Table of Contents

Prerequisites

Before diving into building the blogging website, ensure you have the following:

- **Python** installed on your machine (preferably Python 3.6 or higher).
- **pip** (Python package installer) installed.
- Basic knowledge of Python and HTML.
- Familiarity with Django's MVC (Model-View-Controller) architecture is a plus.

Project Setup

1. Install Django

First, install Django using pip:

```
pip install django
```

2. Create a New Django Project

Navigate to the directory where you want to create your project and run:

```
django-admin startproject myblogproject
cd myblogproject
```

This command creates a new Django project named `myblogproject` with the following structure:

```
myblogproject/
    manage.py
    myblogproject/
        __init__.py
        settings.py
        urls.py
```

```
        wsgi.py
```

3. Create a Virtual Environment (Optional but Recommended)
It's good practice to create a virtual environment to manage dependencies:

```
python -m venv env
source env/bin/activate  # On Windows: env\Scripts\activate
pip install django
```

Creating the Blog App
Django projects are composed of apps. We'll create a `blog` app within our project:

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

The project structure now includes:

```
myblogproject/
    blog/
        __init__.py
        admin.py
        apps.py
        migrations/
            __init__.py
        models.py
        tests.py
        views.py
    manage.py
    myblogproject/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

4. Add the Blog App to `INSTALLED_APPS`
Open `myblogproject/settings.py` and add `'blog'` to the `INSTALLED_APPS` list:

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

```
]
```

Defining Models

Models define the structure of your database. We'll create a `Post` model to represent blog posts.

1. Using Django's Built-in `User` Model

We'll use Django's built-in `User` model to represent authors. This provides built-in authentication and user management.

2. Define the `Post` Model

In `blog/models.py`, define the `Post` model:

```
from django.db import models
from django.contrib.auth.models import User
from django.utils import timezone

class Post(models.Model):
    author = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='posts')
    title = models.CharField(max_length=200)
    content = models.TextField()
    publish_date = models.DateTimeField(default=timezone.now)
    image = models.ImageField(upload_to='post_images/', blank=True,
null=True)

    class Meta:
        ordering = ['-publish_date']

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

Explanation:

- `author`: A foreign key linking to the `User` model. When a user is deleted, all their posts are also deleted (`on_delete=models.CASCADE`).
- `title`: The title of the blog post.
- `content`: The main content of the post.
- `publish_date`: Automatically set to the current date and time when a post is created.
- `image`: An optional image associated with the post. Images are uploaded to the `post_images/` directory.
- `Meta`: Specifies that posts are ordered by `publish_date` in descending order.

- `__str__`: Returns the title of the post as its string representation.

3. Apply Migrations

After defining the model, create and apply migrations to update the database schema:

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

Setting Up the Admin Interface

Django provides a built-in admin interface to manage models.

1. Create a Superuser

To access the admin site, create a superuser:

```
python manage.py createsuperuser
```

Follow the prompts to enter a username, email, and password.

2. Register the `Post` Model

In `blog/admin.py`, register the `Post` model:

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

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'publish_date')
    list_filter = ('author', 'publish_date')
    search_fields = ('title', 'content')
```

Explanation:

- `list_display`: Columns displayed in the admin list view.
- `list_filter`: Filters available in the sidebar of the admin list view.
- `search_fields`: Enables a search box to search through specified fields.

3. Access the Admin Site

Run the development server and navigate to http://127.0.0.1:8000/admin/. Log in using the superuser credentials. You should see the `Posts` model available for management.

```
python manage.py runserver
```

Creating Forms

Forms handle user input. We'll create a form for authors to create and edit blog posts.

1. Create a `PostForm`

In `blog/forms.py`, define `PostForm`:

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

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['title', 'content', 'image']
        widgets = {
            'title': forms.TextInput(attrs={'class': 'form-
control'}),
            'content': forms.Textarea(attrs={'class': 'form-
control'}),
            'image': forms.ClearableFileInput(attrs={'class': 'form-
control-file'}),
        }
```

Explanation:

- `ModelForm`: Creates a form based on the `Post` model.
- `fields`: Specifies which model fields to include in the form.
- `widgets`: Defines HTML attributes for form fields to style them with CSS classes.

Configuring Views

Views handle the logic of your application. We'll create views for listing posts, viewing post details, creating new posts, and editing posts.

