Table of Contents

1. Prerequisites

Before we begin, ensure you have the following installed and set up on your machine:

- **Python 3.6+**: Download Python
- **pip**: Python package installer (comes with Python)
- **Virtual Environment Tool**: venv or `virtualenv`
- **Git**: Download Git
- **Basic Knowledge of Python and Django**: Familiarity with Python syntax and Django framework basics.

2. Setting Up the Development Environment

It's best practice to create a virtual environment for your Django projects to manage dependencies.

Step-by-Step:

1. **Create a Project Directory:**

```
mkdir instagram_clone
cd instagram_clone
```

2. **Create a Virtual Environment:** Using venv:

```
python3 -m venv env
```

3. **Activate the Virtual Environment:**
    a. **On macOS/Linux:**

```
source env/bin/activate
```

b. **On Windows:**

```
env\Scripts\activate
```

4. **Upgrade pip:**

```
pip install --upgrade pip
```

5. **Install Django and Other Dependencies:**

```
pip install django pillow
```

a. `django`: The web framework
b. `pillow`: For image processing

3. Starting a New Django Project

We'll start by creating a new Django project and an app within it.

Step-by-Step:

1. **Create Django Project:**

```
django-admin startproject instagram_clone_project .
```

The dot (`.`) ensures the project is created in the current directory.

2. **Create a Django App:** We'll create an app named `users` to handle user-related functionality.

```
python manage.py startapp users
```

3. **Register the App:** In `instagram_clone_project/settings.py`, add `'users'` to the `INSTALLED_APPS` list:

```
INSTALLED_APPS = [
    # ...
    'users',
]
```

4. **Setup Static and Media Files:** Also in `settings.py`, add configurations for static and media files:

```
import os

STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')

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

5. **Configure URLs:** In `instagram_clone_project/urls.py`, set up URL configurations:

```
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('users.urls')),
]

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

4. Creating the User Authentication System

Implementing user authentication is crucial. We'll use Django's built-in authentication system.

Step-by-Step:

1. **Create URLs for the `users` App:** Create `users/urls.py`:

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

urlpatterns = [
    path('register/', views.register, name='register'),
    path('login/', views.user_login, name='login'),
    path('logout/', views.user_logout, name='logout'),
]
```

2. **Create Views for Authentication:** In `users/views.py`:

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.forms import UserCreationForm,
AuthenticationForm

def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
```

```
        if form.is_valid():
            form.save()
            username = form.cleaned_data.get('username')
            password = form.cleaned_data.get('password1')
            user = authenticate(username=username, password=password)
            login(request, user)
            return redirect('feed')
    else:
        form = UserCreationForm()
    return render(request, 'users/register.html', {'form': form})

def user_login(request):
    if request.method == 'POST':
        form = AuthenticationForm(request, data=request.POST)
        if form.is_valid():
            username = form.cleaned_data.get('username')
            password = form.cleaned_data.get('password')
            user = authenticate(username=username, password=password)
            if user is not None:
                login(request, user)
                return redirect('feed')
    else:
        form = AuthenticationForm()
    return render(request, 'users/login.html', {'form': form})

def user_logout(request):
    logout(request)
    return redirect('login')
```

Explanation:

a. **register**: Handles user registration using Django's `UserCreationForm`. Upon successful registration, the user is authenticated and logged in.
b. **user_login**: Manages user login using `AuthenticationForm`. If authentication is successful, redirects to the feed.
c. **user_logout**: Logs out the user and redirects to the login page.

