin panel
`apps.py`	App configuration
`migrations/`	Tracks database changes

`tests.py`	For writing unit tests

🧩 Add App to Project

After creating the app, add it to your project in `settings.py`:

```
# myproject/settings.py
INSTALLED_APPS = [
    ...
    'myapp',  # Add this line
]
```

Without this step, Django won't recognize your app.

🧠 Important Notes

- You can create multiple apps in a project.
- Apps should be small, reusable, and do one thing well.
- Example apps: `blog`, `accounts`, `store`, `chat`, etc.

✅ **You've learned:**

- What a Django app is
- How to create an app
- How to connect it to a project

Ready to move to **Topic 4: URL Routing**?

Great! Let's continue with **Topic 4: URL Routing** 🌐

🔀 4. URL Routing in Django

🔹 What is URL Routing?

URL routing maps a URL (like `/about/`) to a **view function**, which returns a response (usually HTML).

Think of it as:

URL → View Function → Response

📁 Django's URL System

- **Project-level** `urls.py` → the main router
- **App-level** `urls.py` → routes specific to each app

✅ Step 1: Project `urls.py`

Located in `myproject/urls.py`:

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),  # Connect app-level urls
]
```

✅ Step 2: Create App `urls.py`

Inside your app (`myapp`), **create** a `urls.py` file (if not already there):

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

urlpatterns = [
    path('', views.home, name='home'),  # Homepage
    path('about/', views.about, name='about'),  # About page
]
```

✅ Step 3: Define Views in `views.py`

```
# myapp/views.py
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello, this is the Home Page!")

def about(request):
    return HttpResponse("This is the About Page.")
```

🔁 Dynamic URLs (with variables)

```
# myapp/urls.py
path('user/<int:id>/', views.user_profile, name='user_profile')

# myapp/views.py
def user_profile(request, id):
    return HttpResponse(f"User ID is {id}")
```

⛔ Named URLs

Naming a URL lets you refer to it in templates or redirects:

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

You can use this in templates:

```
<a href="{% url 'about' %}">About</a>
```

✅ Summary:

- Project `urls.py` = entry point
- App `urls.py` = handles app-specific routes
- Use `path()` for static and dynamic URLs
- Use `include()` to connect app URLs to project

Shall we move to **Topic 5: Views** next?

Let's move on to **Topic 5: Views in Django** 👀

🧠 5. Views in Django

🔹 What is a View?

A **view** is a Python function or class that takes a web request and returns a web response.

Think of it as the "brain" — it processes data, talks to the database, and returns HTML, JSON, etc.

✅ Types of Views

1. **Function-Based Views (FBV)** – Simple functions
2. **Class-Based Views (CBV)** – Reusable, object-oriented style (covered later)

We'll start with **Function-Based Views (FBV)**.

🛠️ Function-Based Views (FBV)

🔹 Step 1: Create a View in `views.py`

```
# myapp/views.py
from django.http import HttpResponse

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

🔹 Step 2: Map the View in `urls.py`

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

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

]

🧪 Visit http://127.0.0.1:8000/ → You'll see "Welcome to the Home Page!"

🗒️ Using `render()` to Load HTML Templates

```
from django.shortcuts import render

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

You need a `templates/home.html` file:

```
<!-- templates/home.html -->
<h1>This is the homepage</h1>
```

Make sure Django knows where to find templates (we'll cover in the next topic).

🧪 You can also return JSON or 404 pages:

```
from django.http import JsonResponse, Http404

def data_view(request):
    data = {"message": "Hello JSON!"}
    return JsonResponse(data)

def error_view(request):
    raise Http404("Page not found")
```

✅ Summary

Concept	Purpose
`HttpResponse`	Sends plain text or HTML
`render()`	Loads an HTML template

`JsonResponse`	Sends a JSON response
`Http404`	Shows a 404 error

Next up is **Topic 6: Templates** — where you learn to design beautiful pages with HTML and Django template language.

Ready to continue?

Let's continue with **Topic 6: Templates in Django** 🎨

🖼️ 6. Templates in Django

🔹 What is a Template?

A **template** is an HTML file that contains dynamic content using Django's **Template Language**.

Templates allow you to create dynamic web pages with Python + HTML.

✅ Step-by-Step: Using Templates

🔹 Step 1: Create a `templates/` Folder in Your App

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

📌 Best practice: Create it like `myapp/templates/myapp/home.html`

🔹 Step 2: Load the Template from View

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

def home(request):
    return render(request, 'myapp/home.html')
```

🔹 Step 3: Add Template Directory in Settings (optional for projects)

In `settings.py`, make sure this exists:

```
TEMPLATES = [
    {
        ...
        'DIRS': [],
        ...
    },
]
```

If you want global templates folder (not per app), set:

'DIRS': [BASE_DIR / "templates"]

🧠 Django Template Language

1. Variables – `{{ variable }}`

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

2. Tags – `{% tag %}`

Used for logic:

- `{% if %}`, `{% for %}`, `{% include %}`, `{% block %}`, etc.

Example:

```
{% if user.is_authenticated %}
  <p>Welcome, {{ user.username }}</p>
{% else %}
  <p>Please log in.</p>
{% endif %}
```

---

3. For Loop Example

```
<ul>
  {% for item in items %}
    <li>{{ item }}</li>
  {% endfor %}
</ul>
```

In `views.py`:

```
def home(request):
    return render(request, 'myapp/home.html', {'items': ['Apple', 'Banana', 'Mango']})
```

---

4. Template Inheritance

🔹 **Base Template (`base.html`)**

