Project Setup

1. Create environment

```
python3 -m venv my_env
```

2. Activate virtual environment

```
source my_env/bin/activate
```

3. Install Django

```
pip install Django
```

4. Create `mysite` project

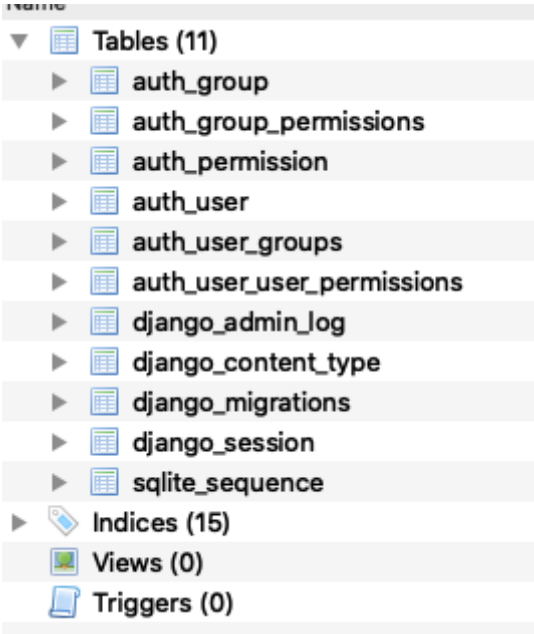
```
django-admin startproject mysite
```

5. Run migration to setup tables for applications

    - NOTE: you must change directory to the project folder before running migration

```
cd mysite
python manage.py migrate
```

    - The following tables are created



6. Run the Django development server to verify installation

```
python manage.py runserver
```

Setup blog application

1. Create basic structure of blog application within project folder

```
python manange.py startapp blog
```

2. Create the Post model

   - The Post model subclasses the `django.models.Model` class in which each attribute field represents a database field
   - A `slug` is a short label that contains only letters, underscores, numbers, or hyphens and is used to create SEO-friendly URLs
   - the `author` field creates a `many-to-one` relationship where many posts are associated with a specific author(e.g. User)
   - the `status` field uses a choices parameter to reference the constant `STATUS_CHOICES` in which only one item can be assigned. This constant is a `tuple of tuples`

```
from django.db import models
from django.utils import timezone # needed for timestamp of publish,
```

```
created, & updated attributes
from django.contrib.auth.models import User

# Create your models here.
class Post(models.Model):
    STATUS_CHOICES = (
        ('draft', 'Draft'),
        ('published', 'Published'),
    )

    title = models.CharField(max_length = 250)
    slug = models.SlugField(max_length = 250, unique_for_date =
'publish')
    author = models.ForeignKey(User, on_delete = models.CASCADE,
related_name = 'blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default = timezone.now) # date
with timezone info
    created = models.DateField(auto_now_add = True) # date when post
initially created
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length = 10, choices =
STATUS_CHOICES, default = 'draft')

    class Meta:     # just a class container with some options
(metadata)
        ordering = ('-publish', )   # the negative puts in
descending order from most recently published

    def __str__(self):   # creates a human-readable representation
of the object
        return self.title
```

3. Activate the application

    - within `apps.py` is the following class

```
class BlogConfig(AppConfig):
    name = 'blog'
```

- add the following to `mysite/settings.py` to activate the app. This tells Django tht this app belongs to projects and to load its models

```
INSTALLED_APPS = [
    'blog.apps.BlogConfig',
    'django.contrib.admin',
```

