DISTILLED BEGINNER'S GUIDE TO DJANGO

By Kabaki Antony

Distilled Beginner’s Guide to Django

Prelude

Welcome to the world of Django! If you're an absolute beginner to web development with Django, you've come to the right place. This book is designed to be your guide, breaking down the most important concepts you need to understand to start building websites using Django quickly and efficiently.

Django is a powerful web framework written in Python that enables you to build dynamic and robust web applications with ease. Whether you're looking to create a simple blog, an e-commerce app, or a complex social media platform, Django provides you with the tools and flexibility to bring your ideas to life.

In this book, we'll walk you through the fundamental concepts of Django, from installation and project setup to building a fully functional web application. Each chapter will focus on a specific aspect of Django development, providing clear explanations, practical examples, and step-by-step instructions to help you grasp the concepts and put them into practice.

By the end of this book, you'll have a solid understanding of Django's core principles and be well-equipped to start building your own web applications with confidence. So, let's dive in and embark on this exciting journey into the world of Django!

Introduction

Welcome to the Distilled Beginner's Guide to Django! In this booklet, we'll explore Django, a powerful web framework written in Python, and its significance in web development. Whether you're completely new to web development or looking to expand your skills, Django offers a robust platform for building dynamic and scalable web applications.

Throughout this guide, we'll take a hands-on approach to learning Django by building a blogging project from scratch. We'll start with the basics and gradually introduce more advanced concepts, all while incrementally building our blogging application. By following along with this project, you'll gain practical experience and insight into how Django can be used to create real-world web applications.

Overview of Django

Django is more than just a web framework – it's a toolkit that empowers developers to create sophisticated websites and web applications with ease. From handling database interactions to managing user authentication, Django streamlines the development process, allowing you to focus on building great features rather than reinventing the wheel.

Topics covered

In this booklet, we'll cover the essential concepts that every beginner needs to understand to start building websites using Django. We'll break down complex topics into simple, easy-to-understand explanations and provide practical examples to help you get started quickly. Here's a brief overview of the topics we'll cover:

1. **Installation and Setup**: Learn how to install Django and set up your development environment.
2. **Apps and models**: Learn how to create apps in Django and how models tie to apps.
3. **Views**: Explore views, what they are, how they handle user requests and generate responses.
4. **Templates**: Learn about Django's template system for building dynamic web pages.
5. **URLs and Routing**: Learn how to map URLs to views using Django's URL routing system.
6. **Forms**: Explore Django's form-handling capabilities for collecting and validating user input.
7. **Authentication and Authorization**: Learn how to implement user authentication and authorization in Django.
8. **Static Files:** Understand what static files are in Django and how Django handles them in development and in production.
9. **Admin Interface**: Learn how to use Django's built-in admin interface for managing site content.

10. **Deployment**: Explore options for deploying Django applications to production servers.

Throughout this booklet, we'll provide clear explanations, practical examples, and step-by-step instructions to help you master Django development. By the end, you'll have the knowledge and confidence to start building web applications with Django. Let's dive in!

Installation and Setup

In this chapter, we'll walk through the process of installing Django, setting up a new Django project, and exploring the project structure. By the end of this chapter, you'll have everything you need to start building your Django applications.

Create a project directory

First, let's create a directory to contain all of our Django project files. Navigate to the desired location on your system and create a directory named **distilled_django_blog**, you can give it any name you want as long as it is descriptive and does not crash with Django's inbuilt methods and modules. Here is how you can do it:

```
$ mkdir distilled_django_blog
$ cd distilled_django_blog
```

The first command **mkdir (make directory)** creates a directory and the second one, **cd (change directory)** gets into the directory or rather you want to change from the directory that you are in currently to the new one that you have created.

Create a virtual environment

Upon creation of the directory, we are going to create a virtual environment. A virtual environment is an isolated environment where you can work on Python projects without affecting the global Python installation. As you become more experienced in Django development you may have more than one Django project that you are working on, each project will have its dependencies and probably might even have different versions of Django. To be able to work on them separately isolate them using a virtual environment. It is important to note that a virtual environment is not a mandatory requirement for a Django project or any Python project creation, but a recommended convention, due to the aforementioned reasons.

To create a virtual environment use the **venv** package that ships with Python:

```
$ python3 -m venv myenv
```

The above code block creates a virtual environment named **myenv,** to use it, you have to activate it:

Linux / Mac OS:

```
$ . myenv/bin/activate
```

Windows:

```
$ . myenv\Scripts\activate
```

Install Django

After activating the virtual environment, the next step will be to install Django:

```
$ python3 -m pip install django
```

The above command installs the latest version of Django, however, you can choose the version of Django you want to install by specifying a version:

```
$ python3 -m pip install django==4.2
```

To confirm that Django is installed, run the following command:

```
$ python3 -m django --version
```

If Django is installed, the version number will be printed in the console. If it is not, you'll get an error "No module named Django".

Assuming that Django has been correctly installed, this will avail various command line utilities and tools that will be used to manage your Django project. You will explore a number of them in the course of this project, the first of such utilities is **django-admin.** This command allows you to perform various administrative tasks for managing Django projects, with it you can start a Django project, create apps for the project, run the development server, and manage databases.

Create Django project

To begin building with Django, you have to take care of some initial setup. This is to auto-generate some code that scaffolds a Django project, the scaffold will include a settings instance of Django, database configuration, Django-specific options, and application-specific settings. Django does the heavy lifting and does not want you to worry about project structure but instead worry about building.

Therefore, to begin building a Django project we will use the **django-admin** command that is used in combination with other commands, one such command is the **startproject** command, this command creates the initial Django project scaffold.

Create a scaffold for the blog project using the following command:

```
$ django-admin startproject django_blog .
```

This command **`django-admin startproject django_blog .`** creates a new Django project named **`django_blog`** in the current directory (denoted by the **`.`**). Instead of creating a new sub-directory for the project, this command places all the necessary Django project files - such as **`manage.py`**, **`settings.py`**, **`urls.py`**, **`asgi.py`**, and **`wsgi.py`** - directly in the existing directory. This is useful when you want to set up a Django project without adding an extra level of directory nesting, keeping the project files neatly organized within the current folder.

Upon creation of the scaffold, the project will have a directory structure similar to the one below:

```
distilled_django_blog/
├── myenv/                  # Virtual environment directory
│   ├── bin/
│   ├── include/
│   ├── lib/
│   └── ...                 # Other virtual environment files and
folders/directories
├── django_blog/            # Django project directory
│   ├── __init__.py         # Marks this directory as a Python package
│   ├── settings.py         # Project settings
│   ├── urls.py             # URL declarations for the project
│   ├── asgi.py             # ASGI configuration
│   ├── wsgi.py             # WSGI configuration
│   └── __pycache__/        # Cache files (created after running the
project)
├── manage.py               # Django's management script
```

Let’s break down what each of the files and directories means in relation to the project:

`distilled_django_blog/`: The root directory containing the entire project setup.

- **`myenv/:`** The virtual environment directory (**`myenv`**), containing the Python environment and packages for your project.
- **`django_blog/`**: The Django project directory containing the core project files. In a way you can refer to it as the main Django project.
    - **`__init__.py`**: Marks this directory as a Python package.
    - **`settings.py`**: Contains the project's settings, such as database configuration, installed apps, and middleware.
    - **`urls.py`:** Contains the URL routing declarations for the project.
    - **`asgi.py`**: ASGI configuration for the project, used for asynchronous applications.
    - **`wsgi.py`**: WSGI configuration for the project, used for deploying the project on a web server.
    - **`__pycache__/`**: This directory will appear after running the project, as Python caches bytecode files.
- **`manage.py`**: A command-line utility for interacting with the Django project (such as running the server, making migrations, etc.) This file is a wrapper around the **django-admin** utility that we used to create the project scaffold.

Take note of the following:

To make a scaffold for this project, we used the **django-admin** `command` together with the **`startproject`** command because we didn't have a Django project yet. However, once the project was created, we will switch to using **`manage.py`** and will continue using **`manage.py`** for all other commands going forward.

The reason for this switch is that **`manage.py`** is a wrapper around **`django-admin`** that is specifically tied to your Django project. It's project context-aware, meaning it understands the settings and environment of the current project, ensuring that commands are executed correctly within that context. On the other hand, **`django-admin`** is a more general command-line utility that is not tied to any specific project and lacks the project-specific context that **`manage.py`** provides.

In summary, once your Django project is set up, it's best to use **`manage.py`** for running commands because it is aware of your project's configuration and ensures everything runs smoothly within that environment. Therefore, going forward we will use **manage.py <command>** to carry out all the tasks that need commands in our projects**.**

With the above files and project scaffold in place, you're now ready to start building your blogging application. In the next section, we'll dive into the concepts of apps and models, exploring how they contribute to the core functionality and data structure of your Django project.