```
<!DOCTYPE html>
<html>
<head><title>{% block title %}My Site{% endblock %}</title></head>
<body>
  <header>Header Here</header>
  {% block content %}{% endblock %}
</body>
</html>
```

🔹 **Child Template (`home.html`)**

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

{% block title %}Home Page{% endblock %}

{% block content %}
  <h1>Welcome to the Home Page!</h1>
{% endblock %}
```

✅ Summary

Concept	Purpose

`{{ variable }}`	Show dynamic data in HTML
`{% if %}`, `{% for %}`	Add logic to templates
`extends`, `block`	Template inheritance
`render()`	Connects view → HTML template

Up next is **Topic 7: Models and Database** — where you'll learn how to create database tables with Python code.

Shall we continue?

Awesome! Let's move to **Topic 7: Models and Database in Django** 🗃️

🧱 7. Models and Database in Django

🔹 What is a Model?

A **model** is a Python class that represents a **database table**.
Each attribute of the class is a **column** in the table.

Django comes with an ORM (Object-Relational Mapper) that handles all DB operations using Python — no SQL needed!

✅ Step-by-Step: Creating a Model

🔹 Step 1: Define a Model in `models.py`

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

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
    is_available = models.BooleanField(default=True)

    def __str__(self):
```

```
        return self.title
```

🔹 Step 2: Register the App in `settings.py`

Make sure `'myapp'` is added to `INSTALLED_APPS`:

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

🔹 Step 3: Make Migrations

Migrations create and update your database schema.

```
python manage.py makemigrations
```

This will create a migration file like `0001_initial.py`.

🔹 Step 4: Apply Migrations

```
python manage.py migrate
```

Now the table `Book` is created in the database! ✅

🔹 Step 5: Check with SQLite (optional)

Django uses SQLite by default. You can open the DB file with:

```
python manage.py dbshell
```

Or use a tool like **DB Browser for SQLite**.

📜 Common Field Types

Field Type	Description
`CharField`	Text (needs `max_length`)
`TextField`	Long text
`IntegerField`	Whole number
`FloatField`	Decimal number
`BooleanField`	True/False
`DateField`	Date only
`DateTimeField`	Date + Time
`EmailField`	Validated email
`ForeignKey`	Many-to-One relation

🔄 Model String Representation

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

This helps when viewing objects in the admin panel or shell.

✅ Summary

Step	What it does
`models.py`	Define your database structure
`makemigrations`	Create migration files
`migrate`	Apply changes to the DB
`__str__()`	String version of model objects

Next up: **Topic 8: Django Admin Panel** – where you'll manage data using Django's built-in interface.

Shall we continue?

Let's continue with **Topic 8: Django Admin Panel** 🛠️

🧑‍💼 8. Django Admin Panel

Django comes with a **built-in admin interface** that lets you manage your models (database tables) through a web UI — no need to write SQL or custom forms.

✅ Step-by-Step: Using the Admin Panel

🔹 Step 1: Create a Superuser

To access the admin panel, create an admin account:

python manage.py createsuperuser

You'll be prompted for:

- Username
- Email
- Password

🔹 Step 2: Run the Server

python manage.py runserver

Then go to:
http://127.0.0.1:8000/admin

Login using the superuser credentials you just created.

🔹 Step 3: Register Your Model in `admin.py`

By default, your models won't appear in the admin until you register them.

```
# myapp/admin.py
from django.contrib import admin
from .models import Book

admin.site.register(Book)
```

Now you'll see **Books** in the admin panel, and you can:

- Add new books
- Edit existing ones
- Delete records
- Search and filter

🧠 Optional: Customize Admin Display

You can customize how models are displayed in admin:

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

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

🛡️ Admin Security

The admin panel:

- Is **only for staff/superusers**
- Should not be exposed in public websites
- Should be behind authentication

✅ Summary

Step	Purpose

`createsuperuser`	Creates admin login
`runserver`	Starts the development server
`admin.site.register()`	Adds model to admin panel
`BookAdmin` class	Customizes model display

Next up is **Topic 9: Django Shell** — for interacting with your models and data using Python commands.

Shall we continue?

Let's move on to **Topic 9: Django Shell** 🐚

🐍 9. Django Shell

Django shell lets you interact with your project's data and models using Python commands. It's like a Python interactive terminal but with Django context loaded.

✅ How to Open Django Shell

Run this command:

python manage.py shell

You'll get a prompt like:

>>>

✅ Common Uses of Django Shell

- Create, read, update, delete (CRUD) database records
- Test queries and logic quickly
- Debug model behavior

🔹 Example: Working with Models in Shell

```
from myapp.models import Book

# Create a new book
book = Book(title='Django Basics', author='John Doe', published_date='2023-01-01')
book.save()

# Retrieve all books
books = Book.objects.all()
print(books)

# Filter books by author
books_by_john = Book.objects.filter(author='John Doe')

# Update a book
book.title = 'Django Basics Updated'
book.save()

# Delete a book
book.delete()
```

🧠 Useful Commands

Command	Description
`Model.objects.all()`	Get all records
`Model.objects.filter()`	Filter records
`Model.objects.get()`	Get single record (throws error if not found)
`.save()`	Save/update record
`.delete()`	Delete record

✅ Summary

Action	How to do it in shell
Open shell	`python manage.py shell`

Import model	`from myapp.models import Book`
Create record	`book = Book(...); book.save()`
Read records	`Book.objects.all()`
Update record	`book.title = 'New'; book.save()`
Delete record	`book.delete()`

Next up: **Topic 10: Forms in Django** — where you'll learn how to create HTML forms and process user input.

Ready to continue?

Let's move on to **Topic 10: Forms in Django** 📝

📝 10. Forms in Django

Forms let users submit data (like registration, login, or adding records). Django makes working with forms easy and secure.

🔹 Two Types of Forms

1. **Django Forms** (manually created forms)
2. **ModelForms** (forms tied to a model for easy database integration)

✅ Step 1: Creating a Simple Form

Create a file `forms.py` in your app:

```
# myapp/forms.py
from django import forms

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

```
    message = forms.CharField(widget=forms.Textarea)
```

✅ Step 2: Use the Form in a View

```
# 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():
            # Process form data
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']
            # For now, just print or save data
            print(name, email, message)
    else:
        form = ContactForm()
    return render(request, 'myapp/contact.html', {'form': form})
```

✅ Step 3: Create Template for Form (`contact.html`)

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

🔹 What is `csrf_token`?

It protects your form from Cross-Site Request Forgery attacks. Always include it inside `<form>` tags.

✅ Step 4: Add URL for Form View

```
# myapp/urls.py
```

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

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

🔹 ModelForm (Form linked to a Model)

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

class BookForm(ModelForm):
    class Meta:
        model = Book
        fields = ['title', 'author', 'published_date', 'is_available']
```

Use it the same way as `ContactForm` in views.

✅ Summary

Concept	Description
`forms.Form`	Manual forms
`ModelForm`	Auto form from model
`is_valid()`	Validate user input
`cleaned_data`	Access cleaned form data
`{% csrf_token %}`	Security token for forms

Next: **Topic 11: Working with Databases - Querysets** — where you learn how to fetch and manipulate data.

Ready?

Continuing with **Topic 11: QuerySets and Database Operations** 🔍

🔍 11. QuerySets and Database Operations

🔹 What is a QuerySet?

A **QuerySet** represents a collection of objects (rows) from your database. It's how you read and filter data in Django.

✅ Common QuerySet Methods

Method	Description	Example
`all()`	Get all records	`Book.objects.all()`
`filter()`	Filter records by condition	`Book.objects.filter(author='John')`
`exclude()`	Exclude certain records	`Book.objects.exclude(is_available=False)`
`get()`	Get one record (throws error if none or many)	`Book.objects.get(id=1)`
`order_by()`	Sort records	`Book.objects.order_by('title')`
`count()`	Count records	`Book.objects.count()`

🔹 Filtering Examples

books = Book.objects.filter(published_date__year=2023)
available_books = Book.objects.filter(is_available=True)

🔹 Chaining QuerySets

You can chain filters:

books = Book.objects.filter(author='John').filter(is_available=True)

🔹 Accessing Records

```
book = Book.objects.get(id=1)
print(book.title)
```

🔹 Creating and Saving Records

```
book = Book(title="New Book", author="Author Name", published_date="2023-01-01")
book.save()
```

🔹 Updating Records

```
book = Book.objects.get(id=1)
book.title = "Updated Title"
book.save()
```

🔹 Deleting Records

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

✅ Summary

Operation	Command
Get all	`Model.objects.all()`
Filter	`Model.objects.filter()`
Get single	`Model.objects.get()`
Create	`instance = Model(...); instance.save()`
Update	Modify fields + `.save()`
Delete	`.delete()`

Next: **Topic 12: Static Files and Media** — handling images, CSS, and JS files.

Ready to continue?

Great! Let's move on to **Topic 12: Static Files and Media in Django** 🎨🖼️

📁 12. Static Files and Media

🔹 What are Static Files?

Static files include **CSS, JavaScript, images**, and other files that don't change dynamically.

🔹 What is Media?

Media files are **user-uploaded files** like profile pictures, documents, etc.

✅ Handling Static Files

Step 1: Create a folder named `static` inside your app

```
myapp/
├── static/
│   └── myapp/
│       ├── style.css
│       └── script.js
```

Step 2: Link static files in your template

```
{% load static %}

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

Step 3: Tell Django where static files are (in `settings.py`)

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

In development, Django automatically serves static files. For production, you need to collect static files (covered later).

✅ Handling Media Files (User Uploads)

Step 1: In `settings.py`, add:

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

Step 2: Modify `urls.py` (project-level)

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

urlpatterns = [
    # your urls here
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Step 3: Create a model with a file/image field

```
from django.db import models

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

Step 4: Upload files using forms (ModelForm)

```
class ProfileForm(forms.ModelForm):
```

```
    class Meta:
        model = Profile
        fields = ['name', 'avatar']
```

In your view, handle file upload:

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

Don't forget to add `enctype="multipart/form-data"` in your HTML form tag!

✅ Summary

Concept	Purpose
`static/` folder	Store CSS, JS, images
`{% load static %}`	Load static files in templates
`MEDIA_URL`	URL path for user uploads
`MEDIA_ROOT`	Folder where uploaded files are saved
`ImageField`	Upload image files

Next topic: **Topic 13: URL Dispatcher and URL Routing** — learn how URLs work in Django!

Want to continue?

Onward to **Topic 13: URL Dispatcher and URL Routing** 🔗

🌐 13. URL Dispatcher and URL Routing

🔹 What is URL Routing?

URL routing maps **URLs** (web addresses) to **views** (functions or classes that return responses).

✅ How URLs Work in Django

1. User requests a URL (e.g., `/books/`)
2. Django checks the **URL patterns** (`urls.py`)
3. Finds matching pattern → Calls corresponding **view**
4. View processes and returns a response (like an HTML page)

Step 1: Create `urls.py` in your app (if not exists)

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

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

Step 2: Include App URLs in Project's `urls.py`

