

## CH1: Building a Blog Application

### The source code

<https://github.com/PacktPublishing/Django-5-by-example/tree/main/Chapter01>

This command used to install all python package included in requirements file

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

### Installing Python

- Django 5.0 supports Python 3.10, 3.11, and 3.12.
- Type command prompt CMD or PowerShell into the search box then open it
- To check Python 3 installed in OS type the command  

```
python --version
```
- In Windows, python is the Python executable of your default Python installation, whereas py is the Python launcher.

The differences between python and py:

#### python

- Refers to the Python interpreter/executable from the default Python installation on your system.
- When you run python in the terminal, Windows will look for the Python executable in your system's PATH.
- If you have multiple versions installed and PATH is not properly set, running python might cause confusion by pointing to the wrong version or giving a "command not found" error if not configured.

## py

- The Python Launcher for Windows is a tool designed to simplify managing multiple versions of Python.
- Installed by default with Python (since version 3.3) on Windows.

The Python launcher for Windows is a utility which aids in locating and executing of different Python versions. It allows scripts (or the command-line) to indicate a preference for a specific Python version, and will locate and execute that version.

```
> python --version
```

```
Python 3.8.10
```

```
> py --version
```

```
Python 3.11.2
```

```
> py -3.9 --version
```

```
Python 3.9.7
```

To display all python launcher in OS , type the follow command in cmd or PowerShell:

```
py --list
```

## Why Use py?

- Version management: Easily switch between versions.
- Reduced path issues: Even if python isn't properly configured in PATH, py can still find and run it correctly.
- If you have multiple versions, py ensures the right version runs without needing manual PATH adjustments.

## Creating a Python virtual environment

Since version 3.3, Python comes with the venv library, which provides support for creating lightweight virtual environments. By using the Python venv module to create isolated Python environments, you can use different package versions for different projects. Another advantage of using venv is that you won't need any administrative privileges to install Python packages.

The command to create virtual environment :

```
python -m venv venv
```

```
py -m venv my_env
```

Any Python libraries you install while your virtual environment is active will go into the my\_env/lib/python3.12/site-packages directory

Run the following command to activate your virtual environment:

```
Linux or macOS: >> source my_env/bin/activate
```

```
Windows: >> .\my_env\Scripts\activate
```

```
from powerShell >>> venv\Scripts\Activate.ps1
```

If you encounter a permissions error when trying to activate the venv, you might need to change the execution policy. You can do this by running the following command in PowerShell (make sure to run it as an administrator):

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

You can deactivate your environment at any time with the deactivate command.

## Installing Django

### Installing Django with pip

The pip package management system is the preferred method of installing Django.

Run the following command at the shell prompt to install Django with pip:

```
python -m pip install Django~=5.0.4
```

To check the Django has been installed , run the following command :

```
python -m django --version
```

## Django overview

- Django is a framework consisting of a set of components that solve common web development problems.
- Django components are loosely coupled, which means they can be managed independently.
- The database layer knows nothing about how the data is displayed, the template system knows nothing about web requests
- Django offers maximum code reusability by following the DRY (don't repeat yourself) principle.
- Django also fosters rapid development and allows you to use less code by taking advantage of Python's dynamic capabilities, such as introspection

**Introspection** allow developers to examine and modify the properties of objects at runtime. This feature is essential for creating flexible and adaptive programs.

**Introspection** refers to the ability of a program to examine the type or properties of an object at runtime. In Python, this can be done using several built-in functions and methods.

## Main framework components

Django follows the MTV (Model-Template-View) pattern. It is a slightly similar pattern to the well known MVC (Model-View-Controller) pattern, where the template acts as the view and the framework itself acts as the controller.

**The responsibilities in the Django MTV pattern are divided as follows:**

- **Model:** This defines the logical data structure and is the data handler between the database and the view.
- **Template:** This is the presentation layer. Django uses a plain-text template system that keeps everything that the browser renders.
- **View:** This communicates with the database via the model and transfers the data to the template for viewing.

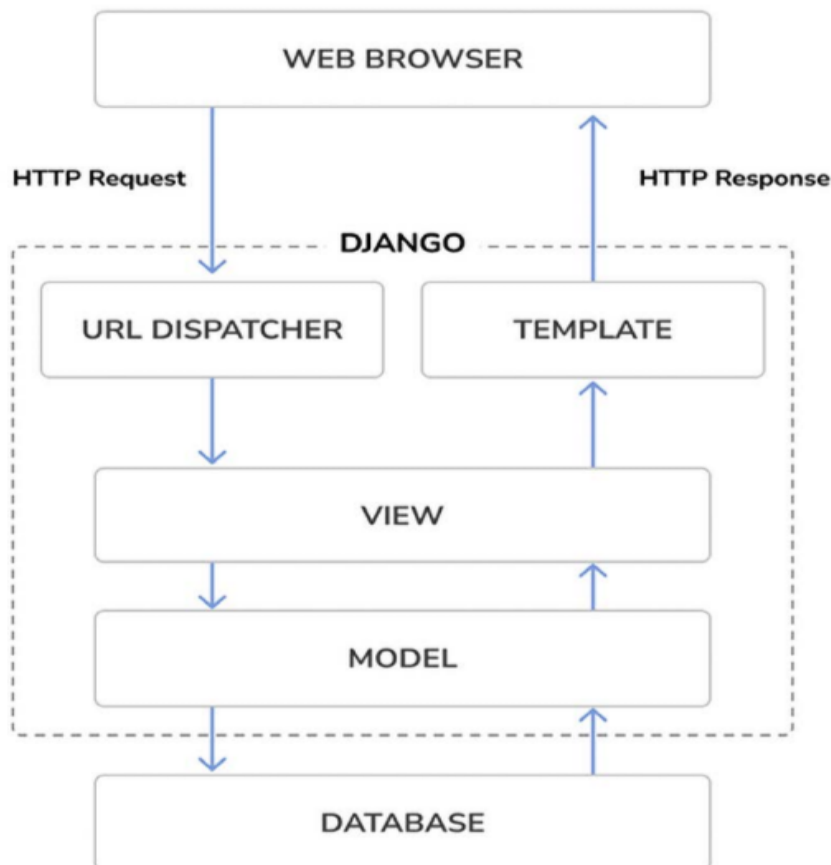
The framework itself acts as the controller. It sends a request to the appropriate view, according to the Django URL configuration.

## The Django architecture

This is how Django handles HTTP requests and generates response::

1. A web browser requests a page by its URL and the web server passes the HTTP request to Django.
2. Django runs through its configured URL patterns and stops at the first one that matches the requested URL.
3. Django executes the view that corresponds to the matched URL pattern.

4. The view potentially uses data models to retrieve information from the database.
5. Data models provide data definitions and behaviors. They are used to query the database.
6. The view renders a template (usually HTML) to display the data and returns it with an HTTP response.



Django also includes hooks in the request/response process, which are called **middleware**.

In Django, **middleware** is a powerful feature that allows developers to hook into the request/response processing cycle. Middleware is essentially a framework of hooks into Django's request/response processing.

## What is Middleware?

Middleware is a way to process requests globally before they reach the view (or after the view has processed them). It can be used for various tasks, such as:

**Request Processing:** Modifying the request before it reaches the view.

**Response Processing:** Modifying the response before it's sent to the client.

**Exception Handling:** Handling exceptions that may occur during the request/response cycle.

**Session Management:** Managing user sessions.

**Cross-Site Request Forgery (CSRF) Protection:** Validating requests to protect against CSRF attacks.

## New features in Django 5

Django 5.0 presents the following new major features:

### **Facet filters in the administration site:**

Admin pages now support facet filters, which display facet counts next to filter options.

### **Simplified templates for form field rendering:**

Django allows defining field groups in templates, making it easier to render related elements (e.g., labels, widgets, errors).

Form field rendering has been simplified with the capability to define field groups with associated templates.

### **Database-computed default values:**

Django adds database-computed default values.

You can define default values that are computed directly in the database. Offloads some logic to the database, ensuring consistency in default field values.

### **Database-generated model fields:**

This is a new type of field that enables you to create database-generated columns. An expression is used to automatically set the field value each time the model is changed.

Introduces GENERATED ALWAYS SQL syntax for fields, allowing the database to compute field values upon model changes.

### **More options for declaring model field choices:**

Fields that support choices no longer require accessing the choices attribute to access enumeration types. A mapping or callable instead of an iterable can be used directly to expand enumeration types.

Fields that support choices now accept mappings or callables instead of only iterables.

### **Creating your first project**

Blogging is the perfect starting point to build a complete Django project.

The blog application will consist of a list of posts including the post title, publishing date, author, a post excerpt, and a link to read the post.

Django provides a command that allows to create an initial project file structure.

```
django-admin startproject mysite
```

**Note:** Avoid naming projects after built-in Python or Django modules in order to prevent conflicts



```
mysite/
  manage.py
  mysite/
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
```

The outer mysite/ directory is the container for our project. It contains the following files:

- manage.py: This is a command-line utility used to interact with project.
- mysite/: This is the Python package for your project, which consists of the following files:
  - ★ \_\_init\_\_.py: An empty file that tells Python to treat the mysite directory as a Python module.
  - ★ asgi.py: This is the configuration to run project as an ASGI application with ASGI-compatible web servers.