3. **Create Templates:** Create a directory `users/templates/users/` and add `register.html` and `login.html`.

   a. **register.html:**

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Register</title>
</head>
<body>
    <h2>Register</h2>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Register</button>
    </form>
    <p>Already have an account? <a href="{% url 'login' %}">Login here</a>.</p>
</body>
</html>
```

b. login.html:

```
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Login</button>
    </form>
    <p>Don't have an account? <a href="{% url 'register' %}">Register
here</a>.</p>
</body>
</html>
```

4. **Create a Home/Feed View:** Let's create a simple feed view to redirect users after login or registration.

a. Update `users/urls.py`:

```
urlpatterns = [
    path('register/', views.register, name='register'),
    path('login/', views.user_login, name='login'),
    path('logout/', views.user_logout, name='logout'),
    path('', views.feed, name='feed'),
]
```

b. Add the Feed View in `users/views.py`:

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

@login_required
def feed(request):
    return render(request, 'users/feed.html')
```

c. **Create `feed.html`:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Feed</title>
</head>
<body>
    <h2>Welcome to the Feed</h2>
    <p>Hello, {{ request.user.username }}!</p>
    <a href="{% url 'logout' %}">Logout</a>
</body>
</html>
```

5. **Run Migrations and Test:**

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

Visit http://127.0.0.1:8000/register/ to create a new user, then log in to see the feed.

5. Building the User Profile

Users should have profiles where they can see their information and uploaded photos.

Step-by-Step:

1. **Extend the User Model:** We'll use Django's `OneToOneField` to extend the default `User` model with a `Profile` model.

a. **In `users/models.py`:**

```
from django.db import models
from django.contrib.auth.models import User
from PIL import Image

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(blank=True)
    profile_pic = models.ImageField(default='default.jpg',
upload_to='profile_pics')
```

```
    def __str__(self):
        return f'{self.user.username} Profile'

    def save(self, *args, **kwargs):
        super().save(*args, **kwargs)

        img = Image.open(self.profile_pic.path)

        if img.height > 300 or img.width > 300:
            output_size = (300, 300)
            img.thumbnail(output_size)
            img.save(self.profile_pic.path)
```

Explanation:

i. `Profile` model extends `User` with additional fields like `bio` and `profile_pic`.
ii. The `save` method ensures that uploaded profile pictures are resized to a maximum of 300x300 pixels.

2. Create Signals to Automatically Create/Save Profile:

a. In `users/models.py`, add the following at the bottom:

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

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

@receiver(post_save, sender=User)
def save_profile(sender, instance, **kwargs):
    instance.profile.save()
```

Explanation:

i. These signals ensure that whenever a `User` instance is created or saved, the corresponding `Profile` is also created or updated.

3. Register the Profile Model with Admin Site:

a. In `users/admin.py`:

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

admin.site.register(Profile)
```

4. Create URLs and Views for Profile:

a. Update `users/urls.py`:

```
urlpatterns = [
    # ... existing paths
    path('profile/', views.profile, name='profile'),
    path('profile/edit/', views.edit_profile, name='edit_profile'),
]
```

b. **Add Views in `users/views.py`:**

```
from django.contrib.auth.decorators import login_required
from .forms import UserUpdateForm, ProfileUpdateForm

@login_required
def profile(request):
    return render(request, 'users/profile.html')

@login_required
def edit_profile(request):
    if request.method == 'POST':
        u_form = UserUpdateForm(request.POST, instance=request.user)
        p_form = ProfileUpdateForm(request.POST,
                                   request.FILES,
                                   instance=request.user.profile)
        if u_form.is_valid() and p_form.is_valid():
            u_form.save()
            p_form.save()
            return redirect('profile')
    else:
        u_form = UserUpdateForm(instance=request.user)
        p_form = ProfileUpdateForm(instance=request.user.profile)

    context = {
        'u_form': u_form,
        'p_form': p_form
    }

    return render(request, 'users/edit_profile.html', context)
```

5. **Create Forms for Profile Update:**

a. **Create `users/forms.py`:**