4. Create initial migration for the `Post` model
    - this defines how the database will be modified

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

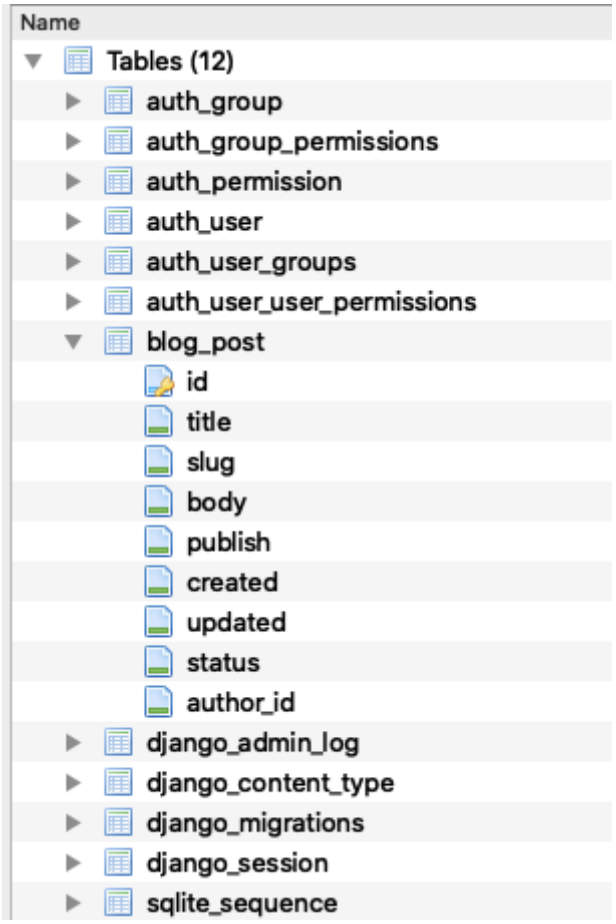
    - to see the SQL(it won't actually be run) that Django will run to implement the migration, run the following

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

5. Apply the migration

```
python manage.py migrate
```

- the database now has the `blog_post` table



Setup Admin

1. Create `superuser`

```
python manage.py createsuperuser
```

2. Register models to admin site

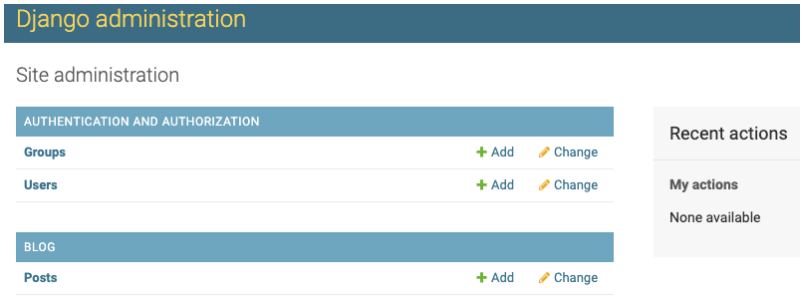
   - orig

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

   - updated

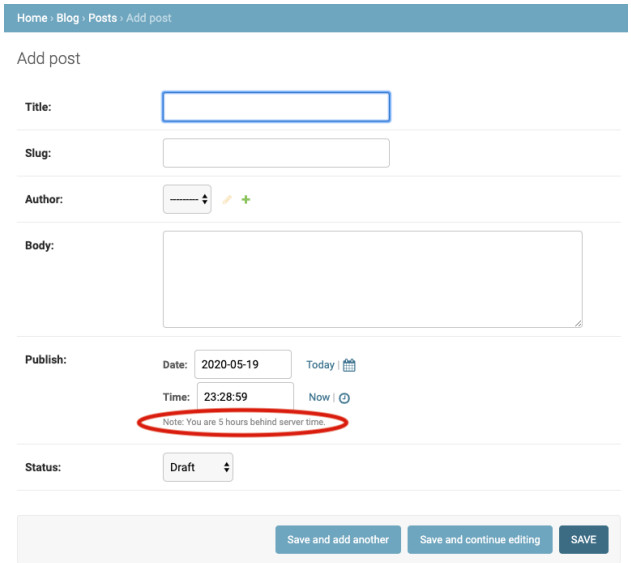
```
from django.contrib import admin
from .models import Post

# Register your models here.
admin.site.register(Post)
```

3. Launch server and log into admin panel at URL `http://127.0.0.1:8000/admin` to see the admin panel



4. Select Add post and note timezone message

   - message varies depending on your actual timezone



- this can be resolved by modifying `TIME_ZONE` in `settings.py` to your actual timezone
- before

```
TIME_ZONE = 'UTC'
```

- after

```
TIME_ZONE = 'America/Chicago'
```

- However, modifying `TIME_ZONE` can cause issues with Daylight Savings Time. It is recommended to use `UTC` time in the database and convert to `local time` for user interactions. see Time zones Django documentation

Customize admin model

1. Add the following model to `admin.py`
   - note the `admin options`
     - Django admin options

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

# Register your models here.
# admin.site.register(Post)
```

```
# Custom models
@admin.register(Post)   # decorator performs same as
admin.site.register(Post)
class PostAdmin(admin.ModelAdmin):
   list_display = ('title', 'slug', 'author', 'publish', 'status')
   list_filter = ('status', 'created', 'publish', 'author')
   search_fields = ('title', 'body')
   prepopulated_fields = {'slug': ('title',)}
   raw_id_fields = ('author',)
   date_hierarchy = 'publish'
   ordering = ('status', 'publish')
```

Create list & detail views

1. Add the following views