ASGI (Asynchronous Server Gateway Interface) is a specification for building asynchronous web servers and applications in Python

ASGI is the emerging Python standard for asynchronous web servers and applications.

ASGI extends WSGI to support asynchronous communication protocols like WebSockets alongside standard HTTP. This makes it suitable for applications requiring real-time features, such as chat systems or live updates.

Asynchronous Server Gateway Interface (ASGI) support was first introduced in Django 3 and improved in Django 4.1 with asynchronous handlers for class-based views and an asynchronous ORM interface.

Django 5 adds asynchronous functions to the authentication framework, provides support for asynchronous signal dispatching, and adds asynchronous support to multiple built-in decorators.

### **Key Features:**

- a. Asynchronous Support: Enables concurrency and non-blocking I/O.
- b. WebSocket Support: Handles WebSocket connections.
- c. Backward Compatibility: Works with WSGI applications.
- d. Middleware: Similar to WSGI, it supports middleware for request/response processing.

### **ASGI Frameworks**

1. FastAPI: Great for building APIs with automatic OpenAPI documentation.
2. Django Channels: Extends Django to handle WebSockets and background tasks.
3. Starlette: Lightweight ASGI framework used by FastAPI.

- ★ settings.py: This indicates settings and configuration for project and contains initial default settings.
- ★ urls.py: This is the place where your URL patterns live. Each URL defined here is mapped to a view.
- ★ wsgi.py: This is the configuration to run project as a Web Server Gateway Interface (WSGI) application with WSGI-compatible web servers

A wsgi.py file is commonly used in Python web applications to serve as the entry point for WSGI (Web Server Gateway Interface) applications.

The wsgi.py file plays a crucial role in deploying Python web applications. It acts as the bridge between the web server and the Python application.

#### **When to Use wsgi.py:**

a. Deployment: It's used when deploying your app to servers like Gunicorn, uWSGI, or Apache (with mod\_wsgi).

b. Separation of Concerns: It ensures the server and application logic are decoupled. The server interacts only with the WSGI interface without knowing the internals of the application.

## **Applying initial database migrations**

Django applications require a database to store data.

The settings.py file contains the database configuration for project in the DATABASES setting.

The default configuration is a SQLite3 database. SQLite comes bundled with Python 3 and can be used in any of Python applications.

SQLite is a lightweight database that you can use with Django for development.

settings.py file includes a list named INSTALLED\_APPS that contains common Django applications that are added to project by default.

Django applications contain data models that are mapped to database tables.

**Open the shell prompt and run the following commands:**

```
cd mysite
```

```
python manage.py migrate
```

```
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying admin.0003_logentry_add_action_flag_choices... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying auth.0012_alter_user_first_name_max_length... OK
Applying sessions.0001_initial... OK
```

This database migrations are applied by Django by applying the initial migrations

## Running the development server

Django comes with a lightweight web server to run code quickly, without needing to spend time configuring a production server.

Start the development server by typing the following command in the shell prompt:

`python manage.py runserver`

```
Watching for file changes with StatReloader
Performing system checks...
System check identified no issues (0 silenced).
January 01, 2024 - 10:00:00
Django version 5.0, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

open **http://127.0.0.1:8000/** in browser. the page state the project is successfully running

You can run the Django development server on a custom host and port or tell Django to load a specific settings file.

```
python manage.py runserver 127.0.0.1:8001 --settings=mysite.settings
```

This server is only intended for development and is not suitable for production use. To deploy Django in a production environment, you should run it as a WSGI application using a web server, such as Apache, Gunicorn, or uWSGI, or as an ASGI application using a server such as Daphne or Uvicorn.

## Project settings

Some of the project settings:

- **DEBUG** is a Boolean that turns the debug mode of the project on and off.

If debug is set to True, Django will display detailed error pages when an uncaught exception is thrown by application.

When moving to a production environment, you have to set it to False.

**Note** : Never deploy a site into production with DEBUG turned on because will expose sensitive project-related data.

- **ALLOWED\_HOSTS** is not applied while debug mode is on or when the tests are run. Once you move site to production and set DEBUG to False, you will have to add domain/host to this setting to allow it to serve your Django site

the ALLOWED\_HOSTS setting acts as a security measure, ensuring that only specific domains or IP addresses can serve your site.

Once you set DEBUG=False (as required in production), Django enforces the ALLOWED\_HOSTS setting. This means you must explicitly list the domain(s) that are permitted to serve your site. Otherwise, Django will raise an Invalid HTTP\_HOST header error.

```
# settings.py
```

```
DEBUG = False
```

```
ALLOWED_HOSTS = ['yourdomain.com', 'www.yourdomain.com']
```

### Why ALLOWED\_HOSTS is not applied in development:

- **DEBUG mode:** When DEBUG=True, Django's security checks are relaxed for convenience during development, which includes not enforcing ALLOWED\_HOSTS.
  - **Tests:** When running tests, Django bypasses ALLOWED\_HOSTS to avoid unnecessary restrictions.
- **INSTALLED\_APPS** is a setting will have to edit for all projects. This setting tells Django which applications are active for this site.

Django includes the following applications:

- ★ **django.contrib.admin:** An administration site.
  - ★ **django.contrib.auth:** An authentication framework.
  - ★ **django.contrib.contenttypes:** A framework for handling content types.
  - ★ **django.contrib.sessions:** A session framework
  - ★ **django.contrib.messages:** A messaging framework.
  - ★ **django.contrib.staticfiles:** A framework for managing static files, such as CSS, JavaScript files, and images.
- **MIDDLEWARE** is a list that contains middleware to be executed.
  - **ROOT\_URLCONF** indicates the Python module where the root URL patterns of application are defined.
  - **DATABASES** is a dictionary that contains the settings for all the databases to be used in the project.

There must always be a default database. The default configuration uses a SQLite3 database.

- **LANGUAGE\_CODE** defines the default language code for Django site.
- **USE\_TZ** tells Django to activate/deactivate timezone support. Django comes with support for timezone-aware datetimes.

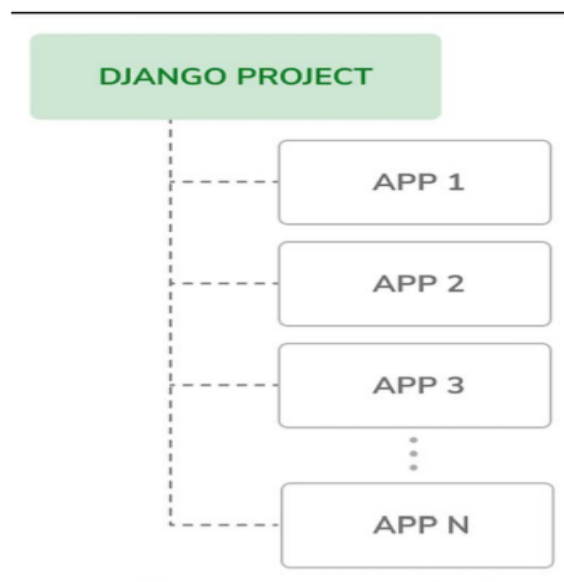
This setting is set to True when you create a new project using the startproject management command.

## Projects and applications

In Django, a project is considered a Django installation with some settings. An application is a group of models, views, templates, and URLs.

Applications interact with the framework to provide specific functionalities and may be reused in various projects.

Project contains several applications, such as a blog, wiki, or forum.



## Creating an application

building a blog application from scratch

The following command to build blog application in the shell prompt.

```
python manage.py startapp blog
```

```
blog/  
    __init__.py  
    admin.py  
    apps.py  
    migrations/  
        __init__.py  
    models.py  
    tests.py  
    views.py
```

- **\_\_init\_\_.py**: This is an empty file that tells Python to treat the blog directory as a Python module.
- **admin.py**: This is where you register models to include them in the Django administration site—using this site is optional.
- **apps.py**: This includes the main configuration of the blog application.
- **migrations**: This directory will contain database migrations of the application. Migrations allow Django to track model changes and synchronize the database. This directory contains an empty `__init__.py` file.
- **models.py**: This includes the data models of application; all Django applications need to have a `models.py` file but it can be left empty.
- **tests.py**: This is where you can add tests for application.
- **views.py**: The logic of application here; each view receives an HTTP request, processes it, and returns a response.

## Creating the blog data models

- ★ Python object is a collection of data and methods.
- Classes are the blueprint for bundling data and functionality



together. Creating a new class creates a new type of object, allowing to create instances of that type.

- A Django model is a source of information about the behaviors of data.
- Model consists of a Python class that subclasses `django.db.models.Model`.
- Each model maps to a single database table, where each attribute of the class represents a database field.
- Generating the database migrations for the models to create the corresponding database tables.
- When applying the migrations, Django will create a table for each model defined in the `models.py` file of the application.

## Creating the Post model

Define a Post model that will allow to store blog posts in the database. Add lines to the `models.py` file of the blog application.