```
# project/urls.py
from django.urls import path, include

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

🔹 URL Pattern Syntax

```
path('route/', view_function, name='url_name')
```

- `'route/'` is the URL pattern (relative path)
- `view_function` is the function that handles requests to that URL
- `name` lets you reference URLs in templates & code

🔹 URL Parameters

Pass variables from URLs to views:

```
# urls.py
path('books/<int:id>/', views.book_detail, name='book_detail')
```

In `views.py`:

```
def book_detail(request, id):
    # Use id to fetch book
    book = Book.objects.get(id=id)
    return render(request, 'myapp/book_detail.html', {'book': book})
```

✅ Summary

Concept	Description
`path()`	Defines a URL route
`include()`	Includes app URLs in project
URL parameters	Pass dynamic parts in URL
`name` parameter	Reference URLs by name

Next topic: **Topic 14: Middleware in Django** — learn what middleware is and how it works.

Shall we continue?

Let's keep going with **Topic 14: Middleware in Django** 🛡️

🛠️ 14. Middleware in Django

🔹 What is Middleware?

Middleware is a **layer** between the request and response. It processes requests before they reach your views and responses before they return to the client.

Think of it as a pipeline where each middleware can modify or inspect the request/response.

🔹 Common Uses of Middleware

- Handle authentication
- Manage sessions
- Add security headers
- Log requests
- Handle CORS (cross-origin resource sharing)

🔹 How Middleware Works

- When a request comes in, Django passes it through each middleware **in order**.
- After the view processes the request, the response goes back through the middleware in **reverse order**.

🔹 Default Middleware Examples in `settings.py`

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

🔹 Creating Custom Middleware (simple example)

```
# myapp/middleware.py
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
```

```
    def __call__(self, request):
        print("Before view")
        response = self.get_response(request)
        print("After view")
        return response
```

Add it in `settings.py`:

MIDDLEWARE.append('myapp.middleware.SimpleMiddleware')

✅ Summary

Concept	Explanation
Middleware	Code that runs before/after views
Order matters	Runs in order on request, reverse on response
Built-in middleware	Security, sessions, auth, etc.
Custom middleware	You can write your own

Next topic: **Topic 15: Django Templates in Detail** — dive deeper into how templates work.

Ready to continue?

Moving on to **Topic 15: Django Templates in Detail** 🎨

🖼️ 15. Django Templates in Detail

🔹 What is a Template?

A **template** is an HTML file mixed with **Django Template Language (DTL)** to create dynamic web pages.

🔹 Template Syntax Basics

Variables:
Display data passed from views:

```
Hello, {{ username }}!
```

-

Tags:
Control logic like loops and conditionals:

```
{% if user.is_authenticated %}
   Welcome, {{ user.username }}!
{% else %}
   Please login.
{% endif %}
```

-

Filters:
Modify variables output:

```
{{ price|floatformat:2 }}  <!-- shows price with 2 decimal places -->
```

-

🔹 Template Inheritance

You can create a **base template** and extend it in other templates.

```
<!-- base.html -->
<html>
  <head>
    <title>{% block title %}My Site{% endblock %}</title>
  </head>
  <body>
    {% block content %}{% endblock %}
  </body>
</html>

<!-- home.html -->
{% extends "base.html" %}

{% block title %}Home Page{% endblock %}
```

```
{% block content %}
  <h1>Welcome to the Home Page</h1>
{% endblock %}
```

🔹 Loading Static Files in Templates

Use `{% load static %}` to include CSS, JS, images:

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

🔹 Passing Context from Views

```
def home(request):
    context = {'username': 'Alice'}
    return render(request, 'home.html', context)
```

In template:

```
Hello, {{ username }}!
```

✅ Summary

Feature	Usage
Variables	`{{ variable }}`
Tags	`{% if %}`, `{% for %}`
Filters	`{{ var
Template Inheritance	`{% extends %}`, `{% block %}`
Static files	`{% load static %}`

Next topic: **Topic 16: Class-Based Views (CBVs)** — learn a powerful way to write views.

Shall we continue?

Alright! Next up is **Topic 16: Class-Based Views (CBVs)** 📚

🏗️ 16. Class-Based Views (CBVs)

🔹 What are CBVs?

CBVs let you write views as **Python classes** instead of functions. They provide reusable, organized, and extendable code.

🔹 Why use CBVs?