```
from django.shortcuts import render, get_object_or_404
from .models import Post

# Create your views here.
def post_list(request):
    posts = Post.published.all()
    return render(request, 'blog/post/list.html', {'posts': posts})


def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
        slug = post,
        status = 'published',
        publish__year = year,
        publish__month = month,
        publish__day = day)

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

2. Add URL patterns for views in the blog app

   - this maps URLs to views
   - the first pattern does not have arguments
   - the second pattern take four arguments
   - angle brackets are used to capture values from a URL as a strings
   - `path converters` are used to capture values. For example, int:year looks for a int parameter and returns an integer. Likewise, slug:post matches a slug string
   - Django path converters
   - `name` maps the view

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

app_name = 'blog'

urlpatterns = [
    # post views
    path('', views.post_list, name = 'post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
views.post_detail, name = 'post_detail'),

]
```

3. Update the project `urls.py`
    - add the `include` import
    - add the following to the `urlpatterns` variable
    - the `namespace` blog allow precise reversing of `names URL patterns`

```
from django.urls import path, include

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

]
```

Implement Canonical URLs for models

```
- Canonical means `preferred` and is a unique URL
- the `reverse` method allows URLs to be built using their name and also
allows passing additional parameters

- Add the following to `models.py`
- import `reverse`

    ~~~ py
    from django.urls import reverse
    ~~~
```

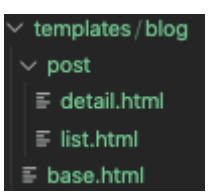
```
- create `get_absolute_url` method to link to specific posts

    ~~~ py

    def get_absolute_url(self):
        return reverse("blog:post_detail",   # define args next, kwargs can
also be implmented
                          args=[self.publish.year,
                                self.publish.month,
                                self.publish.day,
                                self.slug ])

    ~~~
```

Update the models

```
- import `reverse`

    ~~~ py
    from django.urls import reverse
    ~~~

-
```

Create templates for the views

1. Set up the following folders and files inside the `blog` app



   - use template tags, template variables, and template filters to create templates

2. Create the `base.html` template

   - utilizes `static files`

```
{% load static %}
<!DOCTYPE html>
<html>
    <head>
       <title>{% block title %} {% endblock %} </title>
```

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

    <body>

        <div id = "content">
            {% block content %}

            {% endblock %}

        </div>

        <div id = "sidebar">
            <h2> My blog </h2>
            <p> This is my blog </p>

        </div>

    </body>

</html>
```

3. Create the `list.html`template

   - `extends` allows this template to inherit from the `base.html` file
   - Two template filters are applied in the body of the post

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

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

{% block content %}
    <h1> My Blog! </h1>

    {% for post in posts %}
        <h2>
            <a href = "{{ post.get_absolute_url }}">
                {{ post.title }}
            </a>
        </h2>

        <p class = "date">
            Published {{ post.publish }} by {{ post.author }}
        </p>

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

```
        {% endfor %}

    {%endblock%}
```

4. Create `detail.html` template

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

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

{% block content %}
    <h1> {{post.title}} </h1>
    <p class = "date">
        Published {{post.publish}} by {{post.author}}
    </p>

    {{post.body|linebreaks}}
{% endblock %}
```

Add Pagination

1. In `views.py` add the following import

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
```

2. Within `template\blog` create `pagination.html` template

```
<div class = "pagination">
    <span class = "step-links">
    {% if page.has_previous %}
        <a href = "?page = {{ page.previous_page_number }}">Previous</a>
    {% endif %}

        <span class = "current">
            Page {{page.number}} of {{page.paginator.num_pages}}.
        </span>

        {% if page.has_next %}
            <a href = "?page={{page.next_page_number }}">Next</a>
        {%endif%}
    </span>
</div>
```

3. Within the `list.html` template, add the following to refer to the pagination template

```
...
{% endfor %}

    {% include "pagination.html" with page=posts %}

{%endblock%}
```

Using Class-based views

- Views are implemented as Python objects instead of functions

1. Add `from django.views.generic import ListView` to `views.py`

2. Create the following class-based view in `views.py`

- The following two lines are analogous and create the queryset

```
    model = Post
    # queryset = Post.published.all()
```