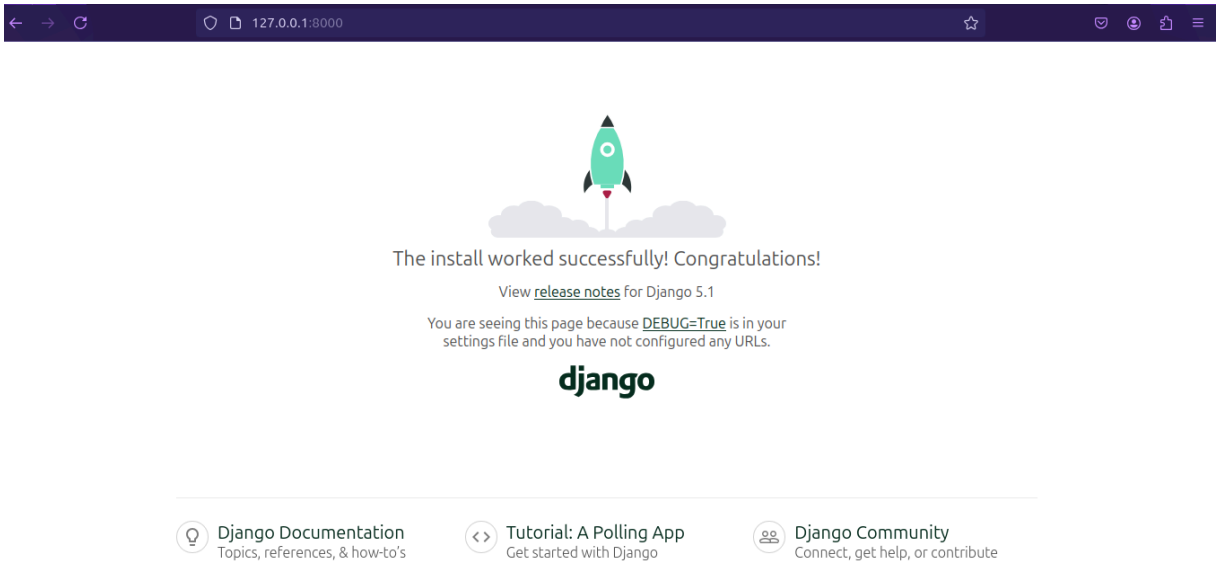
Running the development server

The initial scaffold project provides a runnable application, allowing us to verify that Django is installed correctly and the initial setup is successful. Django comes with a built-in development server, which simplifies the development process by handling hosting during the early stages of development. This means you can quickly test features without worrying about configuring a production server. To run the project, first activate your virtual environment (if it's not already active), and then execute the following command:

```
python manage.py runserver or ./manage.py runserver
```

This will launch the development server and make the project accessible in your browser, in the following link http://127.0.0.1:8000 .

If you open the link to your browser and see the below image then Django has been installed successfully.



The output of the **runserver** command will display a link to access the project on the development server. You might also see a red warning about unapplied migrations. Don't worry—this is a common message when you start a Django project. We'll cover what migrations are in detail in the next section, along with how to address this warning and apply the necessary migrations.

```
Watching for file changes with StatReloader
Performing system checks...
```

```
System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly
until you apply the migrations for app(s): admin, auth, contenttypes,
sessions.
Run 'python manage.py migrate' to apply them.
September 24, 2024 - 08:12:08
Django version 5.1.1, using settings 'dj_test_a.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

[24/Sep/2024 08:12:14] "GET / HTTP/1.1" 200 1206
```

Apps and Models

In this section, we're diving into one of the core components of Django: apps and models. Understanding these concepts is crucial because they form the backbone of any Django project.

Apps in Django are like independent modules that handle specific functionalities within your project. Whether it's managing blog posts, handling user authentication, or processing payments, each app in Django is designed to focus on a particular aspect of your application. This means that a Django project is typically composed of a collection of individual apps, each responsible for a specific functionality. When combined, these apps work together to form the complete application.

Django's philosophy regarding apps emphasizes modularity and reusability. By separating different functionalities into distinct apps, Django promotes a clean and organized codebase, making it easier to understand, maintain, and debug. Each app is self-contained and can be reused across different projects, which aligns with Django's broader ecosystem of third-party apps. This modular approach also enhances scalability, as individual apps can be optimized or replaced without disrupting the entire project. By focusing on specific functions, Django apps allow for tailored solutions, easier collaboration among developers, and the ability to extend your project's functionality as needed.

Models are the heart of these apps. They define the data structures in your application, representing database tables through Python classes. By creating models, you essentially design the database schema, telling Django how to store and manage the data that powers your application. Therefore, generally every app will have its own models.

Create an App

To create an app in Django, use the **`manage.py`** command combined with the **`startapp`** utility. This command sets up a new app within your project, creating the necessary files and directories to start building specific app:

```
$ python3 manage.py startapp blog or ./manage.py startapp blog
```

The above code block creates a new app in your project named **`blog`**. We've chosen this name because it is descriptive of all the functionality related to blogging in our project. This app will handle everything from managing posts and comments to other blog-related features, making it a central part of our Django blog application.

After creating an app, you need to inform Django of its existence by adding it to the **INSTALLED_APPS** setting. This ensures that Django recognizes the app and loads its models, views, and other configurations when running the project. To do this, add the new app to the **`INSTALLED_APPS`** `setting` in your project's settings. Therefore, open the project **`settings.py`** file, located in the (i.e **django_blog/settings.py**) directory, and add **`'blog'`** to the **`INSTALLED_APPS`** setting list like this:

```
# django_blog/settings.py
INSTALLED_APPS = [
# other installed apps
'blog'
]
```

By adding your app to the **`INSTALLED_APPS`** setting, you're instructing Django to recognize it as part of the project. This allows Django to include the app when running commands, checking for models, loading static files, and performing other necessary operations for the app's functionality within the project.

Create Models

Models are the backbone of any Django application, acting as the blueprint for your data structure. In Django, a model is a Python class that subclasses **django.db.models.Model.** It defines the fields for your data—for example, in our blog model, we will have fields for the blog **title** that will be the title for the post, **created_at** to store the date and time the post was created, and **content** to store the blog's actual content. Additionally, models define the behaviors of your data, such as default values or how fields interact with other models through relationships. Each model corresponds to a single database table, and each attribute (field) within the model translates to a column in the database.

Creating models in Django is a straightforward process, yet it's incredibly powerful. You'll define various fields in your model - like **`CharField`** for short text, **`TextField`** for longer text, **`DateTimeField`** for dates and times, and so on, each will correspond to a different type of data. Django handles all the heavy lifting, automatically creating the necessary database tables and providing an intuitive API to interact with your data.

Up to this point you might be wondering why we are creating models and we have not configured any database for our blogging application, one of the advantages of Django is that it ships with **SQLite3** as its default database engine, thus it already has configuration to make it run. **SQLite3** is a lightweight, file-based database that requires no setup or configuration, making it ideal for development and testing. This means you can dive straight into building your models and experimenting with your data without needing to worry about setting up a separate database server. Once you're ready for production, Django makes it easy to switch

to more robust database systems like **PostgreSQL, MySQL,** or **Oracle**. For the purposes of this book we will use the **SQLite3** database.

In this section, we'll guide you through the process of creating models for your Django project. You'll learn how to:

- Define models to represent your data structure.
- Add fields to your models that define the type and behavior of your data.

By understanding and creating models, you'll be laying a solid foundation for your Django application, enabling you to manage and manipulate your data with ease. Let's dive in and start building the core components of your database with Django models!

Create blogs model

To keep things simple we will build just one model for the blogging application. Open the **blog/models.py** file and put in the following code in it:

```
from django.db import models

class Blogs(models.Model):
   title = models.CharField(max_length=200)
   content = models.TextField()
   created_at = models.DateTimeField(auto_now_add=True)

   def __str__(self):
       return self.title

   def snippet(self):
       return self.content[:500] + "  ..."
```

Let's explain what the above code does line by line:

Imports

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

This line imports Django's **models** module, which contains the base classes and fields you need to define your data models.

Model definition

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

This line defines a new class, **`Blogs`**, which inherits from the **`models.Model`** class. In Django, a model is a Python class that represents a table in the database, with each attribute corresponding to a column in that table. Typically, all models in Django inherit from the **`models.Model`** class, which provides the necessary functionality for interacting with the database.

Attributes

`title`:

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

This line defines a **title** attribute as a **`CharField`,** which is used for storing short to medium-length text. The **`max_length=200`** argument limits the length of the title to 200 characters.

`content`:

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

This line defines a **content** attribute as a **`TextField`**, which is used for storing large amounts of text. Unlike **`CharField`, `TextField`** doesn't have a **`max_length`** limit, making it suitable for storing the main content of a blog post.

`created_at`:

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

This line defines a **created_at** attribute as a **`DateTimeField`**, which stores date and time information. The **`auto_now_add=True`** argument ensures that this field is automatically set to the current date and time when a new blog post is created. It does not change once the object is created.

String Representation

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

This method defines the string representation of the **Blogs** object. When you print an instance of this model or view it in the Django admin, it will display the **title** of the blog post. This makes it easier to identify individual instances. Without this method, any representation of the model instance will not make any sense, however with it then you will be able to tell what the model instance represents.

Snippet

The **snippet** method is a custom method defined in your model that returns a shortened version of the **content** field:

```
def snippet(self):
      return self.content[:500] + "  ..."
```

Explanation:

- This method is designed to create a preview or summary of the blog post content.
- **self.content[:500]** takes the first 500 characters of the **content** field. **" ..."** adds an ellipsis (...) at the end to indicate that the content is truncated.
- When you call this **snippet** method on a blog post instance, it will return only the first 500 characters of the post's content, followed by an ellipsis, which can be useful for displaying a summary of the post on a blog listing page or elsewhere in the application.

This approach helps you present a concise preview of the blog content without displaying the entire text.

Summary

- This model defines a simple blog post structure with three attributes: **title,** **content,** and **created_at**.
- The **title** field stores the blog post's title, the **content** field stores the main text of the post, and the **created_at** field automatically records when the post was created.
- The **__str__** method ensures that each instance of the **Blogs** model is represented by its title when printed or displayed in the admin.
- The **snippet** method in the model returns the first 500 characters of the blog post's content followed by an ellipsis, providing a concise preview of the full text. This is useful for displaying summaries in places like a blog listing page.

This model provides a basic foundation for a blogging application, allowing you to create, store, and display blog posts. While this simple model serves as an introduction to creating models in Django, there are many more fields you can define to enrich your application. The fields we've included are sufficient for storing and displaying a basic blog post, but Django

offers a wide range of field types that you can explore and utilize as your project grows. For a comprehensive guide to all the possible fields and how to use them, I encourage you to visit the official Django documentation.

Understanding Migrations

Now that we've created our model, the next step is to apply these changes to our database. This is where Django's powerful migration system comes into play. Migrations are Django's way of propagating changes you make to your models (like adding a new field or creating a new model) into your database schema.