1. Import Necessary Modules

In `blog/views.py`, import required modules:

```
from django.shortcuts import render, get_object_or_404, redirect
from django.contrib.auth.decorators import login_required
from .models import Post
from .forms import PostForm
```

2. Create a View to List All Posts

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

3. Create a View for Post Details

```
def post_detail(request, pk):
    post = get_object_or_404(Post, pk=pk)
    return render(request, 'blog/post_detail.html', {'post': post})
```

4. Create a View to Create a New Post

```
@login_required
def post_create(request):
    if request.method == "POST":
        form = PostForm(request.POST, request.FILES)
        if form.is_valid():
            # Assign the current user as the author
            post = form.save(commit=False)
            post.author = request.user
            post.save()
            return redirect('post_detail', pk=post.pk)
    else:
        form = PostForm()
    return render(request, 'blog/post_form.html', {'form': form})
```

Explanation:

- `@login_required`: Ensures that only authenticated users can create posts.
- Handles both GET and POST requests.
- On POST, validates the form and saves the post with the current user as the author.
- Redirects to the post detail page upon successful creation.

5. Create a View to Edit an Existing Post

```
@login_required
def post_edit(request, pk):
    post = get_object_or_404(Post, pk=pk)
    if request.user != post.author:
        return redirect('post_detail', pk=post.pk)

    if request.method == "POST":
        form = PostForm(request.POST, request.FILES, instance=post)
        if form.is_valid():
            form.save()
            return redirect('post_detail', pk=post.pk)
    else:
        form = PostForm(instance=post)
    return render(request, 'blog/post_form.html', {'form': form})
```

Explanation:

- Checks if the logged-in user is the author of the post.
- Allows editing of the post if the user is the author.
- Uses the same `PostForm` for both creating and editing posts.

Setting Up URLs

URLs map to views. We'll define URL patterns for our blog app.

1. Create `blog/urls.py`

Create a new file `blog/urls.py` and define URL patterns:

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

urlpatterns = [
    path('', views.post_list, name='post_list'),
    path('post/<int:pk>/', views.post_detail, name='post_detail'),
    path('post/new/', views.post_create, name='post_create'),
    path('post/<int:pk>/edit/', views.post_edit, name='post_edit'),
]
```

2. Include Blog URLs in Project's URL Configuration

In `myblogproject/urls.py`, include the blog app's URLs and configure media file serving during development:

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('blog.urls')),
    path('accounts/', include('django.contrib.auth.urls')),  # For
authentication
]

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

Explanation:

- `path('', include('blog.urls'))`: Routes the root URL to the blog app.
- `path('accounts/', include('django.contrib.auth.urls'))`: Includes Django's built-in authentication URLs.
- `static()`: Serves media files during development.

Creating Templates

Templates define the HTML structure of your web pages. We'll create templates for listing posts, viewing details, and forms.

1. Configure Template Settings

In `myblogproject/settings.py`, ensure that the `TEMPLATES` setting includes the `DIRS` option to look for templates in a global `templates` directory:

```
import os

TEMPLATES = [
    {
        # ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        # ...
    },
]
```

2. Create Template Directories

Create the following directory structure:

```
myblogproject/
    templates/
        blog/
            post_list.html
            post_detail.html
            post_form.html
        registration/
            login.html
    static/
        css/
            styles.css
```

3. Create a Base Template

Create `templates/base.html` to serve as the base for other templates:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}My Blog{% endblock %}</title>
    <link rel="stylesheet" href="{% static 'css/styles.css' %}">
</head>
<body>
    <header>
        <h1><a href="{% url 'post_list' %}">My Blog</a></h1>
```

```
        <nav>
            {% if user.is_authenticated %}
                <span>Welcome, {{ user.username }}!</span>
                <a href="{% url 'post_create' %}">New Post</a>
                <a href="{% url 'logout' %}">Logout</a>
            {% else %}
                <a href="{% url 'login' %}">Login</a>
                <a href="{% url 'signup' %}">Sign Up</a>
            {% endif %}
        </nav>
    </header>
    <div class="container">
        {% block content %}
        {% endblock %}
    </div>
</body>
</html>
```

Explanation:

- Defines a consistent structure for all pages.
- Includes navigation links that change based on user authentication status.
- Uses {% block %} tags to allow child templates to insert content.