```
from django.db import models

class Post(models.Model):

    title = models.CharField(max_length=250)

    slug = models.SlugField(max_length=250)

    body = models.TextField()

    def __str__(self):

        return self.title
```

Let's see the fields of the model

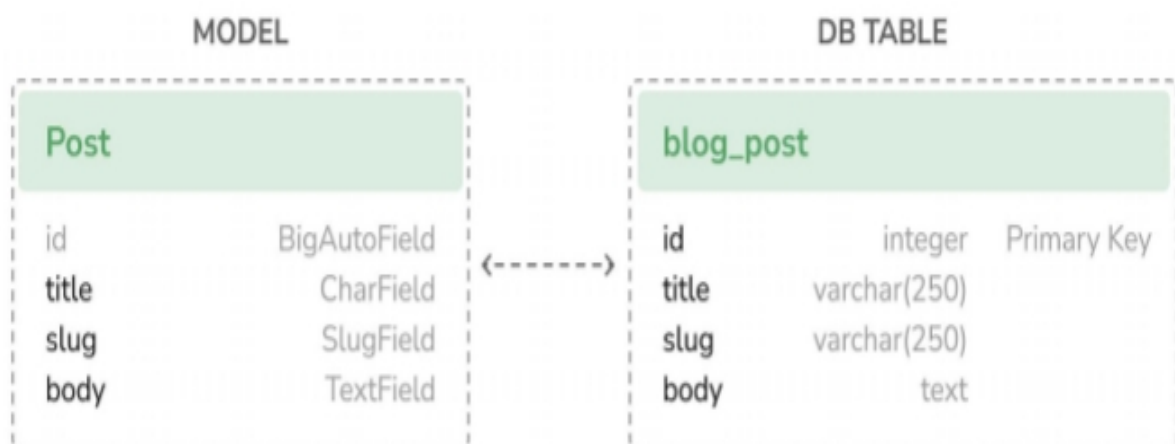
- **title:** This is the field for the post title. This is a CharField field that translates into a VARCHAR column in the SQL database.

- **slug:** This is a SlugField field that translates into a VARCHAR column in the SQL database.

A slug is a short label that contains only letters, numbers, underscores, or hyphens.

- **body:** This is the field for storing the body of the post. This is a TextField field that translates into a TEXT column in the SQL database.
- A `__str__()` method is the default Python method to return a string with the human-readable representation of the object. This used to display the name of the object in many places, such as the Django administration site.

**How the model and its fields will be translated into a database table and columns.**



Django will create a database column for each of the model fields: title, slug, and body.

By default, Django adds an auto-incrementing primary key field to each model. The field type is specified in each application configuration or globally in the `DEFAULT_AUTO_FIELD` setting.

When creating an application with the startapp command, the default value for DEFAULT\_AUTO\_FIELD is BigAutoField.

BigAutoField is a 64-bit integer that automatically increments according to available IDs. If you don't specify a primary key for model, Django adds this field automatically.

Any field can be determined by the model as a primary key by setting primary\_key=True on it.

## Adding datetime fields

Each post will be published at a specific date and time. Therefore, we need a field to store the publication date and time. We want to store the date and time when the Post object was created and when it was last modified.

```
from django.db import models
```

```
from django.utils import timezone
```

```
class Post(models.Model):
```

```
    .....
```

```
    publish = models.DateTimeField(default=timezone.now)
```

```
    created = models.DateTimeField(auto_now_add=True)
```

```
    updated = models.DateTimeField(auto_now=True)
```

- ★ publish : This is a DateTimeField field that translates into a DATETIME column in the SQL database. This used to store the date and time when the post was published.
- ★ created: This is a DateTimeField field. this used to store the date and time when the post was created. By using auto\_now\_add, the date will be saved automatically when creating an object.

- ★ updated: This is a DateTimeField field. This used to store the last date and time when the post was updated. By using auto\_now, the date will be updated automatically when saving an object.

we imported the timezone module to use timezone.now method that returns the current datetime in a timezone-aware format.

Another method to define default values for model fields is using database-computed default values. This feature allows to use underlying database functions to generate default values.

```
from django.db import models

from django.db.models.functions import Now

class Post(models.Model):

    #####

    publish = models.DateTimeField(db_default=Now())
```

To use database-generated default values, we use the db\_default attribute instead of default.

### **Note :**

Utilizing the auto\_now\_add and auto\_now datetime fields in your Django models is highly beneficial for tracking the creation and last modification times of objects.

## **Defining a default sort order**

Blog posts are presented in reverse chronological order, showing the newest posts first.

This ordering takes effect when retrieving objects from the database unless a specific order is indicated in the query.

```
from django.db import models

from django.utils import timezone

class Post(models.Model):
```

```
#####

updated = models.DateTimeField(auto_now=True)

class Meta:

    ordering = ['-publish']
```

Meta class inside model defines metadata for the model. using the ordering attribute to tell Django that it should sort results by the publish field. This ordering will apply by default for database queries when no specific order is provided in the query.

To indicating descending order by using a hyphen before the field name -publish. Posts will be returned in reverse chronological order by default.

## Adding a database index

define a database index for the publish field. This will improve performance for query filtering or ordering results by this field.

```
from django.db import models

class Post(models.Model):

    #####

    updated = models.DateTimeField(auto_now=True)

    class Meta:

        ordering = ['-publish']

        indexes = [models.Index(fields=['-publish']), ]
```

indexes option allows to define database indexes for model, which could comprise one or multiple fields in ascending or descending order, or functional expressions.

The hyphen before the field name to define the index in descending order.

**Note :** Index ordering is not supported on MySQL. If you use MySQL for the database, a descending index will be created as a normal index.

Defining indexes in Django models can significantly enhance the performance of database queries. By strategically indexing fields that are frequently queried, filtered, or sorted.

## Activating the application

To activate the blog application in the project for Django to keep track of the application and be able to create database tables for models.

Edit the settings.py file and add blog.apps.BlogConfig to the INSTALLED\_APPS setting.

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'blog.apps.BlogConfig',  
]
```

The BlogConfig class is the application configuration. Django knows that the application is active for project and will be able to load the application models.

## Adding a status field

A common functionality for blogs is to save posts as a draft until ready for publication. add a status field to model that will allow to manage the status of blog posts.

```

from django.db import models

from django.utils import timezone

class Post(models.Model):

    class Status(models.TextChoices):

        DRAFT = 'DF', 'Draft'

        PUBLISHED = 'PB', 'Published'

    #####

    updated = models.DateTimeField(auto_now=True)

    status = models.CharField(max_length=2,choices=Status, default=Status.DRAFT )

```

We have defined the enumeration class Status by subclassing models.TextChoices. The available choices for the post status are DRAFT and PUBLISHED.

The values are DF and PB, and labels or readable names are Draft and Published.

Django provides enumeration types that you can subclass to define choices simply.

We can access Post.Status.choices to obtain the available choices, Post.Status.names to obtain the names of the choices, Post.Status.labels to obtain the human-readable names, and Post.Status.values to obtain the actual values of the choices.

We have added a new status field to the model that is an instance of CharField. It includes a choices parameter to limit the value of the field to the choices in Status.

Run the following command in the shell prompt to open the Python shell:

```
python manage.py shell
```

Then, type the following lines:

```
>>> from blog.models import Post
>>> Post.Status.choices
```

You will obtain the enum choices with value-label pairs, like this:

```
[('DF', 'Draft'), ('PB', 'Published')]
```

Type the following line:

```
>>> Post.Status.labels
```

You will get the human-readable names of the enum members, as follows:

```
['Draft', 'Published']
```

Type the following line:

```
>>> Post.Status.values
```

You will get the values of the enum members, as follows. These are the values that can be stored in the database for the status field:

```
['DF', 'PB']
```

Type the following line:

```
>>> Post.Status.names
```

You will get the names of the choices, like this:

```
['DRAFT', 'PUBLISHED']
```

You can access a specific lookup enumeration member with `Post.Status.PUBLISHED` and you can access its `.name` and `.value` properties as well.

## Adding a many-to-one relationship

Posts are always written by an author.

We create a relationship between users and posts that will indicate which user wrote which posts.



Django comes with an authentication framework that handles user accounts. The Django authentication framework comes in the `django.contrib.auth` package and contains a User model.

To define the relationship between users and posts, we will use the **AUTH\_USER\_MODEL** setting which points to `auth.User` by default.

```
from django.conf import settings

from django.db import models

class Post(models.Model):

    #####

    author= models.ForeignKey( settings.AUTH_USER_MODEL,on_delete=models.CASCADE,
    related_name='blog_posts' )
```

import the project's settings and add an author field to the Post model. This field defines a many-to-one relationship with the default user model, meaning that each post is written by a user, and a user can write any number of posts.

Django will create a foreign key in the database using the primary key of the related model.

The `on_delete` parameter specifies the behavior to adopt when the referenced object is deleted. This is not specific to Django; it is a SQL standard.

Using **CASCADE**, you specify that when the referenced user is deleted, the database will delete all related blog posts.

we use `related_name` to specify the name of the reverse relationship from User to Post. This will allow to access related objects easily from a user object by using the `user.blog_posts` notation.