Migrations are essentially a version control system for your database schema, similar to how Git is a version control system for your code. They allow you to:

- Create tables and fields in your database that correspond to your models.
- Modify existing tables when you change your models.
- Keep track of changes to your database schema over time.

When you create or modify a model, Django doesn't immediately apply those changes to your database. Instead, it requires you to create a migration file, which is a record of the changes you've made to your models. You can then apply these migrations to your database, ensuring that your database schema matches your models.

In the next section, we'll walk through the process of creating and applying migrations, ensuring that your model is correctly reflected in the database.

Applying Migrations

After writing the code for your model, the next step is to apply these changes to your database. In Django, this is done in two steps using the commands: **`makemigrations`** and **`migrate`**.

`makemigrations:`

The **`makemigrations`** command scans your project's models and creates a migration file that details the changes needed in the database. It's essentially Django's way of preparing the database to match the structure of your models. This command generates the SQL instructions needed to update the database schema based on changes to your models (like adding, modifying, or deleting fields). However, **`makemigrations`** itself doesn't apply these changes—it simply prepares the migration files. To apply the changes to the database, you use the **`migrate`** command.

To create a migration file, run the following command in your terminal:

```
python3 manage.py makemigrations or ./manage.py makemigrations
```

This will generate a migration file in your app's **`migrations`** directory. The file will include instructions for creating the necessary tables and fields in the database.

migrate:

Once the migration file is created, the next step is to apply these migrations to your database. This updates the database schema to match your models.

Run the following command to apply the migrations:

```
python3 manage.py migrate or ./manage.py migrate
```

This will execute the SQL commands needed to update or create tables in your database, ensuring that your model's structure is correctly reflected.

By following these two steps, you'll ensure that your database is always in sync with your models, allowing you to effectively manage and update your application's data structure as it evolves. This is also the same structure and steps that are followed every time you want to add, update, remove a field from your models in Django.

Running migrations

At this point in our project, we've created a **`Blogs`** model, but it hasn't been applied to the database yet. Before we do that, let's first run the project. To do that activate the virtual environment as we discussed earlier and run the project using the following command:

```
python3 manage.py runserver or ./manage.py runserver
```

Upon running the project, take note of the output, which will likely indicate that there are 18 unapplied migrations. These unapplied migrations are database tables that Django has prepared as part of the initial setup for the project, such as **`contenttypes`**, **`auth`**, and others. Django provides pre-made migration files for these, so we only need to apply them by running:

```
python3 manage.py migrate or ./manage.py migrate
```

Once these default migrations are applied, we can then focus on our custom **`Blogs`** model. To do this, we'll run the two commands:

```
python3 manage.py makemigrations or ./manage.py makemigrations
```

This will create a migration file for our **`Blogs`** model.

```
python3 manage.py migrate or ./manage.py migrate
```

This will apply the migration, creating the **`Blogs`** table in the database.

By following these steps, we'll ensure that our project's database is fully set up and that the **`Blogs`** model is ready to store data.

Testing out the database API

After creating the database tables, it's crucial to test the database to ensure your models and interactions are functioning correctly. Once you've defined your model and applied the migrations, you can use Django's database API to interact with the database through your model. This API allows you to perform operations such as creating, retrieving, updating, and deleting records.

Here's how you can get started with testing the database API:

Accessing the Django Shell

Django provides an interactive shell where you can test out database operations in real time. To access the shell, activate your virtual environment (if it's not already active) and run the following command:

```
python3 manage.py shell  or ./manage.py shell
```

Importing the Model

Once you're in the Django shell, you'll need to import the model you want to work with. For our **`Blogs`** model, you would do this:

```
from blog.models import Blogs
```

Creating a New Record

Create a new blog post by instantiating the **`Blogs`** model and saving it to the database:

```
new_blog = Blogs(title="My First Blog Post", content="This is the content
```

```
of my first blog post.")
new_blog.save()
```

After running this, a new entry will be added to the **`Blogs`** table in your database.

Retrieving Records

To retrieve records from the database, you can use Django's query methods. For example, to retrieve all blog posts:

```
all_blogs = Blogs.objects.all()
```

Or to retrieve a specific blog post by its ID:

```
blog_post = Blogs.objects.get(id=1)
```

Updating a Record

Update an existing blog post by first retrieving it and then modifying its attributes:

```
blog_post = Blogs.objects.get(id=1)
blog_post.title = "Updated Blog Post Title"
blog_post.save()
```

Deleting a Record

To delete a record, simply retrieve it and call the **`delete()`** method:

```
blog_post = Blogs.objects.get(id=1)
blog_post.delete()
```

Exploring More Queries

Django's ORM provides a rich set of query methods to filter, order, and aggregate data. Explore these methods to perform more complex queries, such as:

```
# Filtering by a specific condition
filtered_blogs = Blogs.objects.filter(title__contains="Blog")
```

Ordering results

```
ordered_blogs = Blogs.objects.order_by('-created_at')
```

By testing these operations in the shell, you can verify that your model and database are functioning correctly, giving you confidence as you continue to build out your Django application. While you can interact with the database API this way, it's not the most user-friendly approach. However, this quick demonstration is just to whet your appetite and give you a glimpse of the powerful ways you can use Django's database API. As you continue, you'll discover more intuitive and efficient methods for working with your data.

There's a lot more to Django models than what we've covered in this introduction. Models can be far more complex, with relationships like **One-to-One**, **Many-to-One**, and **Many-to-Many**, allowing you to structure your data in intricate ways. Additionally, models can include a wide variety of fields beyond what we've touched on here, enabling you to store and manage much more data than in our simple blog model example.

To get a feel for the various fields and relationships, I strongly encourage you to explore the official Django documentation. Diving into the documentation will give you a deeper understanding of Django's internals and empower you to build more sophisticated and powerful applications.

In the next section, we'll dive into views in Django and explore how they interact with models to present data to users. You'll learn how views process requests, retrieve the necessary data from your models, and render it in a way that's both meaningful and accessible to users. This will give you a solid understanding of how Django's MVT structure comes together to create dynamic, data-driven web applications.

Views, URLs and Routing

A view is a Python function or class that takes a web request and returns a web response. The response can be anything from an HTML page, a redirect, a 404 error, or even data in formats like JSON or XML. Essentially, views are where the logic of your application lives, deciding what data to retrieve and how to present it to the user.

In Django, views are an essential part of the framework, acting as the bridge between your models and what users see in their browsers. Views are responsible for receiving, handling HTTP requests and returning HTTP responses, making them the backbone of your application's functionality.

Django provides two main types of views:

Functional Views:

These are simple Python functions that take a **`request`** object and return a **`response`** object. Functional views are straightforward and easy to understand, making them great for simple use cases and quick development. In this guide we will use this kind of view.

Class-Based Views (CBVs):

These are more powerful and flexible than functional views. CBVs use Python classes to represent views, allowing you to leverage object-oriented programming features like inheritance, mixins, and encapsulation. With CBVs, you can handle different HTTP methods (GET, POST, etc.) in a more organized way, and reuse code more efficiently across your application.

The Role of Views in Handling HTTP Requests

When a user visits a URL on your website, Django matches that URL to a view function or class. The view then processes the request, which might involve querying the database, performing calculations, or validating input. Once the view has gathered or processed the necessary data, it constructs a response and sends it back to the user's browser.

Creating Views

To create a view in Django, you can start with a simple function or class. For example:

Functional View Example:

```
from django.http import HttpResponse

def my_view(request):
      return HttpResponse("Hello, World!")
```

Class-Based View Example:

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

class MyView(View):
      def get(self, request):
          return HttpResponse("Hello, World!")
```

In both examples, the view takes an HTTP request and returns an HTTP response. You can then link this view to a URL in your **`urls.py`** file, making it accessible to users.

Create Blog App Views

In this section, we'll create the initial views for our blog app. Views are a fundamental part of any Django application, as they manage the logic for processing user requests and generating appropriate responses. We'll begin by setting up two basic views, one to display all the blog posts and the other one to show the details of a particular blog post.

Note: We are not yet covering the creation, updating, and deletion of posts. These functionalities require forms and templates, which we will explore in later sections and update them as we get to respective sections. For now, the views will be limited to returning simple HTTP responses, such as showing the string representation of the object they reference.

Overall our blogging application will have the following views at the end.

1. **List View**: A view to display all the blog posts.
2. **Detail View**: A view to show the details of a single blog post.
3. **Create View**: A view to allow users to create a new blog post.
4. **Update View**: A view to edit an existing blog post.
5. **Delete View**: A view to delete a blog post.

We'll be using function-based views (FBVs) for the purposes of this application but once you get the gist of how they operate you will be able to change them into class based views.

Open the **blog/views.py** file and add in the following code:

```
from django.http import HttpResponse
from .models import Blogs
```

Here's a brief explanation of the imports:

`from django.http import HttpResponse`:

This imports the **`HttpResponse`** class, which is used to return simple HTTP responses from a view. It allows you to send text or HTML directly to the user's browser.

`from .models import Blogs`:

This imports the **`Blogs`** model from the current app's **`models.py`** file. The **`Blogs`** model represents a database table where blog-related data is stored.

blog_list View

```
def blog_list(request):
   posts = Blogs.objects.all()
   return HttpResponse(posts)
```

The **`blog_list`** function is a Django **view** that fetches all blog posts from the database and returns them in an HTTP response.

Here is the breakdown of what each line does:

`def blog_list(request):`

This defines a function-based view named **`blog_list`** that takes a **`request`** object as an argument. The **`request`** object contains information about the HTTP request made by the user (like GET or POST requests). Take note that all views take in a **request** object as their first parameter, this represents the current HTTP request or rather it contains the information about the current request.

`posts = Blogs.objects.all():`

This line queries the database to retrieve all the records in the **`Blogs`** model. **`Blogs.objects.all()`** returns a **QuerySet** containing all blog posts stored in the database and assigns them to the **posts** variable.

`return HttpResponse(posts):`

This sends the **posts** QuerySet directly as a response using **HttpResponse**. While this does return the titles of the blog posts (thanks to the **__str__** method defined in the model), it's not an ideal way to display the content. The raw QuerySet output is not user-friendly and lacks essential functionality like making the titles clickable or linking them to individual blog detail pages. Without these features, users would have to manually navigate to each post's details page. To create a better user experience, we'll introduce **templates** later on, which will allow us to present a more organized, interactive, and visually appealing blog list.

blog_detail view