```
from django import forms
from django.contrib.auth.models import User
from .models import Profile

class UserUpdateForm(forms.ModelForm):
    email = forms.EmailField()
```

```
    class Meta:
        model = User
        fields = ['username', 'email']

class ProfileUpdateForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['bio', 'profile_pic']
```

6. **Create Templates for Profile:**

   a. **profile.html:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Profile</title>
</head>
<body>
    <h2>{{ request.user.username }}'s Profile</h2>
    <img src="{{ request.user.profile.profile_pic.url }}" alt="Profile Picture"
width="150">
    <p><strong>Email:</strong> {{ request.user.email }}</p>
    <p><strong>Bio:</strong> {{ request.user.profile.bio }}</p>
    <a href="{% url 'edit_profile' %}">Edit Profile</a>
    <br>
    <a href="{% url 'feed' %}">Back to Feed</a>
    <br>
    <a href="{% url 'logout' %}">Logout</a>
</body>
</html>
```

   b. **edit_profile.html:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Edit Profile</title>
</head>
<body>
    <h2>Edit Profile</h2>
    <form method="post" enctype="multipart/form-data">
        {% csrf_token %}
        {{ u_form.as_p }}
        {{ p_form.as_p }}
        <button type="submit">Update</button>
```

```
    </form>
    <a href="{% url 'profile' %}">Cancel</a>
</body>
</html>
```

7. **Apply Migrations and Test:**

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

a. Register a new user, then navigate to `/profile/` to view the profile.
b. Edit the profile by navigating to `/profile/edit/`.

6. Implementing Photo Uploads

Users should be able to upload photos, similar to Instagram posts.

Step-by-Step:

1. **Create a `Post` Model:**
    a. **In `users/models.py`:**

```
class Post(models.Model):
    author = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='posts')
    image = models.ImageField(upload_to='post_images')
    caption = models.TextField(blank=True)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f'Post by {self.author.username} on
{self.created_at.strftime("%Y-%m-%d")}'

    class Meta:
        ordering = ['-created_at']
```

Explanation:

i. `Post` contains an image, caption, author (user), and timestamp.
ii. Posts are ordered by the creation date in descending order.

2. **Register the `Post` Model in Admin:**
    a. **In `users/admin.py`:**

```
from .models import Profile, Post

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

3. **Create a Form for Post Creation:**

a. **In `users/forms.py`:**

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

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['image', 'caption']
```

4. **Create Views for Creating and Viewing Posts:**
    a. **In `users/views.py`:**

```
from .forms import PostForm
from .models import Post

@login_required
def create_post(request):
    if request.method == 'POST':
        form = PostForm(request.POST, request.FILES)
        if form.is_valid():
            new_post = form.save(commit=False)
            new_post.author = request.user
            new_post.save()
            return redirect('feed')
    else:
        form = PostForm()
    return render(request, 'users/create_post.html', {'form': form})

@login_required
def feed(request):
    posts = Post.objects.all()
    return render(request, 'users/feed.html', {'posts': posts})
```

5. **Update URLs:**
    a. **In `users/urls.py`:**

```
urlpatterns = [
    # ... existing paths
    path('post/new/', views.create_post, name='create_post'),
    path('', views.feed, name='feed'),
]
```

6. **Update the Feed Template to Display Posts:**
    a. **In `feed.html`:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Feed</title>
</head>
<body>
    <h2>Welcome to the Feed</h2>
    <p>Hello, {{ request.user.username }}!</p>
    <a href="{% url 'create_post' %}">Create New Post</a> |
    <a href="{% url 'profile' %}">Profile</a> |
    <a href="{% url 'logout' %}">Logout</a>
    <hr>
    {% for post in posts %}
        <div>
            <p><strong>{{ post.author.username }}</strong> -
{{ post.created_at }}</p>
            <img src="{{ post.image.url }}" alt="Post Image" width="300">
            <p>{{ post.caption }}</p>
            <hr>
        </div>
    {% empty %}
        <p>No posts available.</p>
    {% endfor %}
</body>
</html>
```

7. **Create Template for Creating Posts:**
    a. **create_post.html:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Create Post</title>
</head>
<body>
    <h2>Create a New Post</h2>
    <form method="post" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Post</button>
    </form>
    <a href="{% url 'feed' %}">Back to Feed</a>
</body>
</html>
```

8. **Apply Migrations and Test:**

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

a. Log in, navigate to `/post/new/`, upload an image with a caption, and verify it appears in the feed.

7. Creating the Feed

The feed displays posts from all users in reverse chronological order. We've already set up a basic feed in the previous step. Let's enhance it by adding pagination and likes.

Step-by-Step:

1. **Enhance the Feed View with Pagination:**
   a. **In `users/views.py`:**