Using the `related_name` attribute in Django models is a great way to establish clear and intuitive reverse relationships between models. This allows to easily access related objects without having to query the database again.

`related_name='blog_posts'`: This attribute is added to the user field in the Post model, which establishes a reverse relationship. When you define `related_name`, it allows you to access all posts associated with a user instance using `user_instance.blog_posts.all()`.

## Creating and applying migrations

we need to create the database table for blog data model . Django comes with a migration system that tracks the changes made to models and enables them to propagate into the database.

The migrate command applies migrations for all applications listed in INSTALLED\_APPS. It synchro-nizes the database with the current models and existing migrations.

To create an initial migration for Post model. we run the following command in the shell prompt from the root directory of project:

```
python manage.py makemigrations blog
```

```
Migrations for 'blog':
  blog/migrations/0001_initial.py
    - Create model Post
    - Create index blog_post_publish_bb7600_idx on field(s)
      -publish of model post
```

Django created the 0001\_initial.py file inside the migrations directory of the blog application.

This migration contains the SQL statements to create the database table for the Post model and the definition of the database index for the publish field.

The sqlmigrate command takes the migration names and returns their SQL without executing it.

```
python manage.py sqlmigrate blog 0001
```

'blog': This specifies the name of your app (in this case, 'blog').

'0001': This is the name of the migration file you want to inspect. The first migration is usually named '0001\_initial.py' if you haven't added any other migrations yet.

The output should look as follows:

```
BEGIN;
--
-- Create model Post
--
CREATE TABLE "blog_post" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" varchar(250) NOT NULL,
    "slug" varchar(250) NOT NULL,
    "body" text NOT NULL,
    "publish" datetime NOT NULL,
    "created" datetime NOT NULL,
    "updated" datetime NOT NULL,
    "status" varchar(10) NOT NULL,
    "author_id" integer NOT NULL REFERENCES "auth_user" ("id") DEFERRABLE
INITIALLY DEFERRED);
--
-- Create blog_post_publish_bb7600_idx on field(s) -publish of model post
--
CREATE INDEX "blog_post_publish_bb7600_idx" ON "blog_post" ("publish" DESC);
CREATE INDEX "blog_post_slug_b95473f2" ON "blog_post" ("slug");
CREATE INDEX "blog_post_author_id_dd7a8485" ON "blog_post" ("author_id");
COMMIT;
```

The `sqlmigrate` command in Django is a useful tool that allows to see the SQL code that Django will execute for a specific migration without actually running the migration.

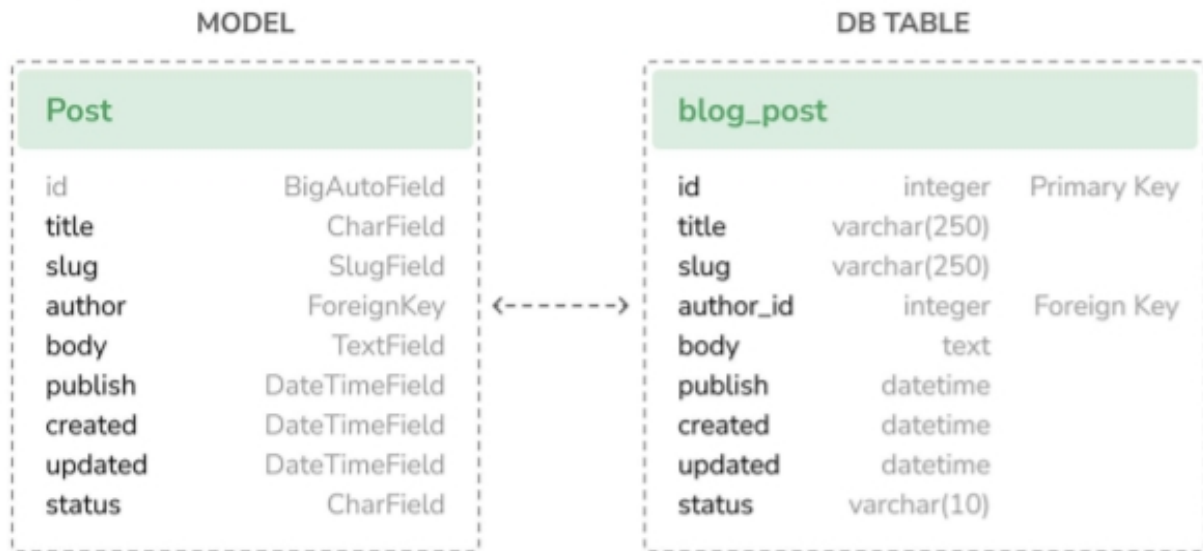
Django generates the table names by combining the application name and the lowercase name of the model (blog\_post), but you can specify a custom database name for model in the Meta class of the model using the db\_table attribute.

Django creates an auto-incremental id column that is used as the primary key for each model, but you can override this by specifying primary\_key=True on one of model fields.

The default id column consists of an integer that is incremented automatically. This column corresponds to the id field that is automatically added to model.

**There three database indexes are created:**

- An index in descending order on the publish column. This is the index with the indexes option of the model's Meta class.
- An index on the slug column because SlugField fields imply an index by default.
- An index on the author\_id column because ForeignKey fields imply an index by default.



To synchronize the model fields with the database columns, we apply the following command to apply the existing migrations.

```
python manage.py migrate
```

If we edit the models.py file to add, remove or change the fields of existing models, or if we add new models, we will have to create a new migration using the makemigrations command. Each migration allows Django to keep track of model changes. Then, we will have to apply the migration using the migrate command to keep the database in sync with models.

## Creating an administration site for models

we create a simple administration site to manage blog posts.

Django comes with a built-in administration interface that is very useful for editing content.

The Django site is built dynamically by reading the model metadata and providing a production-ready interface for editing content.

The django.contrib.admin application is already included in the INSTALLED\_APPS setting.

## Creating a superuser

To create a user to manage the administration site. Run the following command:

```
python manage.py createsuperuser
```

You will see the following output. Enter your desired username, email, and password, as follows:

```
Username (leave blank to use 'admin'): admin
Email address: admin@admin.com
Password: *****
Password (again): *****
```

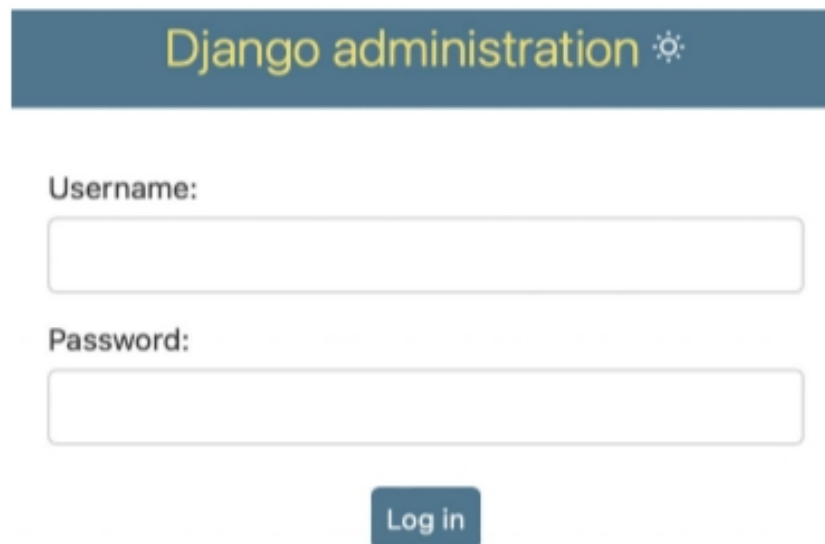
and then will appear message means superuser created successfully.

## The Django administration site

Start the development server with the following command:

```
python manage.py runserver
```

To access the administration login page , we open <http://127.0.0.1/admin/> in the browser.



Log in with credentials for superuser created to open administration site.

## Django administration

### Site administration

#### AUTHENTICATION AND AUTHORIZATION

**Groups** [+ Add](#) [Change](#)

**Users** [+ Add](#) [Change](#)

The Group and User models are part of the Django authentication framework located in `django.contrib.auth`.

To know the users on the site, click on **Users**.

### Adding models to the administration site

To add blog models to the administration site, we edit the `admin.py` file of the blog application and register it in the file.

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

```
from .models import Post
```

```
admin.site.register(Post)
```

### Site administration

#### AUTHENTICATION AND AUTHORIZATION

**Groups** [+ Add](#) [Change](#)

**Users** [+ Add](#) [Change](#)






#### BLOG

**Posts** [+ Add](#) [Change](#)

To add a new post , you click on the Add link beside Posts.

Django uses different form widgets for each type of field. Even complex fields, such as DateTimeField, are displayed with an easy interface such as a JavaScript date picker.