```
def blog_detail(request, id):
   post = Blogs.objects.get(id=id)
   return HttpResponse(post)
```

The **blog_detail** function retrieves a single blog post based on the provided **id** and returns it as an **HttpResponse**. Here's a breakdown of what this code does:

def blog_detail(request, id):
This defines a view function **blog_detail** that handles HTTP requests. It takes two arguments:

- **request**: The current HTTP request object.
- **id:** The unique identifier of the blog post to retrieve.

post = Blogs.objects.get(id=id):

This line retrieves a specific blog post from the database where the **id** matches the one provided in the URL and assigns it to the **post** variable. If no object is found, it will raise an error (a **DoesNotExist** exception).

return HttpResponse(post):

This sends the blog post as an HTTP response to the client (the browser). Since the **__str__** method of the **Blogs** model is set to return the title of the post, this will display the title as plain text in the browser.

Just like with the b**log_list** view, this approach is very basic. When we get to the templates section, we'll replace this with a more robust and user-friendly way of rendering blog details. As mentioned earlier, there will be additional views in our application. However, we will develop them incrementally as we progress through the forms and templates sections.

Having worked on the two views, let's move on to see how we can tie them to URLs using Django's routing system in the next section on **URLs and Routing**. This will allow users to access our views through specific web addresses, making the application more interactive.

What are URLs and Routing?

URLs (Uniform Resource Locators) and routing are essential components in any web application, and in Django, they serve as the bridge between users and your application's views.

URL:

A URL is the web address that a user types into their browser to access a specific page or resource on your website. It defines the route or path to a particular view in your application, enabling users to navigate to different pages, such as a homepage, blog post, or contact form and so on.

For example, **`www.example.com/blog/`** is a URL that might direct users to a list of blog posts, while **`www.example.com/blog/5/`** could direct them to a specific blog post with an ID of 5.

Routing:

Routing is the process by which Django matches the URLs typed by the user to the appropriate views in your application. Django's URL routing system allows you to define patterns that map URLs to views. When a user visits a specific URL, Django's routing system looks through your URL configurations (**urls.py file**) to find a match, then calls the corresponding view to handle the request and generate a response.

For example, if a user visits **`/blog/5/`**, Django's routing system will match this URL to a corresponding view that retrieves and displays the blog post with the ID of 5.

In summary, **URLs** are the paths users navigate to, and **routing** is the system that connects those URLs to the appropriate views in your Django application.

How URLs and Routing Work

Django uses a URL dispatcher to match incoming URLs to views. This dispatcher reads a set of URL patterns defined in your **`urls.py`** files and routes the requests accordingly. Each pattern is associated with a specific view, allowing Django to direct traffic to the appropriate part of your application.

Here's a basic example of a **`urls.py`** file:

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

urlpatterns = [
      path('hello/', views.my_view),
]
```

In this example, the URL pattern **`'hello/'`** is linked to the **`my_view`** function in your **`views.py`** file. When a user visits **`http://yourdomain.com/hello/`**, Django will execute **`my_view`** and return its response.

Connecting the blog app urls.py file to the django_blog urls.py file

Django encourages a modular approach to development, where each app within your project has its own **`urls.py`** file. This design keeps your code organized and makes it easier to manage complex projects.

To connect an app's URL patterns to the main project, you need to include the app's **`urls.py`** in the main **`urls.py`** file of your project. However, it's important to note that the **`urls.py`** file is not automatically created when you create a new app. You'll need to create it manually. Here's how you can do it:

- In the **`blog`** app directory, create a new file called **`urls.py`**. This file will hold the URL patterns specific to your app.
- Once the **`urls.py`** file for the **`blog`** app is created, you need to link it to the main project's **`urls.py`** so that Django knows to route requests to your app.

Here is how you can do it:

Open the **Blog app `urls.py`** (**`blog/urls.py`**) and add in the following code:

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

urlpatterns = [
      path('', views.blog_list),
      path('<int:pk>/', views.blog_detail),
]
```

This code defines the URL patterns for your **blog** app and maps specific URL routes to corresponding view functions.

- **`from django.urls import path`**: This imports the **`path`** function, which is used to define URL patterns in Django.
- **`from . import views`:** This imports the **`views`** module from the current directory (the **blog** app's directory), allowing you to reference the view functions defined in **`views.py`.**

`urlpatterns` is a list that contains all the URL patterns for the app. Each pattern maps a URL to a specific view.

URL Patterns

`path('', views.blog_list)`:

- This pattern matches the root URL of the app (**`''`** means no additional path after the app's base URL).
- It calls the **`blog_list`** view function from the **`views`** module when the root URL is accessed. e.g **http://yourdomain.com/**

`path('<int:pk>/', views.blog_detail)`:

- This pattern matches URLs that include an integer value (**`<int:pk>`**). The **pk** stands for "primary key" and represents a unique identifier for a specific blog post.
- It calls the **`blog_detail`** view, passing the integer (**pk**) as an argument to the view function to display the details of a specific blog post. e.g **http://yourdomain.com/1/**

In summary, this code routes two URLs to their respective views:

- The root URL (**`/`**) is routed to the **`blog_list`** view, which will display all blog posts.
- A URL with a post's primary key (like **`/1/`**) is routed to the **`blog_detail`** view, displaying a specific blog post based on its **pk**.

Main Project's `urls.py (django_blog/urls.py)`:

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

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

In this example, the main project's **`urls.py`** includes the **`urls.py`** of the **`blog`** app. By doing so, all URLs without a preceding route like the **admin/** will be routed to the appropriate views defined in the **`blog`** app. For instance, **`http://yourdomain.com/`** will trigger the **`blog_list`** view, and **`http://yourdomain.com/1/`** will trigger the **`blog_detail`** view.

Using URLs in templates

Once you've set up your URLs, the next step is to use them in your templates to allow users to navigate between different parts of your site. Initially, you might be tempted to hardcode the URLs directly into your templates, which works but has several drawbacks. We'll start by showing how this is done, and then in the next section, we'll explore the benefits of naming URLs and using app namespaces to help scale your application more effectively.

For example say you want to link to a blog detail page, you could write something like this:

```
<a href="/blog/3/">Read more</a>
```

This will link to the details page of the blog with the **id** of **3** while this works, it is problematic for several reasons. If the URL pattern ever changes (e.g., if you decide to include a slug or adjust the URL structure), you would have to manually update every instance of the URL across your templates. This can lead to errors, especially in larger projects.

Django offers a solution to that using URL Names and Namespaces for projects with various apps that might have URLs with similar names. Let's look at that next.

Improving with URL Names and Namespaces

Django offers a more dynamic and maintainable approach using *URL names*. By assigning a name to each URL pattern, you can reference URLs by their name in your templates, making your code cleaner and easier to update.

Here is how you can do it:

- Open the **Blog app `urls.py`** (**`blog/urls.py`**) and add in the name part as shown below:

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

urlpatterns = [
```

```
      path('', views.blog_list, name=”blog_list”),
      path('<int:pk>/', views.blog_detail, name=”blog_detail”),
]
```

Then use it your template as shown below:

```
<a href="{% url 'blog_detail' post.id %}">Read more</a>
```

Here, **'blog_detail'** refers to the URL name, and **post.id** is the argument (typically the blog post's ID). Now, if the URL pattern changes, you only need to update it in the **urls.py** file, and Django will handle the rest. This method enhances flexibility, especially as your project grows or evolves.

While that works, issues arise when you have multiple apps in your project, especially if those apps have similar URL names, like **blog_detail**. In such cases, Django wouldn’t know which URL to route the request to and might mistakenly direct the user to the first matching URL, which could lead to the wrong page being displayed.

To solve this, Django allows you to namespace your URLs. This way, Django can differentiate between similarly named URLs across different apps. By assigning each app its own namespace, you prevent naming collisions, ensuring that Django routes the request to the correct app’s URL, providing clarity and eliminating potential conflicts..

To add a namespace for a particular app urls, this is done in the app's **urls.py** file by defining the **app_name** variable. It allows you to group or namespace all the URLs for a specific app, which is useful when you have multiple apps with similar URL patterns:

To set a namespace for the **blog** app open the **blog/urls.py** and add in **app_name** variable as shown below:

```
# blog/urls.py
app_name = 'blog'

urlpatterns = [
      path('', blog_list, name='blog_list'),
      path('<int:id>/', blog_detail, name='blog_detail'),
]
```

Then, in your templates or views, you can refer to the URLs using the namespace, like so:

```
<a href="{% url 'blog:blog_list' %}">Blog</a>
```

Upon doing that Django will be able to know that you are referencing **blog_list** url from the **blog** app.

Summary

Django's URL routing system is essential for connecting your views to the correct URLs, making your application accessible to users. By organizing your URLs within apps and connecting them through the main project's **`urls.py`** file, you can build a scalable and maintainable web application. As we dive deeper, you'll see how to leverage this system to create dynamic, user-friendly experiences.

Templates

In Django, **templates** are HTML files that define how the data from your views should be displayed to users. They allow you to separate the presentation layer (templates or html pages) from the business logic (views), ensuring a clean and maintainable structure.

Templates are used to dynamically generate HTML pages by combining static content (like HTML tags) with dynamic content (like data from the database). In Django, you pass data from views to templates, where the data is rendered into HTML.

Key points about templates in Django:

- **Separation of Concerns**: Templates handle the presentation, while views handle logic and data processing.
- **Dynamic Content**: Templates use Django's template language to embed dynamic content, like displaying blog posts, user information, etc.
- **Reusability**: You can use template inheritance to create a base layout and extend it across multiple pages, making design changes efficient.