```
from django.core.paginator import Paginator

@login_required
def feed(request):
    post_list = Post.objects.all()
    paginator = Paginator(post_list, 10)  # Show 10 posts per page

    page_number = request.GET.get('page')
    posts = paginator.get_page(page_number)
    return render(request, 'users/feed.html', {'posts': posts})
```

2. **Update the Feed Template for Pagination:**
   a. **In `feed.html`:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Feed</title>
</head>
<body>
    <h2>Welcome to the Feed</h2>
    <p>Hello, {{ request.user.username }}!</p>
    <a href="{% url 'create_post' %}">Create New Post</a> |
    <a href="{% url 'profile' %}">Profile</a> |
    <a href="{% url 'logout' %}">Logout</a>
    <hr>
    {% for post in posts %}
        <div>
            <p><strong>{{ post.author.username }}</strong> -
{{ post.created_at }}</p>
```

```
            <img src="{{ post.image.url }}" alt="Post Image" width="300">
            <p>{{ post.caption }}</p>
            <hr>
        </div>
    {% empty %}
        <p>No posts available.</p>
    {% endfor %}

    <div>
        <span class="step-links">
            {% if posts.has_previous %}
                <a href="?page=1">&laquo; first</a>
                <a href="?page={{ posts.previous_page_number }}">previous</a>
            {% endif %}

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

            {% if posts.has_next %}
                <a href="?page={{ posts.next_page_number }}">next</a>
                <a href="?page={{ posts.paginator.num_pages }}">last &raquo;</a>
            {% endif %}
        </span>
    </div>
</body>
</html>
```

3. **Test Pagination:**
    a. Create multiple posts to test if pagination works as expected.

8. Adding Likes Functionality

Users should be able to like and unlike posts.

Step-by-Step:

1. **Create a Like Model:**
    a. **In `users/models.py`:**

```
class Like(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    post = models.ForeignKey(Post, on_delete=models.CASCADE,
related_name='likes')
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        unique_together = ('user', 'post')
```

```
    def __str__(self):
        return f'{self.user.username} likes {self.post.id}'
```

Explanation:

i. `Like` model captures which users liked which posts.
ii. `unique_together` ensures a user can like a post only once.

2. Register the `Like` Model in Admin:

a. In `users/admin.py`:

```
from .models import Profile, Post, Like

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

3. Create URLs and Views for Liking Posts:

a. In `users/urls.py`:

```
urlpatterns = [
    # ... existing paths
    path('post/new/', views.create_post, name='create_post'),
    path('', views.feed, name='feed'),
    path('like/<int:post_id>/', views.like_post, name='like_post'),
]
```

b. Add the `like_post` View in `users/views.py`:

```
from django.shortcuts import get_object_or_404
from django.http import JsonResponse

def like_post(request, post_id):
    post = get_object_or_404(Post, id=post_id)
    liked = False
    like, created = Like.objects.get_or_create(user=request.user, post=post)
    if not created:
        like.delete()
    else:
        liked = True
    return JsonResponse({'liked': liked, 'total_likes': post.likes.count()})
```

Explanation:

i. The `like_post` view toggles the like status. If the user has already liked the post, it unlikes; otherwise, it likes the post.
ii. Returns a JSON response with the new like status and total likes.

4. Modify the Feed Template to Include Like Buttons:

a. **In feed.html:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Feed</title>
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
</head>
<body>
    <h2>Welcome to the Feed</h2>
    <p>Hello, {{ request.user.username }}!</p>
    <a href="{% url 'create_post' %}">Create New Post</a> |
    <a href="{% url 'profile' %}">Profile</a> |
    <a href="{% url 'logout' %}">Logout</a>
    <hr>
    {% for post in posts %}
        <div>
            <p><strong>{{ post.author.username }}</strong> -
{{ post.created_at }}</p>
            <img src="{{ post.image.url }}" alt="Post Image" width="300">
            <p>{{ post.caption }}</p>
            <button class="like-button" data-post-id="{{ post.id }}">
                {% if user in post.likes.all.values_list('user', flat=True) %}
                    Unlike
                {% else %}
                    Like
                {% endif %}
            </button>
            <span id="like-count-{{ post.id }}">{{ post.likes.count }}</span>
Likes
            <hr>
        </div>
    {% empty %}
        <p>No posts available.</p>
    {% endfor %}

    <div>
        <span class="step-links">
            {% if posts.has_previous %}
                <a href="?page=1">&laquo; first</a>
                <a href="?page={{ posts.previous_page_number }}">previous</a>
            {% endif %}

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