4. Create post_list.html

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

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

{% block content %}
    <h2>All Posts</h2>
    {% for post in posts %}
        <div class="post">
            <h3><a href="{% url 'post_detail'
pk=post.pk %}">{{ post.title }}</a></h3>
            <p>By {{ post.author.username }} on
{{ post.publish_date }}</p>
            {% if post.image %}
                <img src="{{ post.image.url }}"
alt="{{ post.title }}" style="max-width:200px;">
            {% endif %}
            <p>{{ post.content|truncatewords:30 }}</p>
            <a href="{% url 'post_detail' pk=post.pk %}">Read
more</a>
```

```
        </div>
    {% empty %}
        <p>No posts have been published yet.</p>
    {% endfor %}
{% endblock %}
```

5. Create post_detail.html

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

{% block title %}{{ post.title }} - My Blog{% endblock %}

{% block content %}
    <h2>{{ post.title }}</h2>
    <p>By {{ post.author.username }} on {{ post.publish_date }}</p>
    {% if post.image %}
        <img src="{{ post.image.url }}" alt="{{ post.title }}"
style="max-width:400px;">
    {% endif %}
    <div>
        {{ post.content|linebreaks }}
    </div>
    {% if user == post.author %}
        <a href="{% url 'post_edit' pk=post.pk %}">Edit Post</a>
    {% endif %}
    <a href="{% url 'post_list' %}">Back to all posts</a>
{% endblock %}
```

6. Create post_form.html

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

{% block title %}New Post - My Blog{% endblock %}

{% block content %}
    <h2>{% if form.instance.pk %}Edit Post{% else %}New Post{%
endif %}</h2>
    <form method="post" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Save</button>
    </form>
    <a href="{% url 'post_list' %}">Cancel</a>
```

```
{% endblock %}
```

7. Create `login.html`

Copy Django's default `login.html` into `templates/registration/login.html` and customize as needed:

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

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

{% block content %}
    <h2>Login</h2>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Login</button>
    </form>
    <a href="{% url 'signup' %}">Don't have an account? Sign up here.</a>
{% endblock %}
```

Handling User Authentication

We'll implement user registration alongside Django's built-in authentication.

1. Create a Signup View

In `blog/views.py`, add a view for user registration:

```
from django.contrib.auth import login
from django.contrib.auth.forms import UserCreationForm

def signup(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)  # Log the user in after signup
            return redirect('post_list')
    else:
        form = UserCreationForm()
    return render(request, 'blog/signup.html', {'form': form})
```

2. Create `signup.html` Template

Create `templates/blog/signup.html`:

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

{% block title %}Sign Up - My Blog{% endblock %}

{% block content %}
    <h2>Sign Up</h2>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Sign Up</button>
    </form>
    <a href="{% url 'login' %}">Already have an account? Login
here.</a>
{% endblock %}
```

3. Add Signup URL

In `blog/urls.py`, add the signup path:

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

urlpatterns = [
    # Existing paths...
    path('signup/', views.signup, name='signup'),
]
```

4. Update Navigation in `base.html`

Ensure that the "Sign Up" link directs to the signup page. This is already handled in the `base.html` provided earlier.

Managing Media Files

To handle image uploads, configure media settings.

1. Install Pillow

Pillow is required to handle image fields:

```
pip install Pillow
```

2. Configure Media Settings

In `myblogproject/settings.py`, add the following at the end:

```
import os

MEDIA_URL = '/media/'
```

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

3. Update URL Configuration
As previously shown in the URLs section, ensure that media files are served during development by appending the following to `urlpatterns`:

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

Note: In production, serving media files should be handled by the web server (e.g., Nginx, Apache).
4. Use the `image` Field in Templates
Image fields are already handled in the `post_list.html` and `post_detail.html` templates.

Running the Development Server
With all components in place, start your Django development server:

```
python manage.py runserver
```

Navigate to http://127.0.0.1:8000/ to see the blog in action.
Testing the Features:

1. **Access Admin Site:**
    a. Navigate to http://127.0.0.1:8000/admin/.
    b. Log in using the superuser account.
    c. Add some posts if desired.
2. **User Registration:**
    a. Click on "Sign Up" in the navigation bar.
    b. Create a new user account.
3. **Create a New Post:**
    a. After logging in, click on "New Post".
    b. Fill out the form and submit to create a new blog post.
4. **View and Edit Posts:**
    a. Click on a post title to view its details.
    b. If you're the author, an "Edit Post" link will be available to modify the post.
5. **Logout:**
    a. Click on "Logout" to sign out.

Conclusion
You've successfully built a simple blogging website using Django, complete with user authentication, post creation and editing, image uploads, and an admin interface for

managing content. This project serves as an excellent foundation for teaching Django web development, illustrating core concepts such as models, views, templates, forms, user authentication, and media handling.

Next Steps:

- **Enhance Styling:** Improve the appearance using CSS frameworks like Bootstrap.
- **Add Pagination:** Implement pagination for the post listings.
- **Comment System:** Allow users to comment on posts.
- **Categories/Tags:** Organize posts using categories or tags.
- **Search Functionality:** Implement search to find posts by keywords.