- Although `object_list` is generically created for the query results, using `context_object_name` makes your code easier to follow

```
class PostListView(ListView):
    model = Post
    # queryset = Post.published.all()
    context_object_name = 'posts'
    paginate_by = 3
    template_name = 'blog/post/list.html'
```

3. Modify `blog\urls.py` to use the `PostListView` class

```
urlpatterns = [
    # post views
    # path('', views.post_list, name = 'post_list'),
    path('', views.PostListView.as_view(), name = 'post_list'),
    ...
```

4. Update the `list.html` file to receive an obj

- NOTE: you must not put any spaces within `page=page_obj`

```
    {% endfor %}

    <!--  {% include "pagination.html" with page=posts %} -->
    {% include "pagination.html" with page=page_obj %}
{%endblock%}
```

5. Add a link to return to the main blogs page

```
<a href = '/blog'> return to all blogs </a>
```

Adding Forms to blog

1. Create a `forms.py` file inside the blog app
   - this subclassess the base Form class
   - the CharField typcially renders as a HTML `input` element
   - `widget = forms.Textarea` overrides this and renders as an HTML `textarea` element
   - email validation is done on anything with `EmailField()`
   - Django Form Fields documentation

```
from django import forms

class EmailPostForm(forms.Form):
   name = forms.CharField(max_length = 25)
   email = forms.EmailField()
   to = forms.EmailField()
   comments = forms.CharField(required = False, widget =
forms.Textarea)
```

2. Create a view for the form
   - add the `EmailPostForm` import to `views.py`

```
from .forms import EmailPostForm
```

- Add the `post_share` view
- it has both `request` & `post_id` as parameters
- `get_object_or_404` verifies that post has `published status`
- the same view is used for initial blank forms as well as forms with submitted data
- a `GET` request indicates an empty form has to be displayed
- a `POST` request indicates that valid form data has been submitted for the form to process
- `request.method = POST` distinguishes between these twp scenarios

Sending emails with Django

1. Django will write emails to the console if this is added to `settings.py`

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

2. To use the SMTP server for gmail, add the following with a valid gmail account

   - IMPORTANT !! You can hide this info from tracking this sensitive info in github by going into the directory and issuing the following command to halt tracking changes on settings.py
   - TLS is a cryptographic protocol that provides end-to-end security of data sent between applications over the Internet.

```
git update-index --assume-unchanged settings.py
```

   NOTE - This will restore tracking changes !

```
git update-index --no-assume-unchanged settings.py
```

```
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'valid_gmail_account@gmail.com'
EMAIL_HOST_PASSWORD = 'password for the account'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

3. Modify `views.py`

- import send_mail

```
from django.core.mail import send_mail
```

- modify `post_share` in `views.py`
  - A URI (Uniform Resource Identifier) is a string that refers to a resource such as a URL
  - get_absolute_url() method to tell Django how to calculate the canonical URL for an object. To callers, this method should appear to return a string that can be used to refer to the object over HTTP.
  - an example of cd is `cd is {'name': 'ME', 'email': 'sktestdjango@gmail.com', 'to': 'sktestdjango@gmail.com', 'comments': 'Some comment'}`

```
def post_share(request, post_id):
    # Retrieve post by ID
    post = get_object_or_404(Post, id = post_id, status =
"published")
    sent = False

    if request.method == 'POST':
        # form was submitted with data
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Form fields passed validation
            cd = form.cleaned_data
            # ... send email
            post_url =
request.build_absolute_uri(post.get_absolute_url())
            subject = f"{cd['name']} recommends you read " f"
{post.title}"
            message = f"Read {post.title} at {post_url} \n\n"
f"{cd['name']}\'s comments: {cd['comments']}"
            send_mail(subject, message,
'sktestdjango@gmail.com', [cd['to']])
            sent = True

    else:  # show blank form
        form = EmailPostForm()

    context = {'post': post, 'form': form, 'sent': sent}

    return render(request, 'blog/post/share.html', context)
```

4. Add the path in `/blogs/urls.py`