In summary, templates in Django are the bridge between your views and the final HTML that users see, ensuring data from your backend is displayed meaningfully on the frontend.

Brief look into Django templating language

To use templates in Django, it's important to understand the Django templating language, which allows you to embed dynamic content in your HTML pages, they also allow you to control the logic and the structure of your templates.

Let's briefly explore some of the most commonly used template tags:

`{% block %}` and **`{% endblock %}`**

Defines a block where child templates can insert their content when they extend a template:

```
{% block content %}
{% endblock %}
```

`{% extends %}`

Indicates that the current template extends or inherits from another template, usually a base template:

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

`{% include %}`

Includes another template within the current template. Useful for inserting common elements like headers, footers, or reusable components:

```
{% include "header.html" %}
```

`{% for %}` and `{% endfor %}`

Iterates over a list or QuerySet and renders content for each item in the list:

```
{% for post in posts %}
  <h2>{{ post.title }}</h2>
  <p>{{ post.content }}</p>
{% endfor %}
```

{% if %}, {% elif %}, and {% endif %}

Adds conditional logic to your templates. You can render different content depending on conditions:

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

`{{ variable }}`

Outputs the value of a variable, such as a model field or context variable, within the template:

```
<p>{{ post.title }}</p>
```

{% url %}

Generates a URL for a given view, using the view's name defined in your URLconf:

```
<a href="{% url 'blog:blog_detail' post.id %}">Read More</a>
```

{% csrf_token %}

Generates a CSRF token to protect forms from Cross-Site Request Forgery attacks. Must be included in forms that send POST requests:

```
<form method="POST">
  {% csrf_token %}
  <!-- form fields -->
</form>
```

{% comment %} and {% endcomment %}

Adds a comment inside the template that will not be rendered in the output :

```
{% comment %}
  This is a comment and won't be rendered in the final HTML.
{% endcomment %}
```

These tags form the backbone of the Django templating system, allowing you to insert data, control flow, and include dynamic content in your HTML templates.With those tags, you can successfully get started using Django templates, and as you continue working with them, you'll naturally learn more advanced tags and features

Setting up templates

To use templates in the **distilled_django_blog** project, we will set up a root **`templates`** directory that contains the base template, which can be inherited by templates from other apps within the project. This allows for a consistent design across the entire application while avoiding repetitive code.

Create base template

Therefore in the root directory (**distilled_django_blog**) create a directory named **templates** and in it create a **base.html** template and put in the following code in it:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>Distilled Django Blog</title>
</head>
<body>
    <main>
        {% block content %}
        {% endblock %}
    </main>
</body>
</html>
```

The provided HTML code is for the base template in the project and it will serve as the foundation for other templates in your project. Let's break it down and focus on the key parts, especially the **`{% block content %}`** and **`{% endblock %}`** tags:

HTML Structure:

`<!DOCTYPE html>`, `<html>`, **`<head>`**, and **`<body>`**: These tags form the basic structure of any HTML page. The **`<head>`** tag contains metadata like the character set and viewport settings for responsiveness, while the **`<body>`** is where the actual content of the webpage is placed.

Base Template:

The **`base.html`** (as this template is named) is used to define a structure that will be reused by other templates, you can give it a different name like main.html or other as long as it is descriptive. It ensures that common elements, like the **`<html>`, `<head>`**, and **`<body>`** sections, are not repeated in every template file.

Django Template Tags:

`{% block content %}` and **`{% endblock %}`**:

- These are **Django template tags** used to define a **block** in the template.
- The block tag allows child templates to **override** specific parts of the base template, replacing the content inside the block.
- In this case, **`{% block content %}`** defines a section where other templates that inherit from this base template can **insert their own content**. The **`{% endblock %}`** tag marks the end of this block.

Purpose of Using Blocks:

- **Reusability**: By using blocks, you can create one main layout (the base template) and allow child templates to insert their specific content without rewriting the entire HTML structure each time.
- **Flexibility**: You can define multiple blocks in your base template (e.g., **`block content`**, **`block sidebar`**, etc.), allowing different parts of the page to be customized in various child templates.
- **Inheritance**: Other templates can extend this base template and override or fill in the content for the **`{% block content %}`** section, making your code DRY (Don't Repeat Yourself).

For example, in a child template, you would use **`{% extends "base.html" %}`** and then define the specific content for the **`content`** block like this:

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

{% block content %}
      <h1>Welcome to the Blog!</h1>
      <p>This is the content specific to this page.</p>
{% endblock %}
```

This system allows you to maintain a consistent layout across your Django site while keeping the individual page content flexible.

Create blog app templates

In addition to the root **`templates`** directory, each app will have its own **`templates`** directory that contains templates specific to that app. Structuring your templates this way is beneficial because it keeps app-specific templates organized and modular, making it easier to manage and maintain different parts of the project. In addition to having a **`templates`** directory within each app, it's a good practice to namespace your templates. This means placing the app's templates in a subdirectory named after the app inside the app-specific **`templates`** folder (e.g., **`blog/templates/blog/`**).

To create templates for the blog app, we'll set up a directory structure as follows:

1. Create a **`templates`** directory in the root of the blog app.
2. Inside this **`templates`** directory, create another directory named **`blog`**.

In this **`blog`** directory, we'll place all the templates specific to the blog app. This structure will result in a path like **`blog/templates/blog/<sometemplate.html>`**.

This approach has several benefits:

1. **Prevents Naming Conflicts**: If multiple apps have similarly named templates (e.g., **`list.html`** or **`detail.html`**), namespacing ensures that each template is correctly associated with its respective app, preventing accidental overwriting or confusion.
2. **Improves Organization**: Namespacing keeps your project organized by clearly separating templates belonging to different apps. This makes it easier to maintain and extend your application.
3. **Enhances Reusability**: By namespacing templates, you ensure that templates can be reused across different apps and projects without worrying about file conflicts.

This structure will help maintain scalability and modularity as your Django project grows. With those settings in place, we can now create templates for the **blog** app and update the views that need the templates.

In the **blog/templates/blog** directory lets create the templates:

blog_list.html

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

{% block content %}
   <a class="create-button" href="{% url 'blog:blog_create' %}">Create new
post</a>
   <div class="blog-list-grid">
    {%for post in posts%}
       <div class="post">
           <a class="title" href="{% url 'blog:blog_detail' post.id
%}">{{ post.title|title }}</a>
           <p>{{ post.snippet }}</p>
           <p class="published">Published on: {{ post.created_at }}</p>
       </div>
   {% endfor %}
   </div>
{% endblock content %}
```

The **{% extends 'base.html' %}** tag and the **{% block content %}** section are repeated in all child templates. They indicate that the child template is inheriting from a base template (**base.html**), and that the content specific to the child template will be inserted within the **{% block content %}** and **{% endblock %}** tags. This structure allows for consistent layouts while enabling customization in different templates.

Here is a breakdown of the other parts:

Create new post:

This is a link that directs users to the page where they can create a new blog post. The URL is generated using the **blog_create** URL name (from the URLs config), ensuring that if the URL changes, this link still works.

** **:

This starts a container that will hold all the blog posts in a grid format. The **class="blog-list-grid"** allows for styling the layout using CSS.

{% for post in posts %} / {% endfor %}:

This is a Django template language loop that iterates over each blog post in the **posts** QuerySet is passed from the view, rendering each post inside a ** ** block.

{{ post.title|title }}:

This creates a clickable title for each blog post. The **href="{% url 'blog:blog_detail' post.id %}"** creates a link to the blog post detail page by appending the post's ID to the URL.

The **{{ post.title|title }}** part displays the title of the post and applies the **title** filter, which capitalizes each word in the title.

** {{ post.snippet }} **:

This displays a snippet of the blog post's content. The **snippet** method, defined in the model, returns the first part of the blog post content with an ellipsis to indicate it's a preview.

** Published on: {{ post.created_at }} **:

This displays the date and time the post was published. The **{{ post.created_at }}** uses Django's template language to insert the **created_at** field from the model, which holds the timestamp of when the blog post was created.

This template structure allows for rendering a list of blog posts dynamically, with each post linking to its detail page and displaying a snippet of its content and publication date. The use of template tags and filters makes it both efficient and scalable.

Create blog_list view

Upon creation of the **`blog_list`** template, we can now update the **`blog_list`** view to send the necessary data to the template instead of simply returning a plain **`HttpResponse`**. This will allow us to render the list of blog posts dynamically within the template, providing a more user-friendly experience.

Open the **blog/views.py** and add in the following code:

```
from django.shortcuts import render, redirect
from .models import Blogs
```

In the first iteration, when we imported and used **`HttpResponse`**, we directly returned raw text or basic responses to the browser. This method is useful for very simple responses but lacks flexibility for more complex rendering.

Now, in the updated version, we're importing **`render`** and **`redirect`** from **`django.shortcuts`**, which provide more powerful ways to handle views:

`render`:

- It allows us to render templates and combine them with data. Rather than sending plain text, we can use this to send structured HTML that incorporates dynamic content from the database.
- It simplifies the process of sending HTML templates to the browser, embedding the context (data like blog posts) within the template.

`redirect`:

- It allows us to redirect users to a different URL after an action (like saving a form or deleting a post). Instead of manually constructing the response, **`redirect`** sends a new request to a different URL.
- This is helpful for actions like saving a blog post and then returning to the blog list page.

Switching from **`HttpResponse`** to **`render`** and **`redirect`** makes the views more sophisticated, enabling templating and more user-friendly navigation.

After the imports create the **blog_list** view:

```
def blog_list(request):
      posts = Blogs.objects.all()  # Retrieve all blog posts from the
database
```

```
      return render(request, 'blog/blog_list.html', {'posts': posts})  # Render the template and pass the blog posts
```

`posts = Blogs.objects.all()`:

- This retrieves all the blog posts from the **`Blogs`** model and stores them in the variable **`posts`**. It fetches all entries in the blog table, which will be sent to the template for display.

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

- This renders the **blog_list.html** template, passing the retrieved posts as context. Django can then iterate through the **posts** queryset and display each post in a separate **div**, allowing the template to dynamically present each blog entry to the user.

blog_detail.html

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

{% block content %}
   <div class="blog-detail">
       <p class="title">{{ post.title }}</p>
       <p>{{ post.content }}</p>
       <p class="published">Published on: {{ post.created_at }}</p>
       <div class="blog-detail-buttons">
           <a class="update-button" href="{% url 'blog:blog_update' post.id %}">Edit Post</a>
           <a class="delete-button" href="{% url 'blog:blog_delete' post.id %}">Delete Post</a>
       </div>
   </div>
{% endblock content %}
```

This renders the detailed view of a specific blog post. Let's explain the other parts of this child template:

** **

A container for the blog post's detail page.