- Built-in generic views for common tasks (list, detail, create, update, delete)
- Cleaner and more modular code
- Easy to extend or customize behavior

🔹 Basic CBV Example

```
from django.views import View
from django.http import HttpResponse

class HelloView(View):
    def get(self, request):
        return HttpResponse('Hello from Class-Based View!')
```

URL routing:

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

urlpatterns = [
    path('hello/', HelloView.as_view(), name='hello'),
]
```

🔹 Common Generic CBVs

CBV	Purpose
`ListView`	Display list of objects
`DetailView`	Display detail of one object
`CreateView`	Form to create new object
`UpdateView`	Form to update object
`DeleteView`	Confirm and delete object

🔹 Example: ListView for Books

```
from django.views.generic import ListView
from .models import Book

class BookListView(ListView):
    model = Book
    template_name = 'myapp/book_list.html'  # Optional, defaults to myapp/book_list.html
    context_object_name = 'books'           # Optional, default is 'object_list'
```

✅ Summary

Concept	Details
CBV	Views as Python classes
`as_view()`	Method to convert class to view
Generic Views	Pre-built views for common tasks

Next: **Topic 17: Authentication and Authorization** — handling users and permissions.

Ready to continue?

Next is **Topic 17: Authentication and Authorization** 🔐

🔐 17. Authentication and Authorization

🔹 What is Authentication?

Process of verifying **who the user is** (login, logout, signup).

🔹 What is Authorization?

Process of verifying **what the user can do** (permissions, access control).

✅ Built-in Django Authentication Features

- User model with username, email, password
- Login and logout views
- Password hashing and validation
- User permissions and groups

🔹 Basic User Login Example

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

def login_view(request):
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return redirect('home')
        else:
            error = "Invalid credentials"
            return render(request, 'login.html', {'error': error})
    return render(request, 'login.html')
```

🔹 User Access in Templates

```
{% if user.is_authenticated %}
  Hello, {{ user.username }}!
  <a href="{% url 'logout' %}">Logout</a>
```

```
{% else %}
  <a href="{% url 'login' %}">Login</a>
{% endif %}
```

🔹 Permissions and Decorators

- Use `@login_required` decorator to protect views:

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

@login_required
def dashboard(request):
    # protected view
    ...
```

✅ Summary

Feature	Purpose
Authentication	Verify identity (login/logout)
Authorization	Control access (permissions)
`authenticate()`	Check user credentials
`login()`	Log user in
`@login_required`	Protect views

Next up: **Topic 18: Django REST Framework Basics** — building APIs.

Want to proceed?

Continuing to **Topic 18: Django REST Framework Basics** 🌐🔧

🌐 18. Django REST Framework (DRF) Basics

🔹 What is DRF?

Django REST Framework helps you build **APIs** (interfaces to communicate with apps and services) quickly and easily using Django.

🔹 Key Features of DRF

- Serializers to convert data between JSON and Python objects
- ViewSets for reusable views
- Routers to generate URLs automatically
- Authentication and permissions for APIs

🔹 Installing DRF

```
pip install djangorestframework
```

Add to `settings.py`:

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

🔹 Basic Serializer Example

```
from rest_framework import serializers
from .models import Book

class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = '__all__'
```

🔹 Simple API View

```
from rest_framework.views import APIView
from rest_framework.response import Response
from .models import Book
from .serializers import BookSerializer
```

```
class BookListAPI(APIView):
    def get(self, request):
        books = Book.objects.all()
        serializer = BookSerializer(books, many=True)
        return Response(serializer.data)
```

🔹 URL Routing for API

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

urlpatterns = [
    path('api/books/', BookListAPI.as_view(), name='api_books'),
]
```

✅ Summary

Concept	Description
Serializer	Convert models to JSON and vice versa
APIView	Class-based view for APIs
Response	Return JSON response
`rest_framework`	DRF app for building APIs

Next: **Topic 19: Testing in Django** — how to write tests.

Ready?

Here we go with **Topic 19: Testing in Django** ✅🧪

🧪 19. Testing in Django

🔹 Why test?

- Ensure your app works correctly
- Catch bugs early

- Make future changes safer

🔹 Django's Testing Framework

Built on Python's `unittest` module.

Tests go into files named `tests.py` inside apps.

🔹 Writing a Simple Test

```
from django.test import TestCase
from .models import Book

class BookModelTest(TestCase):
    def setUp(self):
        Book.objects.create(title="Test Book", author="Author")

    def test_book_title(self):
        book = Book.objects.get(title="Test Book")
        self.assertEqual(book.title, "Test Book")
```

🔹 Running Tests

```
python manage.py test
```

🔹 Testing Views Example

```
from django.test import TestCase
from django.urls import reverse

class HomePageTest(TestCase):
    def test_home_page_status(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
```

✅ Summary

What to Test	How to Test
Models	Test model methods and fields
Views	Test responses and templates
Forms	Test form validation

Next topic: **Topic 20: Deployment of Django Applications** — how to put your app live.

Shall we go on?

Moving on to **Topic 20: Deployment of Django Applications** 🚀

🚀 20. Deployment of Django Applications

🔹 What is Deployment?

Making your Django app live on the internet so users can access it.

🔹 Common Deployment Platforms

- **Heroku** — easy cloud platform
- **DigitalOcean** — VPS servers
- **AWS / Google Cloud / Azure** — cloud providers
- **PythonAnywhere** — beginner-friendly hosting

🔹 Basic Steps to Deploy

1. **Prepare your app for production:**
    - Set `DEBUG = False` in `settings.py`
    - Set `ALLOWED_HOSTS` to your domain or IP
    - Use a production-ready database (PostgreSQL, MySQL)
    - Configure static files (`collectstatic`)
2. **Use a WSGI server:**

    - Gunicorn is popular (`pip install gunicorn`)
    - Serves your app in production
3. **Configure a web server:**
    - Nginx or Apache to serve static files & proxy to Gunicorn
4. **Set environment variables securely**
5. **Use HTTPS for security**

🔹 Example: Deploying on Heroku

- Install Heroku CLI

Create `Procfile` with:
web: gunicorn myproject.wsgi

-
- Commit code and push to Heroku Git repo
- Run migrations on Heroku
- Set config vars for secrets

✅ Summary

Step	Description
Prepare settings	DEBUG off, ALLOWED_HOSTS set
Static files	Run `collectstatic`
Use Gunicorn	WSGI server for Django
Use Nginx/Apache	Web server and proxy
Deploy on Platform	Heroku, DigitalOcean, etc.