```
            {% if posts.has_next %}
                <a href="?page={{ posts.next_page_number }}">next</a>
                <a href="?page={{ posts.paginator.num_pages }}">last &raquo;</a>
            {% endif %}
        </span>
    </div>

    <script>
        $(document).ready(function(){
            $('.like-button').click(function(){
                var btn = $(this);
                var postId = btn.data('post-id');
                $.ajax({
                    url: '/like/' + postId + '/',
                    type: 'GET',
                    success: function(response){
                        if(response.liked){
                            btn.text('Unlike');
                        } else {
                            btn.text('Like');
                        }
                        $('#like-count-' + postId).text(response.total_likes);
                    }
                });
            });
        });
    </script>
</body>
</html>
```

Explanation:

i. Includes a "Like" button for each post.
ii. Uses jQuery to send an AJAX request when the button is clicked.
iii. Updates the button text and like count based on the response.

5. Apply Migrations and Test:

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

a. Log in and like/unlike posts to see the like counts update dynamically.

9. Implementing Comments

Users should be able to comment on posts.

Step-by-Step:

1. **Create a Comment Model:**
    a. **In `users/models.py`:**

```
class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE,
related_name='comments')
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    text = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

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

2. **Register the Comment Model in Admin:**
    a. **In `users/admin.py`:**

```
from .models import Profile, Post, Like, Comment

admin.site.register(Profile)
admin.site.register(Post)
admin.site.register(Like)
admin.site.register(Comment)
```

3. **Create a Form for Comments:**
    a. **In `users/forms.py`:**

```
class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ['text']
        widgets = {
            'text': forms.Textarea(attrs={'rows': 2, 'placeholder': 'Add a
comment...'}),
        }
```

4. **Create Views for Adding Comments:**
    a. **In `users/views.py`:**

```
def add_comment(request, post_id):
    post = get_object_or_404(Post, id=post_id)
    if request.method == 'POST':
        form = CommentForm(request.POST)
        if form.is_valid():
            comment = form.save(commit=False)
            comment.author = request.user
            comment.post = post
```

```
            comment.save()
            return redirect('feed')
    else:
        form = CommentForm()
    return render(request, 'users/add_comment.html', {'form': form})
```

5. **Update URLs:**
    a. **In `users/urls.py`:**

```
urlpatterns = [
    # ... existing paths
    path('post/new/', views.create_post, name='create_post'),
    path('', views.feed, name='feed'),
    path('like/<int:post_id>/', views.like_post, name='like_post'),
    path('post/<int:post_id>/comment/', views.add_comment, name='add_comment'),
]
```

6. **Modify the Feed Template to Display and Add Comments:**
    a. **In `feed.html`:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Feed</title>
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
</head>
<body>
    <h2>Welcome to the Feed</h2>
    <p>Hello, {{ request.user.username }}!</p>
    <a href="{% url 'create_post' %}">Create New Post</a> |
    <a href="{% url 'profile' %}">Profile</a> |
    <a href="{% url 'logout' %}">Logout</a>
    <hr>
    {% for post in posts %}
        <div>
            <p><strong>{{ post.author.username }}</strong> -
{{ post.created_at }}</p>
            <img src="{{ post.image.url }}" alt="Post Image" width="300">
            <p>{{ post.caption }}</p>
            <button class="like-button" data-post-id="{{ post.id }}">
                {% if user in post.likes.all.values_list('user', flat=True) %}
                    Unlike
                {% else %}
                    Like
                {% endif %}
            </button>
```