### Add post

Title:	<input type="text"/>
Slug:	<input type="text"/>
Author:	<div><div>----- ▾</div><div>  </div></div>
Body:	<div><div></div></div>
Publish:	<div><div>Date: <input type="text" value="2024-01-01"/> Today   </div><div>Time: <input type="text" value="23:35:13"/> Now   </div><div>Note: You are 2 hours ahead of server time.</div></div>
Status:	<div><div>Draft ▾</div></div>

SAVE

Save and add another

Save and continue editing

## Customizing how models are displayed

To add customization for administration site for blog application, edit admin.py file

The `@admin.register()` decorator performs the same function as the `admin.site.register()` function that replaced, registering the `ModelAdmin` class that it decorates.

```
from django.contrib import admin

from .models import Post

@admin.register(Post)

class PostAdmin(admin.ModelAdmin):

    list_display = ['title', 'slug', 'author', 'publish', 'status']
```

The `list_display` attribute allows to set the fields of model that want to display on the administration object list page.

```
from django.contrib import admin

from .models import Post

@admin.register(Post)

class PostAdmin(admin.ModelAdmin):

    list_display = ['title', 'slug', 'author', 'publish', 'status']

    list_filter = ['status', 'created', 'publish', 'author']

    search_fields = ['title', 'body']

    prepopulated_fields = {'slug': ('title',)}

    raw_id_fields = ['author']

    date_hierarchy = 'publish'

    ordering = ['status', 'publish']
```

The fields displayed on the post list page are the ones specified in the **list\_display** attribute. The list page includes a right sidebar that allows to filter the results by the fields included in the **list\_filter** attribute.



## Note :

Filters for ForeignKey fields like author are only displayed in the sidebar if more than one object exists in the database.

A search bar has appeared on the page. This is defined a list of searchable fields using the **search\_fields** attribute below the search bar.

There are navigation links to navigate through a date hierarchy defined by the **date\_hierarchy** attribute.

The posts ordered by STATUS and PUBLISH columns by default that specified using the **ordering** attribute.

When type the title of a new post, the slug field is filled in automatically using the **prepopulated\_fields** attribute.

The author field is displayed with a lookup widget by using the **raw\_id\_fields** attribute.

-----

**Author:**



## Adding facet counts to filters

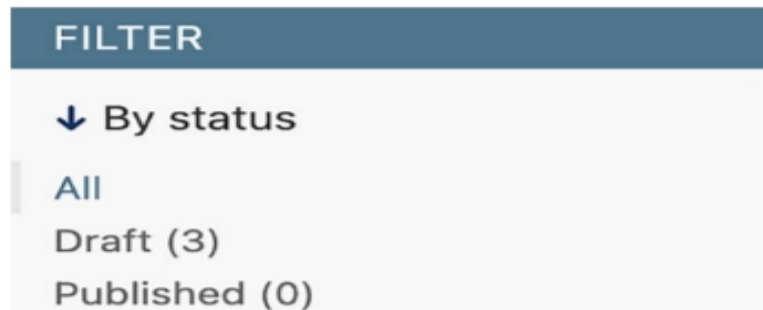
Django introduces facet filters to the administration site, showcasing facet counts. This counts indicate the number of objects corresponding to each specific filter.

```
@admin.register(Post)
```

```
class PostAdmin(admin.ModelAdmin):
```

```
    #####
```

```
    show_facets = admin.ShowFacets.ALWAYS
```



## Working with QuerySets and managers

The Django object-relational mapper (ORM) is a powerful database abstraction API that lets create, retrieve, update, and delete objects easily.

ORM allows to generate SQL queries using the object-oriented paradigm of Python. This is considered a way to interact with database in a Pythonic fashion instead of writing raw SQL queries.

The ORM maps models to database tables and provides with a simple Pythonic interface to interact with database. The ORM generates SQL queries and maps the results to model objects.

The Django ORM is compatible with MySQL, PostgreSQL, SQLite, Oracle, and MariaDB.

You can define the database of project in the DATABASES setting of project's settings.py file.

Django can work with multiple databases at a time, and you can program database routers to create custom data routing schemes.

The Django ORM is based on QuerySets. A QuerySet is a collection of database queries to retrieve objects from database.

The QuerySet equates to a SELECT SQL statement and the filters are limiting SQL clauses such as WHERE or LIMIT.

## Creating objects

To open the Python shell , run the command: `python manage.py shell`

```
>>>from django.contrib.auth.models import User
>>>from blog.models import Post
>>>user = User.objects.get(username='admin')
>>>post = Post(title='Another post', slug='another-post', body='Post body.', author=user)
>>>post.save()
```

Analysis the previous code :

```
>>> user = User.objects.get(username='admin')
```

we retrieve the user object with the username admin.

The `get()` method allows to retrieve a single object from the database. This method executes a SELECT SQL statement.

The `get()` method is commonly used in Object-Relational Mapping (ORM) frameworks (like Django or SQLAlchemy) to fetch a single object from the database.

If no results are returned by the database, this method will raise a `DoesNotExist` exception, and if the database returns more than one result, it will raise a `MultipleObjectsReturned` exception. Both exceptions are attributes of the model class that the query is being performed on.

```
>>>post = Post(title='Another post', slug='another-post', body='Post body.', author=user)
```

**DoesNotExist:** This exception is raised when no records are found. It's an indication that the object you were trying to retrieve does not exist in the database.

**MultipleObjectsReturned:** This exception is raised when the query finds more than one record that matches the provided criteria. This indicates a need for more specific filtering.

we create a Post instance with a custom title, slug, body and author.

This object is in memory and not persisted to the database that can be used during runtime but is not saved into the database.

```
>>>post.save()
```

we are saving the Post object in the database using the save() method and This action performs an INSERT SQL statement.

Previously we created an object in memory and then saved it in the database, but the object can be created and kept in the database in one operation using the Create() method.

```
>>> Post.objects.create(title='One more post', slug='one-more-post', body='Post body.',  
author=user)
```

Maybe need to fetch an object from the database or create it if it's absent. This get\_or\_create() method facilitates by either retrieving an object or creating it if not found.

The `get_or_create()` method in Django is a very useful function that allows to either retrieve an existing object from the database or create a new one if it does not exist. This method is helpful for ensuring that duplicate entries are avoided while also simplifying the code.

Example:

```
from django.contrib.auth.models import User  
  
# Attempt to retrieve a User with the username 'user2'  
  
user, created = User.objects.get_or_create(username='user2', defaults={'first_name':  
'John', 'last_name': 'Doe', 'email': 'john.doe@example.com'})  
  
if created:  
    print("A new user was created:", user)  
  
else:  
    print("User already exists:", user)
```

This method returns a tuple with the object retrieved and a Boolean indicating whether a new object was created.

```
>>> user, created = User.objects.get_or_create(username='user2')
```

Using the Django shell is a great way to test and interact with models and database. The `get_or_create()` method simplifies the process of ensuring a specific object exists, handling both retrieval and creation seamlessly.

## Updating objects

change the title of the Post object to something different and save the object, the save() method performs an UPDATE SQL statement.

```
>>> post.title = 'New title'
>>> post.save()
```

### Note:

The changes you make to a model object are not persisted to the database until you call the save() method.

## Retrieving objects

Each Django model has at least one manager, and the default manager is called objects.

To retrieve all objects from a table, we use the all() method on the default objects manager

```
>>> all_posts = Post.objects.all()
```

Note this QuerySet has not been executed yet. Django QuerySets are lazy, which means they are only evaluated when they are forced to.

If you don't assign the QuerySet to a variable, the SQL statement of the QuerySet is executed.

```
>>> Post.objects.all()
<QuerySet [<Post: Who was Django Reinhardt?>, <Post: New title>]>
```

## Filtering objects

To filter a QuerySet, we use the filter() method of the manager. This method allows to specify the content of a SQL WHERE clause by using field lookups.

you can use the filter Post objects by their title:

```
>>> Post.objects.filter(title='Who was Django Reinhardt?')
```

This QuerySet will return all posts with the exact title Who was Django Reinhardt?.

```
>>> posts = Post.objects.filter(title='Who was Django Reinhardt?')
>>> print(posts.query)
```

we can get the SQL produced by printing the query attribute of the QuerySet.

```
SELECT "blog_post"."id", "blog_post"."title", "blog_post"."slug", "blog_post"."author_id",
"blog_post"."body", "blog_post"."publish", "blog_post"."created", "blog_post"."updated",
"blog_post"."status" FROM "blog_post" WHERE "blog_post"."title" = Who was Django
Reinhardt? ORDER BY "blog_post"."publish" DESC
```

The generated WHERE clause performs an exact match on the title column. The ORDER BY clause specifies the default order defined in the ordering attribute of the Post model's Meta options since we haven't provided any specific ordering in the QuerySet.

## Using field lookups

The QuerySet interface provides with multiple lookup types. Two underscores are used to define the lookup type, with the format field\_\_lookup. For example:

```
>>> Post.objects.filter(id__exact=1)
```

Note:

When no specific lookup type is provided, the lookup type is assumed to be exact. >>> Post.objects.filter(id=1)

You can generate a case-insensitive lookup with iexact:

```
>>> Post.objects.filter(title__iexact='who was django reinhardt?')
```

the `iexact` lookup is used for case-insensitive exact matching in queries.