Next: **Topic 21: Signals in Django** — how apps communicate.

Want me to continue?

Let's move to **Topic 21: Signals in Django** ⚡

⚡ 21. Signals in Django

🔹 What are Signals?

Signals let different parts of your app **talk to each other** when certain events happen.

🔹 Common Use Cases

- Automatically create profile after user signup
- Log actions after saving a model
- Send notifications when data changes

🔹 How Signals Work

- A **sender** sends a signal when something happens (like saving a model)
- A **receiver** listens for that signal and runs code in response

---

🔹 Example: Create Profile on User Creation

```
from django.db.models.signals import post_save
from django.contrib.auth.models import User
from django.dispatch import receiver
from .models import Profile

@receiver(post_save, sender=User)
def create_user_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
```

✅ Summary

Term	Meaning
Signal	Event that happens (e.g., model saved)

Sender	The model or action that triggers signal
Receiver	Function that listens and reacts
Connect	Link receiver to signal

Next topic: **Topic 22: Django Caching** — speed up your app.

Ready to continue?

Here's **Topic 22: Django Caching** ⚡🚀

⚡ 22. Django Caching

🔹 What is Caching?

Caching stores data temporarily to **speed up your site** by reducing database hits or expensive calculations.

🔹 Types of Cache in Django

- **In-memory cache:** Fastest, stores data in RAM (e.g., Memcached, Redis)
- **File-based cache:** Stores cache data in files
- **Database cache:** Stores cache in your database (slower)
- **Local memory cache:** For development only (per process)

🔹 Basic Setup

Add cache settings in `settings.py`:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake',
    }
}
```

🔹 Using Cache in Views

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)  # Cache for 15 minutes
def my_view(request):
    ...
```

🔹 Low-level Cache API

```
from django.core.cache import cache

cache.set('my_key', 'value', 30)  # Store for 30 seconds
value = cache.get('my_key')
```

✅ Summary

Concept	Description
Cache	Store data temporarily
Cache backends	Memory, file, database, etc.
`cache_page`	Decorator to cache view output
`cache.get/set`	Low-level cache access

Next up: **Topic 23: Django Signals Advanced**

Want to continue?

Let's dive into **Topic 23: Advanced Django Signals** 🔄

🔄 23. Django Signals Advanced

🔹 Signal Types

- `pre_save` — before saving a model instance
- `post_save` — after saving a model instance
- `pre_delete` — before deleting an instance
- `post_delete` — after deleting an instance
- `m2m_changed` — when many-to-many relations change

🔹 Example: Update Related Model on Save

```
from django.db.models.signals import post_save
from django.dispatch import receiver
from .models import Order, Inventory

@receiver(post_save, sender=Order)
def update_inventory(sender, instance, **kwargs):
    # Reduce stock when order is placed
    inventory_item = Inventory.objects.get(product=instance.product)
    inventory_item.stock -= instance.quantity
    inventory_item.save()
```

🔹 Disconnect Signals

Sometimes you want to disable a signal temporarily:

```
from django.db.models.signals import post_save
from django.dispatch import receiver

post_save.disconnect(receiver_function, sender=Model)
```

✅ Summary

Signal	When it fires
`pre_save`	Before a model instance saves
`post_save`	After a model instance saves
`pre_delete`	Before a model instance deletes
`post_delete`	After a model instance deletes

`m2m_changed`	When many-to-many relation changes

Next topic: **Topic 24: Django Management Commands**

Want to continue?

Moving on to **Topic 24: Django Management Commands** ⚙️

⚙️ 24. Django Management Commands

🔹 What are Management Commands?

Custom commands you can run from the command line using `manage.py` — useful for tasks like cleanup, data import/export, or custom scripts.

🔹 Creating a Custom Command

1. Inside your app, create a folder structure:

```
your_app/
  management/
    commands/
      __init__.py
      your_command.py
```

2. In `your_command.py`:

```
from django.core.management.base import BaseCommand

class Command(BaseCommand):
  help = 'Description of your command'

  def handle(self, *args, **kwargs):
    self.stdout.write('Hello, this is my custom command!')
```

🔹 Running Your Command

python manage.py your_command

🔹 Use Cases

- Import/export data
- Clear caches
- Generate reports
- Run batch jobs

✅ Summary

Concept	Description
Management commands	Scripts run via `manage.py`
Custom commands	You define tasks to automate
Folder structure	`management/commands/` inside app

Next: **Topic 25: Django Forms Advanced**

Ready to continue?

Got it! Let's cover **Topic 25: Django Forms Advanced** and **Topic 26: Working with Django Signals (Advanced concepts)** together in detail.

📝 25. Django Forms Advanced

🔹 Formsets

- Manage multiple forms on one page.
- Useful for creating/editing multiple related objects.