```
            <span id="like-count-{{ post.id }}">{{ post.likes.count }}</span>
Likes
            <hr>
            <h4>Comments:</h4>
            {% for comment in post.comments.all %}
                <p><strong>{{ comment.author.username }}</strong>:
{{ comment.text }}</p>
            {% empty %}
                <p>No comments yet.</p>
            {% endfor %}
            <form action="{% url 'add_comment' post.id %}" method="post">
                {% csrf_token %}
                {{ form.as_p }}
                <button type="submit">Add Comment</button>
            </form>
            <hr>
        </div>
    {% empty %}
        <p>No posts available.</p>
    {% endfor %}

    <div>
        <span class="step-links">
            {% if posts.has_previous %}
                <a href="?page=1">&laquo; first</a>
                <a href="?page={{ posts.previous_page_number }}">previous</a>
            {% endif %}

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

            {% if posts.has_next %}
                <a href="?page={{ posts.next_page_number }}">next</a>
                <a href="?page={{ posts.paginator.num_pages }}">last &raquo;</a>
            {% endif %}
        </span>
    </div>

    <script>
        // Like button AJAX functionality as before
        $(document).ready(function(){
            $('.like-button').click(function(){
                var btn = $(this);
```

```
                var postId = btn.data('post-id');
                $.ajax({
                    url: '/like/' + postId + '/',
                    type: 'GET',
                    success: function(response){
                        if(response.liked){
                            btn.text('Unlike');
                        } else {
                            btn.text('Like');
                        }
                        $('#like-count-' + postId).text(response.total_likes);
                    }
                });
            });
        });
    </script>
</body>
</html>
```

Explanation:

i. Displays existing comments under each post.
ii. Includes a form to add new comments.
iii. The `CommentForm` needs to be passed to the template context.

7. Update the Feed View to Include Comment Form:

a. In `users/views.py`:

```
@login_required
def feed(request):
    post_list = Post.objects.all()
    paginator = Paginator(post_list, 10)  # Show 10 posts per page

    page_number = request.GET.get('page')
    posts = paginator.get_page(page_number)
    form = CommentForm()
    return render(request, 'users/feed.html', {'posts': posts, 'form': form})
```

8. Apply Migrations and Test:

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

a. Log in, view a post, add comments, and ensure they display correctly.

10. Enhancing with Follow System

To allow users to follow others, we'll implement a follow system.

Step-by-Step:

1. **Create a `Follow` Model:**
    a. **In `users/models.py`:**

```
class Follow(models.Model):
    follower = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='following')
    following = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='followers')
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        unique_together = ('follower', 'following')

    def __str__(self):
        return f'{self.follower.username} follows {self.following.username}'
```

2. **Register the `Follow` Model in Admin:**
    a. **In `users/admin.py`:**

```
from .models import Profile, Post, Like, Comment, Follow

admin.site.register(Profile)
admin.site.register(Post)
admin.site.register(Like)
admin.site.register(Comment)
admin.site.register(Follow)
```

3. **Create URLs and Views for Following/Unfollowing Users:**
    a. **In `users/urls.py`:**

```
urlpatterns = [
    # ... existing paths
    path('post/new/', views.create_post, name='create_post'),
    path('', views.feed, name='feed'),
    path('like/<int:post_id>/', views.like_post, name='like_post'),
    path('post/<int:post_id>/comment/', views.add_comment, name='add_comment'),
    path('user/<int:user_id>/follow/', views.follow_user, name='follow_user'),
]
```

    b. **Add the `follow_user` View in `users/views.py`:**

```
def follow_user(request, user_id):
    target_user = get_object_or_404(User, id=user_id)
    if target_user != request.user:
        follow, created = Follow.objects.get_or_create(follower=request.user,
following=target_user)
```