The contains lookup translates to a SQL lookup using the LIKE operator:

```
>>> Post.objects.filter(title__contains='Django')
```

```
>>> Post.objects.filter(title__icontains='django')
```

You can check for a given iterable (often a list, tuple, or another QuerySet object) with the in lookup.

```
>>> Post.objects.filter(id__in=[1, 3])
```

There is the greater than (gt) lookup that is equivalent to SQL clause is WHERE ID > 3. `>>> Post.objects.filter(id__gt=3)`

This example shows the greater than or equal to lookup:

```
>>> Post.objects.filter(id__gte=3)
```

There is the less than (lt) lookup that is equivalent to SQL clause is WHERE ID < 3. `>>> Post.objects.filter(id__lt=3)`

This example shows the less than or equal to lookup:

```
>>> Post.objects.filter(id__lte=3)
```

A case-sensitive/insensitive starts-with lookup can be performed with the startswith and istartswith lookup types, respectively:

```
>>> Post.objects.filter(title__startswith='who')
```

A case-sensitive/insensitive ends-with lookup can be performed with the endswith and iendswith lookup types, respectively:

```
>>> Post.objects.filter(title__iendswith='reinhardt?')
```

There are different lookup types for date lookups.

```
>>> from datetime import date
```

```
>>> Post.objects.filter(publish__date=date(2024, 1, 31))
```

This shows how to filter a DateField or DateTimeField field by year, by month, by day:

```
>>> Post.objects.filter(publish__year=2024)
```

```
>>> Post.objects.filter(publish__month=1)
```

```
>>> Post.objects.filter(publish__day=1)
```

You can chain additional lookups to date, year, month, and day. For example, here is a lookup for a value greater than a given date.

```
>>> Post.objects.filter(publish__date__gt=date(2024, 1, 1))
```

To lookup related object fields, you use the two-underscores notation. For example, to retrieve the posts written by the user with the admin username:

```
>>> Post.objects.filter(author__username='admin')
```

You can chain additional lookups for the related fields. For example, to retrieve posts written by any user with a username that starts with ad:

```
>>> Post.objects.filter(author__username__startswith='ad')
```

You can filter by multiple fields. For example:

```
>>> Post.objects.filter(publish__year=2024, author__username='admin')
```

## Chaining filters

You can chain QuerySets together.

```
>>> Post.objects.filter(publish__year=2024).filter(author__username='admin')
```

## Excluding objects

You can exclude certain results from QuerySet by using the `exclude()` method of the manager.

```
>>> Post.objects.filter(publish__year=2024).exclude(title__startswith='Why')
```

## Ordering objects

The default order is defined in the ordering option of the model's Meta.

You can override the default ordering using the `order_by()` method of the manager. 

```
>>> Post.objects.order_by('title')
```

Ascending order is implied. You can indicate descending order with a negative sign prefix. 

```
>>> Post.objects.order_by('-title')
```

You can order by multiple fields.

```
>>> Post.objects.order_by('author', 'title')
```

To order randomly, use the string '?', as follows:

```
>>> Post.objects.order_by('?')
```



## Limiting QuerySets

You can limit a QuerySet to a certain number of results by using a subset of Python's array-slicing syntax. `>>> Post.objects.all()[:5]`

Note The negative indexing is not supported.

To retrieve a single object, we can use an index instead of a slice.

```
>>> Post.objects.order_by('?')[0]
```

## Counting objects

The `count()` method counts the total number of objects matching the QuerySet and returns an integer.

This method translates to a `SELECT COUNT(*)` SQL statement.

```
>>> Post.objects.filter(id__lt=3).count()
```

## Checking if an object exists

The `exists()` method allows to check if a QuerySet contains any results.

This method returns `True` if the QuerySet contains any items and `False` otherwise.

```
>>> Post.objects.filter(title__startswith='Why').exists()
```

```
False
```

## Deleting objects

To delete an object, by using the `delete()` method from an object instance.

```
>>> post = Post.objects.get(id=1)
```

```
>>> post.delete()
```

Note that deleting objects will also delete any dependent relationships for `ForeignKey` objects defined with `on_delete` set to `CASCADE`.

## Complex lookups with Q objects

To build more complex queries, such as queries with OR statements, you can use Q objects. A Q object allows to encapsulate a collection of field lookups.

You can compose statements by combining Q objects with the & (and), | (or), and ^ (xor) operators.

```
>>> from django.db.models import Q
>>> starts_who = Q(title__startswith='who')
>>> starts_why = Q(title__startswith='why')
>>> Post.objects.filter(starts_who | starts_why)
```

## When QuerySets are evaluated

evaluation refers to the moment when the QuerySet is processed to retrieve data from the database.

Creating a QuerySet doesn't involve any database activity until it is evaluated.

When a QuerySet is evaluated, it translates into a SQL query to the database.

QuerySets are a powerful way to interact with the database, creating a QuerySet doesn't immediately hit the database; it remains unevaluated until certain actions prompt evaluation.

**QuerySets are only evaluated in the following cases:**

- The first time you iterate over them. (e.g., using a loop)
- When you slice them, for instance, `Post.objects.all()[:3]`
- When you pickle or cache them. (e.g., storing it in a session)
- When you call `repr()` or `len()` on them
- When you explicitly call `list()` on them
- When you test them in a statement, such as `bool()`, `or`, `and`, or `if`.

## Creating model managers

The default manager for every model is the objects manager. This manager retrieves all the objects in the database.

To create a custom manager to retrieve all posts that have a PUBLISHED status.

there are two main strategies for adding or customizing model managers to improve how can retrieve and filter data.

There are **two** ways to add or customize managers for models: you can **add extra manager methods** to an existing manager or **create a new manager** by modifying the initial QuerySet that the manager returns.

The first method provides with a QuerySet notation like **Post.objects.my\_manager()**, and the second method provides you with a QuerySet notation like **Post.my\_manager.all()**.

Here, we use the second method (create new manager)

```
class PublishedManager(models.Manager):  
    def get_queryset(self):  
        return (  
            super().get_queryset().filter(status=Post.Status.PUBLISHED)  
        )  
  
class Post(models.Model):  
    #####  
    objects = models.Manager() # The default manager.  
    published = PublishedManager() # Our custom manager.
```

The `get_queryset()` method of a manager returns the QuerySet that will be executed.

We have overridden this method to build a custom QuerySet that filters posts by their status and returns a successive QuerySet that only includes posts with the PUBLISHED status.

Start the development server with the following command in the shell prompt:

```
python manage.py shell
```

You can import the Post model and retrieve all published posts whose title starts with Who:

```
>>> from blog.models import Post
```

```
>>> Post.published.filter(title__startswith='Who')
```

### Method 1: Adding Extra Manager Methods to an Existing Manager

In this method, you can extend the default manager by adding additional methods to it. This allows to call these methods using the standard `Post.objects` notation.

#### Example

Suppose you have a `Post` model, and you want to add a method to retrieve published posts:

```
from django.db import models
```

```
# Adding a custom manager method
```

```
class PostManager(models.Manager):
```

```
    def published(self):
```

```
        return self.filter(status='PUBLISHED')
```

```
class Post(models.Model):
```

```
    title = models.CharField(max_length=200)
```

```
    content = models.TextField()
```

```
    status = models.CharField(max_length=10, choices=[('DRAFT', 'Draft'), ('PUBLISHED', 'Published')])
```

```
    def __str__(self):
```

```
        return self.title
```

```
# Attaching the custom manager to the model
```

```
objects = PostManager() # Use custom manager
```

You can now call the `published` method like this:

```
published_posts = Post.objects.published()
```

## Method 2: Creating a New Manager by Modifying the Initial QuerySet

This method involves creating a new manager that returns a custom QuerySet. This allows for more flexibility and cleaner chaining of methods.

### Example

Here's how to create a custom QuerySet and manager that allows retrieving published posts using `Post.published.all()` notation:

```
class PublishedPostQuerySet(models.QuerySet):
```

```
    def published(self):
```

```
        return self.filter(status='PUBLISHED')
```

```
class PublishedPostManager(models.Manager):
```

```
    def get_queryset(self):
```

```
        return PublishedPostQuerySet(self.model, using=self._db)
```

```
# Attaching the custom manager to the model
```

```
class Post(models.Model):
```

```
    # ... (fields here)
```

```
    published = PublishedPostManager() # New manager for published posts
```

you can use the custom manager like this:

```
published_posts = Post.published.all()
```

## Building list and detail views

A Django view is a Python function that receives a web request and returns a web response. All the logic to return the desired response goes inside the view.

First, you will create application views, then you will define a URL pattern for each view, and finally, you will create HTML templates to render the data generated by the views.

Each view will render a template, passing variables to it, and will return an HTTP response with the rendered output.

## Creating list and detail views

To display the list of posts, we will create view to it. Edit the views.py file of the blog application.

we retrieve all the posts with the PUBLISHED status using the published manager that we created.

```
from django.shortcuts import render

from .models import Post

def post_list(request):

    posts = Post.published.all()

    return render( request, 'blog/post/list.html', {'posts': posts} )
```

The post\_list view takes the request object as the only parameter. This parameter is required by all views.

we use the render() shortcut to render the list of posts with the given template.

**render()** function takes the **request object**, the **template path**, and the **context variables** to render the given template. It returns an HttpResponse object with the rendered text (normally HTML code).