```
urlpatterns = [
    # post views
    # path('', views.post_list, name = 'post_list'),
    path('', views.PostListView.as_view(), name = 'post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
views.post_detail, name = 'post_detail'),
    path('<int:post_id>/share/', views.post_share, name = 'post_share'),
]
```

5. Create the `share` template inside `blog/post`

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

{% block title %} Share a post {% endblock %}

{% block content %}
    {% if sent %}
        <h1> E-mail succesfully sent </h1>
        <p>
            "{{ post.title}}" was succesfully sent to {{
form.cleaned_data.to}}.
        </p>
    {% else %}
        <h1> Share "{{ post.title }}" by e-mail </h1>
        <form method = "post">
        <!-- Example data that is looped in
        cd is {'name': 'ME', 'email': 'sktestdjango@gmail.com', 'to':
'sktestdjango@gmail.com', 'comments': 'DEBUG test AGAIN'}  -->

            {% for field in form%}
                <div>
                    {{ field.errors }}
                    {{ field.label_tag }} {{ field }}
                </div>
            {% endfor %}
            {% csrf_token %}
            <input type = "submit" value = "Send e-mail">
        </form>
    {% endif %}

{% endblock %}
```

Comment functionality

1. Add a model for storing comments
    - The `ForeignKey` associates one `Post` to many `Commments`
    - this is a `one-to-many` relationship
    - the `related_name` attribute allows retrieval all of a post's comments using `post.comments.all()`
    - If `related_name` was not defined, Django would use `comment_set` instead
    - Generally, `related_name` is the name to use for the relation from the related object back to this one
    - the `active` attribute allows for comments to be turned off(e.g. hidden)

    ```
    class Comment(models.Model):
        post = models.ForeignKey(Post, on_delete=models.CASCADE,
    related_name='comments')
        name = models.CharField(max_length=80)
        email = models.EmailField()
        body = models.TextField()
        created = models.DateTimeField(auto_now_add = True)
        updated = models.DateTimeField(auto_now=True)
        active = models.BooleanField(default = True)

        class Meta:  # just a class container with some options
    (metadata)
           ordering: ('created',)

        def __str__(self):
           return f'Comment by {self.name} on {self.post}'
    ```

2. Create a new migration in terminal of the virtual environment

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

3. Run the migration

    ```
    python manage.py migrate
    ```

4. Register model with the admin interface in `admin.py`

- include the `Comment` import
- add the custom Model

```
@admin.register(Comment)
class CommentAdmin(admin.ModelAdmin):
   list_display = ('name', 'email', 'post', 'created', 'active' )
   list_filter = ('active', 'created', 'updated')
   search_fields = ('name', 'email', 'body')
```

5. Modify `forms.py` to allow dynamically built forms from `Comment` model
   - include the `Comment` import
   - add the following class

```
class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ('name', 'email', 'body')
```

6. Modify the `post_detail` view
   - import the `Comment` model and `CommentForm`

```
from .models import Post, Comment
from .forms import EmailPostForm, CommentForm
```

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                             slug = post,
                             status = 'published',
                             publish__year = year,
                             publish__month = month,
                             publish__day = day)



    # list of active comments for this post
    comments = post.comments.filter(active = True)

    new_comment = None
```

```
    if request.method == 'POST':
        # A comment was posted
        comment_form = CommentForm( data=request.POST )

        if comment_form.is_valid():
            # create comment obj but do not save to database yet
            new_comment = comment_form.save(commit = False)
            # Assign current post to comment
            new_comment.post = post
            # Save the comment to the database
            new_comment.save()

    else:  # provide blank comment form
        comment_form = CommentForm()


    context = {'post': post, 'comments': comments,
               'new_comment': new_comment, 'comment_form':
comment_form }

    return render(request, 'blog/post/detail.html', context)
```

7. Add comments to `post_detail` template content block

-
```
{% with comments.count as total_comments %}
    <h2>
        {{ total_comments }} comment {{ total_comments|pluralize }}
    </h2>
{% endwith %}

{% for comment in comments %}
        <div class = "comment">
            <p clss = "info">
                Comment {{ forloop.counter }} by {{ comment.name }}
            </p>
            {{ comment.body|linebreaks }}
        </div>
    {% empty %}
        <p> There are no comments yet </p>
    {% endfor %}
{% if new_comment %}
    <h2> Your comment has been added </h2>
{% else %}
    <h2> Add a new comment </h2>
    <form method = 'post'>
        {{ comment_form.as_p }}
        {% csrf_token %}
        <p>
```

```
                <input type = "submit" value = "Add Comment">
            </p>
        </form>


    {% endif %}
```

8. Move `return to all blogs` link to below `Share this post link`

```
    <p>
        <a href = "{% url "blog:post_share" post.id %}"> Share this post
<a/>
    </p>

    <p>
        <a href = '/blog'> return to all blogs </a>
    </p>
```

Add tagging functionality

1. Utilize the 3rd party app `django-taggit`
    - from virtual environment in terminal run

    ```
    pip install django_taggit
    ```

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