```
        if not created:
            follow.delete()
    return redirect('profile', user_id=target_user.id)
```

4. **Modify the Profile View to Display Follow Options:**

   a. **Update `users/views.py`:**

```
def profile(request, user_id=None):
    if user_id:
        user = get_object_or_404(User, id=user_id)
    else:
        user = request.user
    is_following = False
    if request.user != user:
        is_following = Follow.objects.filter(follower=request.user,
following=user).exists()
    posts = user.posts.all()
    return render(request, 'users/profile.html', {'profile_user': user,
'is_following': is_following, 'posts': posts})
```

   b. **Update `users/urls.py` to accept `user_id` parameter:**

```
urlpatterns = [
    # ... existing paths
    path('profile/', views.profile, name='profile'),
    path('profile/<int:user_id>/', views.profile, name='profile'),
    # ... other paths
]
```

   c. **Update `profile.html`:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Profile</title>
</head>
<body>
    <h2>{{ profile_user.username }}'s Profile</h2>
    <img src="{{ profile_user.profile.profile_pic.url }}" alt="Profile Picture"
width="150">
    <p><strong>Email:</strong> {{ profile_user.email }}</p>
    <p><strong>Bio:</strong> {{ profile_user.profile.bio }}</p>
    {% if request.user != profile_user %}
        <form action="{% url 'follow_user' profile_user.id %}" method="post">
            {% csrf_token %}
            {% if is_following %}
                <button type="submit">Unfollow</button>
            {% else %}
```

```
                <button type="submit">Follow</button>
            {% endif %}
        </form>
    {% endif %}
    <h3>Posts:</h3>
    {% for post in posts %}
        <div>
            <p>{{ post.caption }}</p>
            <img src="{{ post.image.url }}" alt="Post Image" width="200">
        </div>
    {% empty %}
        <p>No posts yet.</p>
    {% endfor %}
    <a href="{% url 'feed' %}">Back to Feed</a>
    <br>
    <a href="{% url 'logout' %}">Logout</a>
</body>
</html>
```

5. **Add Followers Count to Profile:**
    a. **Modify `profile.html` to show follower and following counts:**

```
<p><strong>Followers:</strong> {{ profile_user.followers.count }}</p>
<p><strong>Following:</strong> {{ profile_user.following.count }}</p>
```

6. **Apply Migrations and Test:**

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

11. Implementing Search Functionality

To allow users to search for other users and posts, we'll implement a search bar prominently in the navigation area. Users can search by usernames to find other profiles or search post captions to find specific photos.

Step-by-Step:

11.1. Updating URLs

First, we'll add new URL patterns to handle search requests.

- **In `users/urls.py`:**

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

urlpatterns = [
```

```
    # ... existing paths
    path('search/', views.search, name='search'),
]
```

11.2. Creating the Search View

We'll create a view that handles search queries and returns relevant users and posts.

- **In `users/views.py`:**