Example:

```
from django.forms import modelformset_factory
from .models import Book
```

```
BookFormSet = modelformset_factory(Book, fields=('title', 'author'), extra=3)
```

In views:

```
def manage_books(request):
    formset = BookFormSet(queryset=Book.objects.all())
    if request.method == 'POST':
        formset = BookFormSet(request.POST)
        if formset.is_valid():
            formset.save()
    return render(request, 'manage_books.html', {'formset': formset})
```

🔹 Custom Validation

Add validation to forms by overriding `clean()` or `clean_fieldname()`:

```
class BookForm(forms.ModelForm):
    def clean_title(self):
        title = self.cleaned_data['title']
        if "badword" in title:
            raise forms.ValidationError("Inappropriate title")
        return title
```

🔹 Widgets and Styling

Customize form fields with widgets, e.g.:

```
from django import forms

class ContactForm(forms.Form):
    message = forms.CharField(widget=forms.Textarea(attrs={'rows':4, 'cols':40}))
```

⚡ 26. Working with Django Signals (Advanced concepts)

🔹 Using `dispatch_uid`

Avoid duplicate signals by adding `dispatch_uid`:

post_save.connect(receiver_function, sender=MyModel, dispatch_uid="unique_id")

🔹 Sending Custom Signals

Create your own signals:

```
from django.dispatch import Signal

order_completed = Signal(providing_args=["order_id", "user"])

# Send signal
order_completed.send(sender=None, order_id=123, user=request.user)

# Receive signal
@receiver(order_completed)
def handle_order(sender, **kwargs):
    print(f"Order {kwargs['order_id']} completed by {kwargs['user']}")
```

🔹 Signal Performance Tips

- Keep signal handlers fast.
- Avoid complex DB queries in signals.
- Use asynchronous tasks (e.g., Celery) if needed.

Ready for next two topics?

Next up:
27. Django Middleware (Advanced) and **28. Django Internationalization (i18n) and Localization (l10n)**

🛡️ 27. Django Middleware (Advanced)

🔹 What is Middleware?

Middleware is a **hook** between the request and response processing in Django.

It processes requests before they reach views and responses before they reach the browser.

🔹 Writing Custom Middleware

```
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Code before view
        print("Before view")

        response = self.get_response(request)

        # Code after view
        print("After view")
        return response
```

Add to `settings.py` in `MIDDLEWARE` list.

🔹 Common Use Cases

- Logging requests
- Handling CORS
- User authentication checks
- Modifying responses (e.g., add headers)

🌍 28. Django Internationalization (i18n) and Localization (l10n)

🔹 What is i18n & l10n?

- **i18n** = Internationalization, preparing app for multiple languages.
- **l10n** = Localization, adapting app to a specific language or region.

🔹 Setup

In `settings.py`:

```
USE_I18N = True
USE_L10N = True
LANGUAGE_CODE = 'en-us'
LOCALE_PATHS = [BASE_DIR / 'locale']
```

🔹 Mark strings for translation

```
from django.utils.translation import gettext_lazy as _

class MyModel(models.Model):
    name = models.CharField(max_length=100, verbose_name=_("Name"))
```

🔹 Generate translation files

```
django-admin makemessages -l es
```

Translate `.po` files, then compile:

```
django-admin compilemessages
```

Would you like to continue with topics 29 and 30 next?

Here are **Topics 29 and 30** covered in detail:

📂 29. Working with Files and Uploads

🔹 Handling File Uploads

- Use `FileField` or `ImageField` in models:

```
from django.db import models

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

- In forms:

```
from django import forms

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

- In views, handle `request.FILES`:

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

🔹 Configure Media Settings

In `settings.py`:

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

In `urls.py` (development only):

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

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

⚡ 30. Asynchronous Programming in Django

🔹 Async Views

Django 3.1+ supports async views to improve concurrency:

```
from django.http import JsonResponse
import asyncio

async def async_view(request):
    await asyncio.sleep(1)
    return JsonResponse({'message': 'This is async'})
```

🔹 When to Use Async?

- Handling many concurrent connections
- Waiting for slow I/O (e.g., API calls, DB queries with async support)
- Real-time features (websockets)

🔹 Limitations

- Not all Django ORM calls are async yet
- Use async cautiously with third-party libs

Want me to cover **31. Django Channels for WebSockets** and **32. Django Admin Customization (Advanced)** next?

Here's **Topic 31: Django Channels for WebSockets** and **Topic 32: Django Admin Customization (Advanced)** in detail:

🔗 31. Django Channels for WebSockets

🔹 What is Django Channels?

- Extends Django to handle **WebSockets** and **asynchronous tasks**.
- Enables real-time features (chat, notifications).

🔹 Basic Setup

pip install channels

In `settings.py`:

```
INSTALLED_APPS += ['channels']
ASGI_APPLICATION = 'your_project.asgi.application'
```

Create `asgi.py`:

```
import os
from channels.routing import ProtocolTypeRouter, URLRouter
from django.core.asgi import get_asgi_application
import your_app.routing

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'your_project.settings')

application = ProtocolTypeRouter({
    "http": get_asgi_application(),
    "websocket": URLRouter(your_app.routing.websocket_urlpatterns),
})
```

🔹 Example WebSocket Consumer

```
from channels.generic.websocket import AsyncWebsocketConsumer
import json

class ChatConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        await self.accept()

    async def receive(self, text_data):
        await self.send(text_data=json.dumps({'message': text_data}))
```

🛠️ 32. Django Admin Customization (Advanced)

🔹 Custom Admin Actions

Add buttons in admin list page:

```
from django.contrib import admin

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

make_published.short_description = "Mark selected as published"

class PostAdmin(admin.ModelAdmin):
    actions = [make_published]
```

🔹 Customize Admin Forms

Override form fields or validation in admin:

```
class PostAdminForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = '__all__'

    def clean_title(self):
        title = self.cleaned_data['title']
        if "bad" in title:
            raise forms.ValidationError("Invalid title")
        return title

class PostAdmin(admin.ModelAdmin):
    form = PostAdminForm
```

Ready to continue with **Topics 33 and 34: Security & Debugging**?