The render() shortcut takes the request context into account, so any **variable set by the template context processors is accessible** by the given **template**.

**Template context processors** are just callables that set variables into the context.

Create a second view to display a single post. Add the function to the views.py file:

```

from django.http import Http404

def post_detail(request, id):
    try:
        post = Post.published.get(id=id)
    except Post.DoesNotExist:
        raise Http404("No Post found.")
    return render( request, 'blog/post/detail.html', {'post': post})

```

The `post_detail` view takes the `id` argument of a post.

In the view, we try to retrieve the `Post` object with the given `id` by calling the `get()` method on the `published` manager. We raise an `Http404` exception to return an HTTP 404 error if the `model DoesNotExist` exception is raised because no result is found.

we use the `render()` shortcut to render the retrieved post using a template.

```

from django.shortcuts import get_object_or_404, render

def post_detail(request, id)
    post = get_object_or_404( Post, id=id, status=Post.Status.PUBLISHED )
    return render( request, 'blog/post/detail.html', {'post': post} )

```

## Using the `get_object_or_404` shortcut

Django provides a shortcut to call `get()` on a given model manager and raises an `Http404` exception instead of a `DoesNotExist` exception when no object is found.

import the `get_object_or_404` shortcut and change the `post_detail` view.

```

from django.shortcuts import get_object_or_404, render

#####

def post_detail(request, id):
    post = get_object_or_404( Post, id=id,status=Post.Status.PUBLISHED )
    return render( request, 'blog/post/detail.html',{'post': post} )

```

we use the `get_object_or_404()` shortcut to retrieve the desired post.

`get_object_or_404()` function retrieves the object that matches the given parameters or an HTTP 404 (not found) exception if no object is found.

The '`get_object_or_404()`' shortcut is a convenient way to retrieve an object from the database. If the object does not exist, it raises a '`404 Not Found`' error instead of returning '`None`'.

The '`get_object_or_404()`' takes two main arguments:

1. A model class : This is the Django model from which you want to retrieve the object.

2. A query : This can be one or more keyword arguments that represent the fields and their values to filter the objects.( You can pass any number of field lookups (e.g., '`id=id`', '`slug=slug_value`') to filter the query.).

### Example with Multiple Filters

```
from django.shortcuts import render, get_object_or_404

from .models import Article

def article_detail(request, slug):

    # Fetch the article by its slug

    article = get_object_or_404(Article, slug=slug, published=True)

    return render(request, 'article_detail.html', {'article': article})
```

## Adding URL patterns for your views

URL patterns allow to map URLs to views. A URL pattern is composed of a string pattern, a view, and, optionally, a name that allows to name the URL project-wide.

Django runs through each URL pattern and stops at the first one that matches the requested URL.

Django imports the view of the matching URL pattern and executes it, passing an instance of the `HttpRequest` class and the keyword or positional arguments.



```

from django.urls import path

from . import views

app_name = 'blog'

urlpatterns = [

    # post views

    path("", views.post_list, name='post_list'),

    path('<int:id>/', views.post_detail, name='post_detail'),

]

```

application namespace is defined with the `app_name` variable. This allows to organize URLs by application and use the name when referring to them.

Organizing URLs by application is a common practice, especially in larger projects. Defining an application namespace using the ``app_name`` variable allows to reference URLs specific to an application, making your URL configuration cleaner and more manageable.

### Step 1: Define ``app_name`` in Your URLs

In the ``urls.py`` file of your Django app, you can define the ``app_name`` variable. This variable will serve as the namespace for your application's URLs.

```

# myapp/urls.py

from django.urls import path

from . import views

app_name = 'myapp' # Define the application namespace

urlpatterns = [

    path("", views.index, name='index'), # Home page

    path('detail/<int:id>/', views.detail, name='detail'), # Detail page for an item

]

```

## Step 2: Define URL Patterns

In the `urlpatterns` list, you can define the URL patterns using the `path()` function. Each pattern associates a URL path with a specific view.

1. Home Page: The first pattern defines the root URL of the app, which points to the `index` view.

2. Detail Page: The second pattern includes a dynamic segment `<int:id>`, allowing you to pass an integer `id` to the `detail` view.

## Step 3: Reference URLs in Templates or Views

When you want to refer to these URLs in your templates or views, you can use the namespace defined by `app_name`. This ensures that there is no conflict with URLs from other applications.

### #### In Templates

`<!-- In your HTML template -->`

```
<a href="{% url 'myapp:index' %}">Home</a>
```

```
<a href="{% url 'myapp:detail' id=some_id %}">Detail</a>
```

### #### In Views

```
from django.urls import reverse
```

```
from django.http import HttpResponseRedirect
```

```
def some_view(request):
```

```
    return HttpResponseRedirect(reverse('myapp:index')) # Redirect to the home page
```

You define two different patterns using the `path()` function. The first URL pattern doesn't take any arguments and is mapped to the `post_list` view. The second pattern is mapped to the `post_detail` view and takes only one argument `id`, which matches an integer, set by the path converter `int`.

we use angle brackets to capture the values from the URL. Any value specified in the URL pattern as `<parameter>` is captured as a string. we use path converters, such as `<int:year>` to match and return an integer.

URL patterns can include dynamic segments that capture values from the

URL. This is done using angle brackets (< >) and path converters.

## Capturing Values from the URL

### 1. Basic Capture:

You can capture a value from the URL by using angle brackets. For example, in the URL pattern `<parameter>`, anything that matches will be captured as a string.

```
path('blog/<slug:post>/', views.post_detail, name='post_detail')
```

In this example, if a user visits `/blog/my-first-post/`, the value `my-first-post` will be captured and passed to the `post_detail` view as the parameter `post`.

### 2. Path Converters:

Django provides several built-in path converters that allow you to define the type of data you expect from the URL. Here are some common converters:

Converter	Description
<code>&lt;int:year&gt;</code>	Captures a value and returns it as an integer.
<code>&lt;str:name&gt;</code>	Captures a value and returns it as a string (default behavior).
<code>&lt;slug:post&gt;</code>	Captures a slug, which can include letters, numbers, underscores, or hyphens.
<code>&lt;uuid:id&gt;</code>	Captures a UUID string and returns it as a UUID object.
<code>&lt;path:filename&gt;</code>	Captures a path, allowing for slashes in the value.

For example `<slug:post>` would specifically match a slug (a string that can only contain letters, numbers, underscores, or hyphens).

If using `path()` and converters isn't sufficient, you can use `re_path()` instead to define complex URL patterns with Python regular expressions.

## Example URL Patterns

```

from django.urls import path

from . import views

urlpatterns = [

    # Matches URLs like /blog/2024/ or /blog/2025/
    path('blog/<int:year>/', views.year_archive, name='year_archive'),

    # Matches URLs like /blog/my-first-post/
    path('blog/<slug:post>/', views.post_detail, name='post_detail'),

    # Matches URLs like /files/my_document.pdf
    path('files/<path:filename>/', views.file_download, name='file_download'),

    # Matches URLs like /user/123/ or /user/456/
    path('user/<int:id>/', views.user_profile, name='user_profile'),

]

```

**Note:** Creating a urls.py file for each application is the best way to make applications reusable by other projects.

Edit the urls.py file located in the mysite directory of project.

```

from django.contrib import admin

from django.urls import include, path

urlpatterns = [

    path('admin/', admin.site.urls),

    path('blog/', include('blog.urls', namespace='blog')),

]

```

The new URL pattern defined with include refers to the URL patterns defined in the blog application so that they are included under the blog/ path. You include these patterns under the namespace blog.

The `path()` function provides a simple and readable way to define URL patterns using converters. you can use the `re_path()` function that allows to define URL patterns using Python's regular expressions.

The `re_path()` function is particularly useful when you need to match URLs that have specific requirements or patterns that cannot be easily defined using the simpler `path()` function.

```
from django.urls import re_path

urlpatterns = [

    re_path(r'^your-regex-pattern/$', views.your_view, name='your_name'),

]
```

### Example Patterns with `re_path()`

Here's how you might use `re_path()` to define more complex URL patterns:

### 1. Matching Numeric IDs:

- If you want to capture a numeric ID that may also have leading zeros:

```
from django.urls import re_path

from . import views

urlpatterns = [

    re_path(r'^blog/(?P<id>\d{1,5})/$', views.blog_detail, name='blog_detail'), # Matches
IDs from 1 to 99999

]
```

### 2. Capturing Slugs with Optional Parts:

- If you want to match a URL that may have an optional section after a slug:

```
urlpatterns = [

    re_path(r'^blog/(?P<slug>[-\w]+)/(?P<page>\d+)?/$', views.blog_page,
name='blog_page'), # Optional page parameter

]
```

### 3. Matching Date Patterns:

- If you want to capture a date in a specific format (e.g., YYYY-MM-DD):

```
urlpatterns = [
    re_path(r'^events/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/$',
views.event_detail, name='event_detail'),
]
```

Namespaces have to be unique across entire project. Later, you will refer to blog URLs easily by using the namespace followed by a colon and the URL name, for example, `blog:post_list` and `blog:post_detail`.