```
...
INSTALLED_APPS = [
    'blog.apps.BlogConfig',
    'taggit',
    'django.contrib.admin',
...
```

3. Add `taggit` to `models.py`
    - import taggit

    ```
    from taggit.managers import TaggableManager
    ```

- append to the `Post` model
- the `tags` manager allows adding, retreiving, & removal of tags from `Post` objects

```
tags = TaggableManager()
```

4. Create a migration for the changes to `model.py`

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

5. Run migration

```
python manage.py migrate
```

6. Modify the `list` template to display tags
   - the `join` template filter
   - Django template filters

```
...
{{ post.title }}

<p class = 'tags'> Tags: {{ post.tags.all|join:", " }}</p>
...
```

```
```

7. Modify `views.py` to allow listing of posts with a specific tag
   - import `Tag` model

```
from taggit.models import Tag
```

   - modify `post_list` view to filter posts by tag

```
```

```
def post_list(request, tag_slug = None):
    object_list = Post.published.all()

    tag = None

    if tag_slug:
        tag = get_object_or_404(Tag, slug = tag_slug)
        object_list = object_list.filter(tags__in = [tag])
```

- include `tags` in the `context`

```
context = {'page': page, 'posts': posts, 'tag': tag}
```

8. Modify `urls.py`

- `name` allows calling the same view with and without parameters

```
urlpatterns = [
    # post views
    path('', views.post_list, name = 'post_list'),
    # path('', views.PostListView.as_view(), name = 'post_list'),
    path('tag/<slug:tag_slug>/', views.post_list, name =
'post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
views.post_detail, name = 'post_detail'),
    path('<int:post_id>/share/', views.post_share, name =
'post_share'),
]
```

9. Modify the `list` template

- before

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

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

{% block content %}
    <h1> My Blog! </h1>

    {% for post in posts %}
        <h2>
            <a href = "{{ post.get_absolute_url }}">
```

```
                    {{ post.title }}
                </a>
            </h2>

            <p class = "date">
                Published {{ post.publish }} by {{ post.author }}
            </p>

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

        {% endfor %}

        <!--  {% include "pagination.html" with page=posts %} -->
        {% include "pagination.html" with page=page_obj %}
    {%endblock%}
```

- after

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

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

    {% block content %}
        <h1> My Blog! </h1>

        {% if tag %}
            <h2> posts tagged with "{{ tag.name }}" </h2>
        {% endif %}

        {% for post in posts %}
            <h2>
                <a href = "{{ post.get_absolute_url }}">
                    {{ post.title }}
                </a>
            </h2>

            <p class = "tags">
                Tags:
                {% for tag in post.tags.all %}
                    <a href = "{% url "blog:post_list_by_tag"
tag.slug %}">
                        {{ tag.name }}
                    </a>
                    {% if not forloop.last %}, {% endif %}
                {% endfor %}
            </p>
            <p class = "date">
                Published {{ post.publish }} by {{ post.author }}
```

```
            </p>

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

        {% endfor %}

        {% include "pagination.html" with page=posts %}
        <!-- {% include "pagination.html" with page=page_obj %} -->
    {%endblock%}
```

Retrieve similar posts

1. Modify `views.py`
    - add `Count` import

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

    - add the following to the bottom of the `post_detail` function
    - the last four aggregated posts are sliced using the calculated field `-same_tags`

    ```
        post_tags_ids = post.tags.values_list('id', flat = True)
        similar_posts =
    Post.published.filter(tags__in=post_tags_ids).exclude(id=post.id)
        similar_posts =
    similar_posts.annotate(same_tags=Count('tags')).order_by('-
    same_tags', '-publish')[:4]

        context = {'post': post, 'comments': comments,
                'new_comment': new_comment, 'comment_form':
    comment_form,
                'similar_posts': similar_posts }

        return render(request, 'blog/post/detail.html', context)
    ```

2. Modify the `detail` template to show posts that are similar

```
    ...
    {{post.body|linebreaks}}

        <p>
            <a href = "{% url "blog:post_share" post.id %}"> Share this
```

```
post </a>
        </p>

        <h2> Similar Posts </h2>
        {% for post in similar_posts %}
            <p>
                <a href = "{{ post.get_absolute_url }}"> {{post.title}}
</a>
            </p>
        {% empty %}
            There are no similar posts yet
        {% endfor %}

        <p>
            <a href = '/blog'> return to all blogs </a>
        </p>
    ...
```