Let's continue with **Topic 33: Security Best Practices in Django** and **Topic 34: Django Debugging and Profiling**:

🔒 33. Security Best Practices in Django

🔹 Key Points

- **Keep `DEBUG = False` in production**

- Use strong, unique **SECRET_KEY**
- Set `ALLOWED_HOSTS` correctly
- Use **HTTPS** (enable SSL/TLS)
- Protect against **CSRF** attacks (Django does it by default)
- Use **X-Content-Type-Options**, **X-Frame-Options** headers
- Keep dependencies updated
- Avoid **SQL Injection** by using ORM, not raw SQL
- Use **Django's authentication system** securely

🔹 Useful Django Security Settings

```
SECURE_BROWSER_XSS_FILTER = True
SECURE_CONTENT_TYPE_NOSNIFF = True
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
X_FRAME_OPTIONS = 'DENY'
```

🐞 34. Django Debugging and Profiling

🔹 Tools and Tips

- Use `print()` or **logging** for quick checks
- Use **Django Debug Toolbar** for detailed info (SQL, templates)
- Use `pdb` for interactive debugging
- Profile slow queries with Django Debug Toolbar or logging
- Use `django-extensions` shell_plus for enhanced shell

🔹 Installing Debug Toolbar

```
pip install django-debug-toolbar
```

In `settings.py`:

```
INSTALLED_APPS += ['debug_toolbar']

MIDDLEWARE = [
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    # other middleware
```

```
]

INTERNAL_IPS = ['127.0.0.1']
```

In `urls.py`:

```
import debug_toolbar
urlpatterns = [
    path('__debug__/', include(debug_toolbar.urls)),
]
```

Shall we continue with **Topic 35: Performance Optimization** and **Topic 36: Django with Frontend Frameworks**?

Here are **Topics 35 and 36** explained in detail:

⚡ 35. Performance Optimization in Django

🔹 Key Tips

- **Use database indexing** on frequently queried fields.
- **Optimize queries:** use `select_related()` and `prefetch_related()` to reduce DB hits.
- **Cache** expensive computations or query results.
- Use **pagination** to limit data sent to users.
- Serve **static files** with a web server or CDN.
- Avoid loading unnecessary data in templates.
- Use asynchronous tasks (like Celery) for heavy jobs.
- Monitor performance with profiling tools.

🌐 36. Django with Frontend Frameworks

🔹 Integrating React/Vue/Angular

- Build frontend as a **separate app** using React, Vue, etc.
- Use **Django REST Framework** (DRF) to provide API endpoints.

- Frontend communicates with Django backend via API calls.
- Serve frontend separately or use Django to serve built frontend files.

🔹 Basic Workflow

1. Create React/Vue app (`npx create-react-app`)
2. Build API in Django using DRF
3. Fetch API data in frontend using fetch/Axios
4. Use CORS middleware in Django to allow frontend requests:

```
INSTALLED_APPS += ['corsheaders']
MIDDLEWARE = ['corsheaders.middleware.CorsMiddleware'] + MIDDLEWARE
CORS_ORIGIN_ALLOW_ALL = True  # for development only
```

Ready for **Topics 37 and 38: Integrating Third-party Packages & Django Logging**?

Here's **Topic 37: Integrating Third-party Packages** and **Topic 38: Django Logging** in detail:

📦 37. Integrating Third-party Packages

🔹 Why Use Them?

- Save development time
- Add complex features quickly (e.g., authentication, payments)

🔹 Common Useful Packages

- **Django REST Framework** (for APIs)
- **django-allauth** (for user authentication)
- **Celery** (for async tasks)
- **django-cors-headers** (for handling CORS)
- **django-crispy-forms** (better form rendering)

🔹 How to Integrate

1. Install via pip:

```
pip install package_name
```

2. Add to `INSTALLED_APPS` in `settings.py`

3. Follow package docs to configure and use

📝 38. Django Logging

🔹 Why Logging?

- Track errors and warnings
- Monitor app behavior
- Debug issues

🔹 Basic Logging Setup (`settings.py`):

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': 'debug.log',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['file'],
            'level': 'DEBUG',
            'propagate': True,
        },
    },
}
```

Logs will be saved in `debug.log`.

Want to continue with **Topics 39 and 40: Signals Advanced & Deployment**?

Wrapping up with:

🔔 39. Django Signals Advanced (summary)

- Use `dispatch_uid` to avoid duplicate signals
- Create and send **custom signals**
- Keep signal handlers lightweight for performance
- Use async tasks if signals need heavy work

🚀 40. Deployment

Steps for deploying Django app:

1. Choose server (e.g., VPS, Heroku, AWS)
2. Use **WSGI** (Gunicorn) or **ASGI** for async
3. Set up web server (Nginx/Apache) as reverse proxy
4. Configure database (PostgreSQL recommended)
5. Set `DEBUG = False` and `ALLOWED_HOSTS` properly
6. Collect static files (`python manage.py collectstatic`)
7. Use environment variables for secrets
8. Set up SSL certificates for HTTPS
9. Monitor logs and uptime

If you want, I can help you start a deployment step-by-step or revisit any topic in more detail!