** {{ post.title }} **

Displays the title of the blog post. The variable **{{ post.title }}** syntax is part of the Django templating language, and it outputs the value of the **title** field from the **post** object.

** {{ post.content }} **

Displays the content of the blog post by accessing the **`content`** field of the **`post`** object.

** Published on: {{ post.created_at }} **

Displays the date and time when the blog post was created. The **`created_at`** field stores the timestamp of the blog post's creation.

** **

A container for buttons that allow the user to update or delete the blog post.

Edit Post Button:

```
<a class="update-button" href="{% url 'blog:blog_update' post.id %}">Edit Post</a>
```

This creates a link to the blog post update page. The **`{% url 'blog:blog_update' post.id %}`** part uses the Django **`url`** tag to generate the correct URL for updating the blog post, based on the post's **`id`**.

Delete Post Button:

```
<a class="delete-button" href="{% url 'blog:blog_delete' post.id %}">Delete Post</a>
```

This creates a link to the blog post delete page. Similar to the update button, the **`{% url 'blog:blog_delete' post.id %}`** generates the URL for deleting the specific blog post, using its **`id`**.

In summary, this template renders a detailed view of a blog post, showing the title, content, and creation date, and provides links to update or delete the post.

Create blog_detail view

Upon creation of the **`blog_detail.html`** template, we can now proceed to update the **`blog_detail`** view function. This function will retrieve a specific blog post based on its unique **`id`** and pass it to the template for rendering.

So open the **blog/views.py** and add in the follow code:

```
def blog_detail(request, id):
   post = Blogs.objects.get(id=id)
   return render(request, 'blog/blog_detail.html', {'post':post})
```

`def blog_detail(request, id):`

- This defines the **`blog_detail`** view function, which takes a **`request`** object and an **`id`** parameter to identify the specific blog post to retrieve.

`post = Blogs.objects.get(id=id):`

- This line queries the database to retrieve a single blog post whose **`id`** matches the **`id`** passed in the URL. It uses the **`get()`** method to fetch this object from the **`Blogs`** model.

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

- This renders the **`blog_detail.html`** template, passing the retrieved **`post`** as context to the template so its details (like title, content, and published date) can be displayed.

blog_confirm_delete.html

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

{% block content %}
      <div class="blog delete">
      <h1>Confirm Deletion</h1>

      <p>Are you sure you want to delete the blog post titled "<strong>{{
post.title }}</strong>"?</p>

      <form method="post">
            {% csrf_token %}
            <button type="submit">Yes, delete</button>
            <a class="update-button" href="{% url 'blog:blog_detail'
post.id %}">No, take me back</a>
      </form>
      </div>
{% endblock content %}
```

`<div class="blog delete">`:

A **`<div>`** element that wraps the entire deletion confirmation section. It uses the class **`blog delete`** for potential styling related to blog deletion pages.

`<h1>Confirm Deletion</h1>`:

A header prompting the user to confirm the deletion of the blog post.

`<p>Are you sure you want to delete the blog post titled "<strong>{{ post.title }}</strong>"?</p>`:

This message asks the user to confirm the deletion of the specific blog post, displaying the title of the post dynamically with **`{{ post.title }}`.**

`<form method="post">`:

This form is used to submit the deletion action. It uses the **POST** method since deleting data is considered a "dangerous" operation that modifies the state of the server.

- **`{% csrf_token %}`**: This tag provides protection against Cross-Site Request Forgery (CSRF) attacks, ensuring that the form submission is secure.

`<button type="submit">Yes, delete</button>`:

This button submits the form to confirm the deletion of the blog post.

No, take me back:

This link allows the user to cancel the deletion and return to the detailed view of the blog post. The URL is generated using Django's **`{% url %}`** template tag, dynamically inserting the post's **`id`** to route the user back to the correct blog detail page.

With these three templates in place, you can now display blogs in a more structured manner. While the appearance is not ideal yet, since we haven't applied any styling (we'll address that in the static files section), you can view a list of blogs, click on their titles to navigate to their detail pages, and even delete a blog. The remaining tasks are the creation and updating of blogs, which we'll cover in the next section on forms. However, creating templates alone isn't enough for Django to display them. We still need to configure Django to know where to find the templates in our project. Let's set that up next.

Adding templates setting to the settings.py file

To enable Django to use templates, you must configure the project by specifying the directories in which Django should look for templates within the **TEMPLATES** setting in the project's **settings.py** file. This ensures Django can correctly locate and render your templates when needed.

Here's how you can do it:

1. Open the **django_blog/settings.py** file in your Django project.
2. Locate the **TEMPLATES** setting, which typically looks like this:

```
TEMPLATES = [
      {
      'BACKEND': 'django.template.backends.django.DjangoTemplates',
      'DIRS': [BASE_DIR / 'templates'],  # Add this line to specify your
template directories
      'APP_DIRS': True,
      'OPTIONS': {
            'context_processors': [
                  'django.template.context_processors.debug',
                  'django.template.context_processors.request',
                  'django.contrib.auth.context_processors.auth',
                  'django.contrib.messages.context_processors.messages',
            ],
      },
      },
]
```

DIRS: This is where you tell Django where to find your global templates directory. By setting it to **BASE_DIR / 'templates'**, you're pointing Django to look for templates in a **templates** folder located at the root of your project.

APP_DIRS: When set to **True**, this tells Django to automatically look for templates within the **templates** directory of each app.

With this setup, Django will be able to find both project-wide templates and app-specific templates when rendering views.

Once those configurations are in place, you can now run the project to ensure Django reads the templates and displays the blogs correctly. First, activate your virtual environment (if it's not already active), then start the development server.

Linux / Mac OS:

```
$ . myenv/bin/activate
```

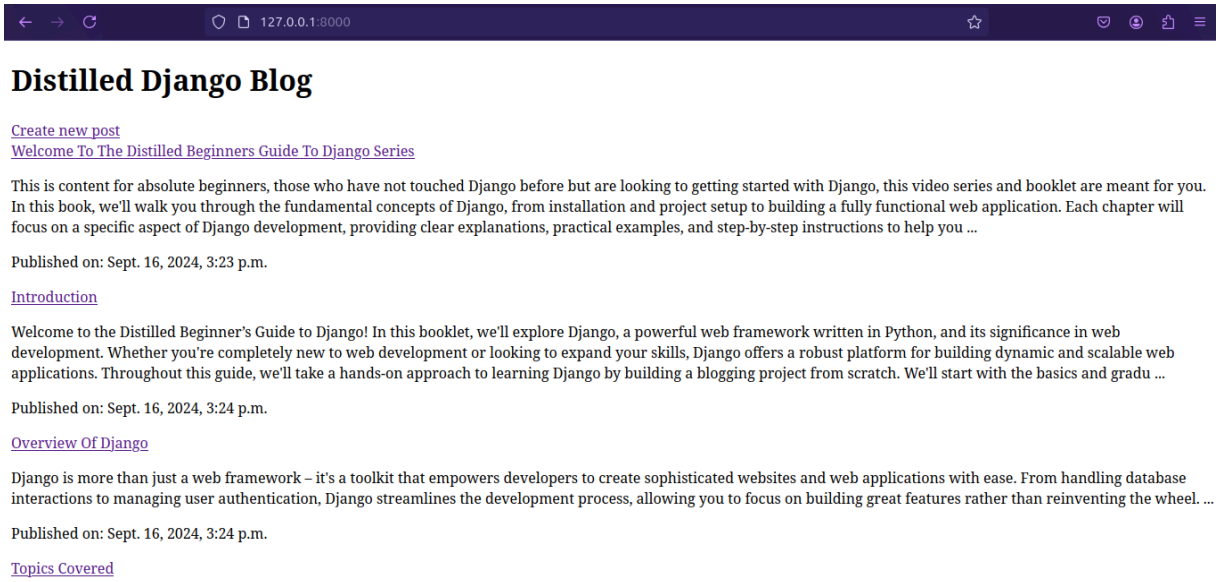
Windows:

```
$ . myenv\Scripts\activate
```

Run the development server:

```
python manage.py runserver or ./manage.py runserver
```

On visiting http://127.0.0.1:8000 you should have a view similar to the one below:



This confirms that you can successfully list the blogs and view their details. In the next section on forms, we'll explore how Django processes form data. After that, we'll update our code to implement the "create blog" and "update blog" views.

Forms

While our application currently displays a list of blog posts and allows viewing the details of a specific post, we still lack the ability to create new posts or edit existing ones. Therefore, we need to implement forms in our application to facilitate the creation and updating of blog posts. Forms provide an interface for users to input and submit data to the backend, enabling us to save new entries or fetch existing data for editing and resubmission.

In this section, we will explore how Django handles forms, understanding its form processing framework, and how we can leverage it to build the create and update views for our application. We will also create form templates to handle the input and editing of blog posts.

What is a form?

At a basic level a HTML form is a collection of elements enclosed within **`<form>...</form>`** that allows users to input data, select options, or manipulate controls such as text fields, checkboxes, or buttons, and then submit that data to the server for processing. These form elements can range from simple input fields to more complex controls enhanced by JavaScript and CSS, such as date pickers or sliders.

In addition to containing input elements, a form must specify two key attributes:

- **where:** the URL to which the form data should be sent, defined in the **`action`** attribute.
- **how:** the HTTP method used to send the data, defined in the **`method`** attribute (usually **`GET`** or **`POST`**).

For example, a Django admin login form has **`input`** elements for the username, password, and submit button. It sends the form data to **`/admin/`** using the **`POST`** method when the user submits the form.

Form Handling in Django

While it is possible to manually write forms in HTML and handle the processing of submitted data yourself, this approach is not recommended. Django simplifies this process by providing the **Form** class, which defines how forms work and appear. From this base **Form** class, Django offers two primary ways to build forms:

1. **Forms not tied to models**: Built using the basic **Form** class, these forms are used when you need custom input fields that are not directly linked to your database models.

2. **Model-bound forms**: Built using the **ModelForm** class, these forms are automatically tied to a specific model, allowing you to easily create and update model instances.
Both subclasses handle form creation, validation, and submission, making it easier to work with forms in Django.

Building a Form

Suppose you want to create a simple form on your website, in order to obtain the user's name. You'd need something like this in your template:

```
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{
current_name }}">
    <input type="submit" value="OK">
</form>
```

This HTML form manually collects a user's name and submits it to the **/your-name/** URL using the POST method. It includes a text input field for the name, pre-filled with the **current_name** variable if available, and a submit button to send the data. Handling this form manually in Django requires writing a view to process the POST request, validate input, and manage errors.

Django simplifies this process with its **Form** and **ModelForm** classes, which automatically generate forms, handle validation, manage errors, and can easily map form fields to model fields, significantly reducing the need for manual form handling.

Building a Form in Django

Before we build the forms for our blog project, let's explore how we can build the above form using the Django's Form class:

Define the Form in Django

```
from django import forms

class NameForm(forms.Form):
      your_name = forms.CharField(label='Your name', max_length=100,
widget=forms.TextInput(attrs={'placeholder': 'Enter your name'}))
```

The above code imports the forms module from Django, providing access to the Form class, which allows us to define forms in a structured and reusable way. By using the Form class, we can easily manage form fields, apply validation, and handle user input without manually creating and processing HTML forms.

Here is an explanation:

Importing the Forms Module

```
from django import forms
```

This line imports Django's forms module, making the **`Form`** class and various field types available for use in creating forms.

Defining the Form Class

```
class NameForm(forms.Form):
```

This line defines a new form class called **`NameForm`**, which inherits from **`forms.Form`**. This means it will have all the properties and methods of a Django form.

Creating a Field

```
your_name = forms.CharField(label='Your name', max_length=100,
widget=forms.TextInput(attrs={'placeholder': 'Enter your name'}))
```

`your_name`: This is the name of the form field. It will be used to reference this input field when processing form submissions.

`forms.CharField`: This indicates that the field will accept character input (i.e., text).

- **`label='Your name'`**: This sets the label for the input field that will be displayed to the user in the form. It tells the user what information they need to enter.

- **`max_length=100`:** This specifies that the maximum length of the input text is 100 characters. If the user attempts to enter more than this limit, validation will fail, and an error message will be displayed.
- **`widget=forms.TextInput(...)`:** This specifies the type of HTML input element to be rendered for this field. In this case, it will be a standard text input box.
    - **`attrs={'placeholder': 'Enter your name'}`:** This adds additional HTML attributes to the input element. Here, a **`placeholder`** attribute is set, which displays a hint to the user inside the text box before they start typing. It helps improve user experience by indicating what should be entered in that field.

The **NameForm** class defines a simple Django form with a single text input field for capturing the user's name. It includes a label and validation constraints, ensuring clarity and guidance for users. When rendered in an HTML template, this form allows users to input their names, while the label and placeholder provide context for the expected input.

This structure exemplifies the typical format for forms in Django, which involves importing the forms module, defining a class that inherits from **forms.Form**, and creating fields with specific attributes for user interaction.

In Django, forms are designed to make it easy for developers to specify input fields and their properties. While model forms—those linked to database models—may have slight variations in their structure and functionality, the foundational principles remain consistent. Understanding this basic format equips you to effectively create both standard and model forms in your Django applications.

The **`NameForm`** class not only illustrates the structure of a Django form but also highlights the importance of **fields** and **widgets** in creating interactive user interfaces:

1. **Fields**: In Django, a field represents a specific piece of data that a user can input. In the **`NameForm`** class, the **`your_name`** field is defined as a **`CharField`,** which is used to collect text input from the user. Each field can have attributes such as **`label`**, **`max_length`**, and various validation constraints, which help ensure that the data entered meets specific criteria.
2. **Widgets**: A widget is a representation of the HTML element used to render a field in a form. In the **`NameForm`,** the **`widget`** attribute is set to **`forms.TextInput`**, which specifies that the field should be rendered as a text input box in the HTML. Additionally, the **`attrs`** parameter allows for customization of the HTML attributes, such as setting a placeholder text, which enhances user experience by providing guidance on what to enter.

These concepts are fundamental in Django forms, allowing developers to create user-friendly and functional forms that gather and validate input effectively.

Build blog app form

With that introduction to forms, we can now build a form specifically for the blog app. The key distinction in this case is that we will create a **model form**, as this will allow us to directly create or edit posts associated with the Blogs model. Model forms in Django simplify the process of handling form data by automatically mapping form fields to model fields, ensuring consistency between user input and database records. Here's how you can create a model form for the blog app:

In the blog app create a new file and name it **forms.py** and place in the following code in it.

blog/forms.py

```
from django import forms
from .models import Blogs

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

The code above defines a model form for the **`Blogs`** model in Django. Here's a breakdown of its components:

Importing Required Modules:

```
from django import forms
from .models import Blogs
```

The **`forms`** module from Django is imported to gain access to the form classes and functionality.

The **`Blogs`** model is imported from the local application's **`models`** module, which allows the form to reference the model's fields.

Defining the BlogForm Class:

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

The **`BlogForm`** class inherits from **`forms.ModelForm`.** This inheritance provides a framework to create a form that is directly tied to a Django model, which simplifies the process of form handling.

Meta Class:

```
class Meta:
```

```
    model = Blogs
    fields = ['title', 'content']
```

The inner **Meta** class is a special configuration class that provides metadata to the **BlogForm.**

The **model** attribute specifies that this form is associated with the **Blogs** model.

The **fields** attribute lists the specific fields from the **Blogs** model that will be included in the form. In this case, the form will have input fields for the **title** and **content** of a blog post.

Using the custom form class

Once you have created the custom form class, you'll need to incorporate it into your HTML templates to render the form for users. Additionally, you will want to access this form in your views to handle the data it submits. To achieve this, we will update the blog views by implementing two new views: **blog_create** and **blog_update**.

These views will utilize the form class to facilitate the creation and editing of blog posts, allowing users to submit their input, which can then be processed and saved to the database.

Create blog_create view

To enable the creation of new blog posts, we need to implement a view that will instantiate the form when a user first visits the URL for creating a post. Additionally, this view must handle the data that the user submits through the form and send it back to the backend for processing. Here's how to accomplish this:

Open the **blog/views.py** and add in the following code:

```
# other imports
from .forms import BlogForm

# other views
def blog_create(request):
   if request.method == 'POST':
       form = BlogForm(request.POST)
       if form.is_valid():
           form.save()
           return redirect('blog:blog_list')
   else:
       form = BlogForm()
   return render(request, 'blog/blog_form.html', {'form':form})
```

```

```

Here is a breakdown of the above code for the **blog_create** view:

Importing the Form:

```
from .forms import BlogForm
```

This line imports the **`BlogForm`** class from the forms module, allowing us to use it within the view.

Defining the View Function:

```
def blog_create(request):
```

This line defines a view function called **`blog_create`,** which will handle requests to create a new blog post.

Handling POST Requests:

```
if request.method == 'POST':
      form = BlogForm(request.POST)
```

This checks if the request method is **POST**, indicating that the user has submitted the form.

If so, it instantiates the **`BlogForm`** with the data submitted (**`request.POST`**).

Validating the Form:

```
if form.is_valid():
      form.save()
      return redirect('blog:blog_list')
```

The form's **`is_valid()`** method is called to check if the submitted data meets the specified validation criteria.

If the form is valid, **`form.save()`** is executed to save the new blog post to the database.

After saving, the user is redirected to the blog list page.

Handling GET Requests:

```
else:
      form = BlogForm()
```

If the request method is not **POST** (i.e., it's a **GET** request), a new instance of **BlogForm** is created to display an empty form for the user to fill out.

Rendering the Template:

```
return render(request, 'blog/blog_form.html', {'form': form})
```

Finally, the view renders the **blog_form.html** template, passing the form instance to the template context so that it can be displayed.

Create blog_update view

To enable the updating of blog posts, we need to implement a view that instantiates the form with the specific post the user has selected to edit. This view will prefill the form with the existing information from that post, allowing the user to make changes before saving. Here's how to create the update view:

Open the **blog/views.py** and add in the following code:

```
def blog_update(request, id):
   post = Blogs.objects.get(id=id)
   if request.method == 'POST':
       form = BlogForm(request.POST, instance=post)
       if form.is_valid():
           form.save()
           return redirect('blog:blog_detail', id=post.id)
   else:
       form = BlogForm(instance=post)
   return render(request, 'blog/blog_form.html', {'form': form})
```

Defining the view function:

```
def blog_update(request, id):
```

This line defines the **blog_update** view function, which takes two parameters: **request**, representing the HTTP request, and **id**, which identifies the specific blog post to be updated.

Retrieve the Blog Post:

```
post = Blogs.objects.get(id=id)
```

Here, the function retrieves the blog post using its **id** and assigns it to the post variable.

`Handle POST Requests`:

```
if request.method == 'POST':
```

This conditional checks if the incoming request is a **POST** request, indicating that the user has submitted the form for updating the blog post.