## Creating templates for your views

Templates define how the data is displayed; they are usually written in HTML in combination with the Django template language.

we add templates to application to display posts in a user-friendly manner. Create the following directories and files inside blog application directory.

```
templates/
  blog/
    base.html
    post/
      list.html
      detail.html
```

The `base.html` file include the main HTML structure of the website and divide the content into the main content area and a sidebar.

The `list.html` and `detail.html` files will inherit from the `base.html` file to render the blog post list and detail views.

Django has a powerful template language that allows to specify how data is displayed. It is based on **template tags**, **template variables**, and **template filters**.

- ★ Template tags control the rendering of the template and look like this: `{% tag %}`.
- ★ Template variables get replaced with values when the template is rendered and look like this: `{{ variable }}`.

- ★ Template filters allow you to modify variables for display and look like this:  
`{{ variable|filter }}`.

## Creating a base template

Edit the base.html file.

```
{% load static %}

<!DOCTYPE html>

<html>

    <head>

        <title>{% block title %}{% endblock %}</title>

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

    </head>

    <body>

        <div id="content">

            {% block content %}

            {% endblock %}

        </div>

        <div id="sidebar">

            <h2>My blog</h2>

            <p>This is my blog.</p>

        </div>

    </body>

</html>
```

`{% load static %}` tells Django to load the static template tags that are provided by the `django.contrib.staticfiles` application, which is contained in the `INSTALLED_APPS` setting. After loading them, you can use the `{% static %}` template tag throughout this template.

There are two `{% block %}` tags. These tell Django that you want to define a block in area. Templates that inherit from this template can fill in the blocks with content.

## Creating the post list template

edit the `post/list.html` file

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

{% block title %}My Blog{% endblock %}

{% block content %}

    <h1>My Blog</h1>

    {% for post in posts %}

        <h2>

            <a href="{% url 'blog:post_detail' post.id %}">

                {{ post.title }}

            </a>

        </h2>

        <p class="date">

            Published {{ post.publish }} by {{ post.author }}

        </p>

        {{ post.body|truncatewords:30|linebreaks }}

    {% endfor %}

{% endblock %}
```

With the `{% extends %}` template tag, you tell Django to inherit from the `blog/base.html` template.

You fill the title and content blocks of the base template with content. You iterate through the posts and display their title, date, author, and body, including a link in the title to the detail URL of the post.



This template tag allows to build URLs dynamically by their name. We use `blog:post_detail` to refer to the `post_detail` URL in the `blog` namespace. We pass the required `post.id` parameter to build the URL for each post.

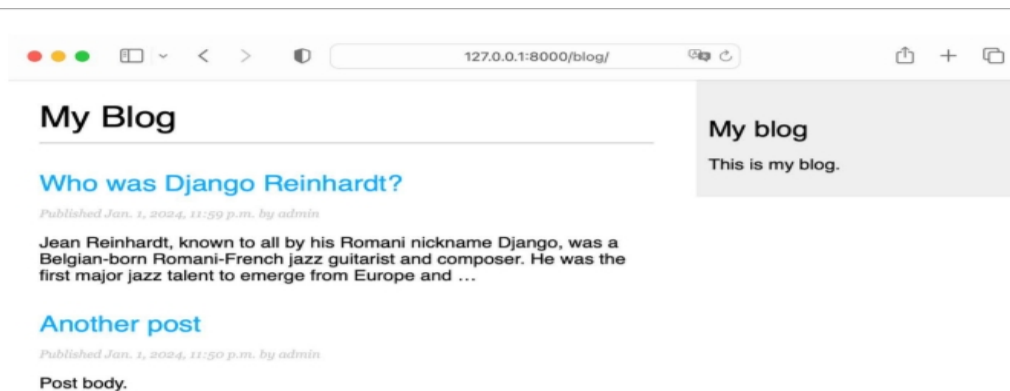
### Note:

Always use the `{% url %}` template tag to build URLs in templates instead of writing hardcoded URLs. This will make your URLs more maintainable.

In the body of the post, we apply two template filters: `truncatewords` truncates the value to the number of words specified, and `linebreaks` converts the output into HTML line breaks. You can concatenate as many template filters as you wish; each one will be applied to the output generated.

## Accessing our application

We open shell prompt to start the development server, then open `http://127.0.0.1:8000/blog/` in browser. `python manage.py runserver`



## Creating the post detail template

edit the `post/detail.html` file.

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

{% block title %}{{ post.title }}{% endblock %}

{% block content %}

    <h1>{{ post.title }}</h1>

    <p class="date">
```

Published {{ post.publish }} by {{ post.author }}

</p>

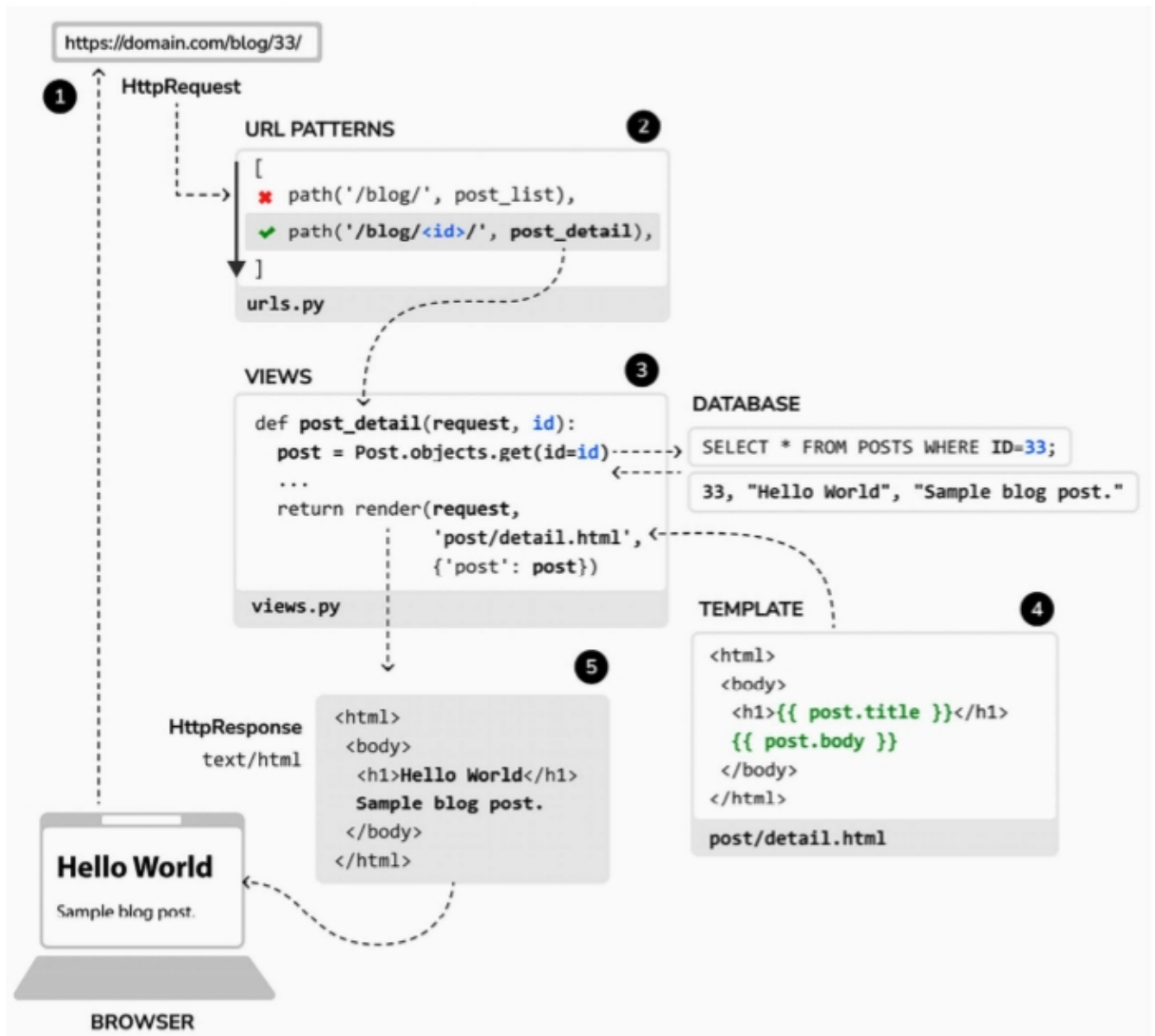
{{ post.body|linebreaks }}

{% endblock %}



## The request/response cycle

The following schema shows a simplified example of how Django processes HTTP requests and generates HTTP responses:



- A web browser requests a page by its URL, for example, `https://domain.com/blog/33/`. The web server receives the HTTP request and passes it over to Django.

- Django runs through each URL pattern defined in the URL patterns configuration. The framework checks each pattern against the given URL path, and stops at the first one that matches the requested URL.
- Django imports the view of the matching URL pattern and executes it, passing an instance of the HttpRequest class and the keyword or positional arguments. The view uses the models to retrieve information from the database. Using the Django ORM, QuerySets are translated into SQL and executed in the database.
- The view uses the render() function to render an HTML template passing the Post object as a context variable.
- The rendered content is returned as a HttpResponse object by the view with the text/html content type by default.