Instantiate the Form with Submitted Data:

```
form = BlogForm(request.POST, instance=post)
```

When the request is a **`POST`**, a **`BlogForm`** is created using the submitted data (**`request.POST)`** and the existing post instance. This allows the form to be prefilled with the current values of the post.

Validate the Form:

```
if form.is_valid():
```

The **`is_valid()`** method is called to validate the form data. If the data is valid, the following steps will execute.

Save the Updated Post:

```
form.save()
```

If the form is valid, the updated post is saved to the database.

Redirect After Success:

```
return redirect('blog:blog_detail', id=post.id)
```

After saving, the user is redirected to the detail view of the updated post, which prevents accidental resubmissions if the page is refreshed.

Handle GET Requests:

```
else:
      form = BlogForm(instance=post)
```

If the request method is not **`POST`** (i.e., it's a **`GET`** request), a new **`BlogForm`** is instantiated with the existing post data to populate the form fields with the current information for editing.

Render the Template:

```
return render(request, 'blog/blog_form.html', {'form': form})
```

Finally, the view renders the **`blog_form.html`** template, passing the form as context so it can be displayed to the user.

This structure allows users to efficiently edit existing blog posts while maintaining data integrity and providing a seamless user experience.

In summary, by using Django's form handling capabilities, you can efficiently create views that handle both the creation and updating of posts. The Form and ModelForm classes handle much of the heavy lifting, such as rendering HTML forms, validating input, and saving the data to the database. This built-in functionality saves you from having to manually create and manage forms, making the process of building and maintaining forms in Django streamlined and more secure.

Create blog_form.html template

With the two views in place, you now have the ability to create and update posts. However, these views reference a **blog_form.html** template, which we haven't created yet. To make everything functional, we need to create this template and tie it all together. Let's do that now:

In the **blog/templates/blog** directory create a template and name it **blog_form.html** and put the following code:

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

{% block content %}
   <div class="blog-form-page">
       <h1 class="title">{% if form.instance.id %}Edit{% else %}Create{%
endif %} Post</h1>
       <form class="blog-form" method="post">
           {% csrf_token %}
           {{ form.as_p }}
           <button type="submit">{% if form.instance.pk %}Update{% else
%}Publish{% endif %} Post</button>
       </form>
       </div>
{% endblock content %}
```

Here is a breakdown of the **blog_form.html** template:

** **

This **`div`** wraps the entire form and is used for styling purposes. It gives a structure to the page content.

<h1 class="title">{% if form.instance.id %}Edit{% else %}Create{% endif %} Post</h1>

This is a dynamic heading that changes depending on whether the form is being used to create a new post or edit an existing one:

- If the form has an instance with an ID (**`form.instance.id`**), it shows "Edit Post."
- Otherwise, it shows "Create Post."

This gives users context about whether they are editing or creating a post.

<form class="blog-form" method="post">

This is the HTML **`<form>`** element that will wrap all input elements related to the blog form. It uses the POST method to submit the form data to the server.

{% csrf_token %}

This Django template tag adds a CSRF (Cross-Site Request Forgery) token for security purposes. It ensures that the form submission is coming from a trusted source.

{{ form.as_p }}

This line renders the form fields using Django's **`as_p`** method, which wraps each field in a **`<p>`** element. It automatically generates the necessary HTML for all fields defined in the form class (**`BlogForm`**), which includes the title and content fields in this case.

<button type="submit">{% if form.instance.pk %}Update{% else %}Publish{% endif %} Post</button>

This is the submit button for the form. Its label dynamically changes based on whether the form is for creating or updating a post:

- If the form's instance has a primary key (**`form.instance.pk`**), it shows "Update Post."
- Otherwise, it shows "Publish Post" for creating a new post.

{% endblock content %}

This ends the **`content`** block, ensuring that the rest of the layout from the parent template (**`main.html`**) will be displayed as expected.

This template dynamically adjusts based on whether the user is creating a new post or editing an existing one. It leverages Django's form handling capabilities to simplify the HTML form generation and CSRF protection.

With the templates and views in place, users can now create and edit blog posts. The integration of forms signals that our application has all the essential views needed to function fully. At this point, it's time to test the system and ensure everything works as expected. Let's give it a whirl and verify that all the components are performing correctly.

First, activate your virtual environment (if it's not already active), then start the development server.

Linux / Mac OS:

```
$ . myenv/bin/activate
```

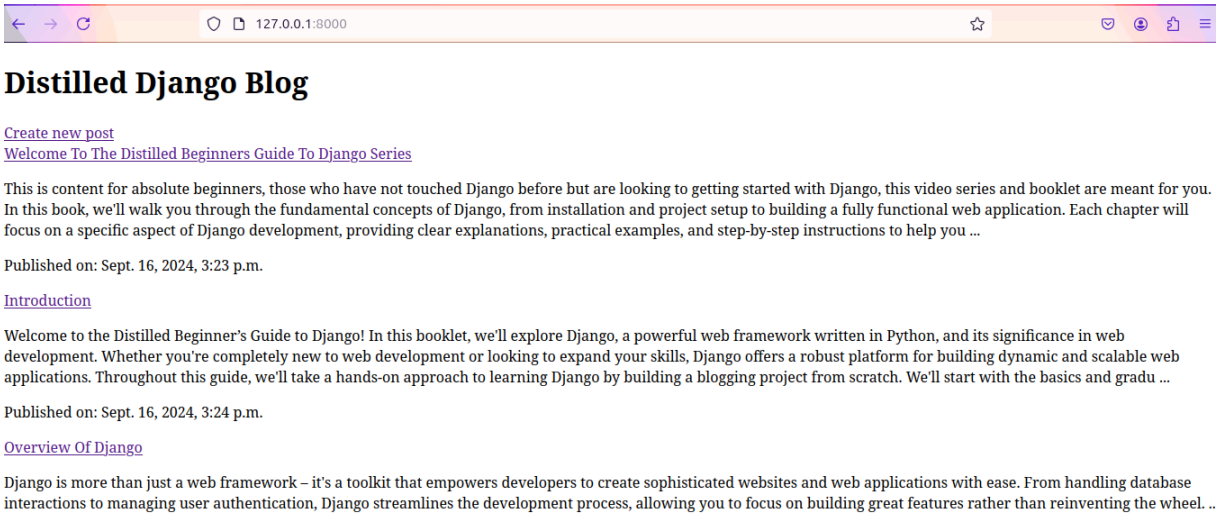
Windows:

```
$ . myenv\Scripts\activate
```

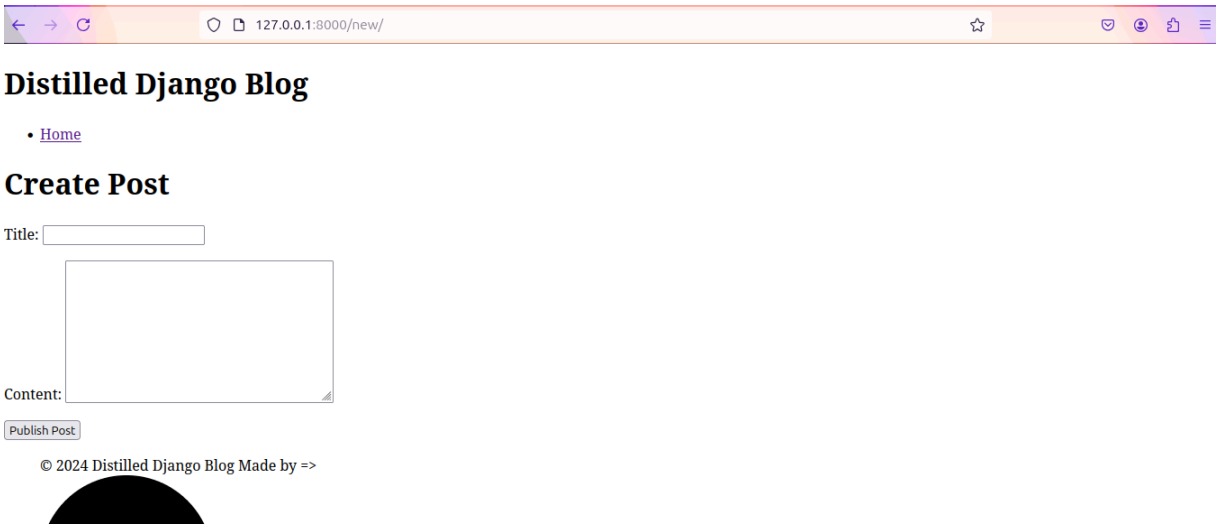
Run the development server:

```
python manage.py runserver or ./manage.py runserver
```

On visiting http://127.0.0.1:8000 you should have a view similar to the one below:



That view is the home page or the blog listing page. To test whether you can create a post click on the **create new post** link and that will take you to a page similar to the one below:



If you can see such a page then the form is being rendered correctly and you can create a dummy post and click on publish post to save it. I will not try editing the post. I will leave you to explore that.

With all the form-related settings now in place, this concludes the section on forms. However, it's important to note that we've only scratched the surface of Django's powerful form handling capabilities. There is much more functionality you can explore, and for a deeper understanding, I highly recommend referring to the official Django documentation.

As we wrap up, you might have noticed that the site's design still lacks a polished look and feel. To address this, we'll dive into static files in the next section and explore how Django manages them. This will allow us to style the project and enhance its overall visual appeal.

Static Files

- Managing static files (CSS, JavaScript, images) in Django projects.
- Serving static files during development and production.

Authentication

- Implementing user authentication in Django.
- Managing user accounts and passwords.
- Customizing authentication views and templates.

Admin Interface

- Exploring Django's built-in admin interface for managing site content.
- Registering models with the admin site.
- Customizing the admin interface.

Deployment

- Deploying Django websites to production servers.
- Configuring server environments for Django applications.
- Best practices for deploying Django projects.

Conclusion

- Recap of key concepts covered in the booklet.
- Next steps for further learning and exploration in Django development.