```
from django.db.models import Q

@login_required
def search(request):
    query = request.GET.get('q', '')
    users = User.objects.filter(username__icontains=query) if query
else []
    posts = Post.objects.filter(Q(caption__icontains=query) |
Q(author__username__icontains=query)) if query else []
    return render(request, 'users/search.html', {'query': query,
'users': users, 'posts': posts})
```

Explanation:

- **`query = request.GET.get('q', '')`:** Retrieves the search query from the GET parameters. If no query is provided, defaults to an empty string.
- **`User.objects.filter(username__icontains=query)`:** Searches for users whose usernames contain the query string, case-insensitive.
- **`Post.objects.filter(Q(caption__icontains=query) | Q(author__username__icontains=query))`:** Searches for posts where the caption contains the query or the author's username contains the query.
- **`return render(...)`:** Renders the `search.html` template with the search results.

11.3. Updating the Base Template with a Search Bar

To make the search bar accessible from all pages, we'll add it to a base template.

- **Create `users/templates/users/base.html`:**

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Instagram Clone{% endblock %}</title>
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/bootstrap/5.3.0/css/boots
trap.min.css">
</head>
```

```
<body>
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <div class="container-fluid">
            <a class="navbar-brand" href="{% url
'feed' %}">InstaClone</a>
            <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target="#navbarSupportedContent"
                    aria-controls="navbarSupportedContent" aria-
expanded="false" aria-label="Toggle navigation">
                <span class="navbar-toggler-icon"></span>
            </button>
            <div class="collapse navbar-collapse"
id="navbarSupportedContent">
                <form class="d-flex ms-auto" method="get" action="{%
url 'search' %}">
                    <input class="form-control me-2" type="search"
placeholder="Search" aria-label="Search" name="q"
value="{{ request.GET.q }}">
                    <button class="btn btn-outline-success"
type="submit">Search</button>
                </form>
                <ul class="navbar-nav ms-3">
                    <li class="nav-item">
                        <a class="nav-link" href="{% url
'create_post' %}">New Post</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{% url
'profile' %}">Profile</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{% url
'logout' %}">Logout</a>
                    </li>
                </ul>
            </div>
        </div>
    </nav>
```

```
    <div class="container mt-4">
        {% block content %}
        {% endblock %}
    </div>

    <script
src="https://cdnjs.cloudflare.com/ajax/libs/bootstrap/5.3.0/js/bootstr
ap.bundle.min.js"></script>
</body>
</html>
```

Explanation:

- **Bootstrap Integration:** Utilizes Bootstrap for responsive and aesthetic design.
- **Navigation Bar (<nav>):** Includes links to the feed, new post creation, profile, and logout.
- **Search Form:** A form within the navbar that sends a GET request to the `search` URL with the query parameter q.

11.4. Updating Other Templates to Extend the Base Template

Ensure all templates extend the base template to include the search bar.

- **For example, update `feed.html`:**

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

{% block title %}Feed{% endblock %}

{% block content %}
<h2>Welcome to the Feed</h2>
<p>Hello, {{ request.user.username }}!</p>
<hr>
{% for post in posts %}
    <div class="mb-4">
        <p><strong>{{ post.author.username }}</strong> -
{{ post.created_at }}</p>
        <img src="{{ post.image.url }}" alt="Post Image" class="img-
fluid">
        <p>{{ post.caption }}</p>
        <button class="btn btn-primary like-button" data-post-
id="{{ post.id }}">
            {% if user in post.likes.all.values_list('user',
flat=True) %}
                Unlike
```

```
            {% else %}
                Like
            {% endif %}
        </button>
        <span id="like-count-
{{ post.id }}">{{ post.likes.count }}</span> Likes
        <hr>
        <h4>Comments:</h4>
        {% for comment in post.comments.all %}
            <p><strong>{{ comment.author.username }}</strong>:
{{ comment.text }}</p>
        {% empty %}
            <p>No comments yet.</p>
        {% endfor %}
        <form action="{% url 'add_comment' post.id %}" method="post">
            {% csrf_token %}
            {{ form.as_p }}
            <button type="submit" class="btn btn-secondary">Add
Comment</button>
        </form>
        <hr>
    </div>
{% empty %}
    <p>No posts available.</p>
{% endfor %}

<div>
    <span class="step-links">
        {% if posts.has_previous %}
            <a href="?page=1">&laquo; first</a>
            <a
href="?page={{ posts.previous_page_number }}">previous</a>
        {% endif %}

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

```
        {% if posts.has_next %}
            <a href="?page={{ posts.next_page_number }}">next</a>
            <a href="?page={{ posts.paginator.num_pages }}">last
&raquo;</a>
        {% endif %}
    </span>
</div>

<script>
    // Like button AJAX functionality
    document.addEventListener('DOMContentLoaded', function(){
        document.querySelectorAll('.like-
button').forEach(function(button){
            button.addEventListener('click', function(){
                var btn = this;
                var postId = btn.getAttribute('data-post-id');
                fetch('/like/' + postId + '/')
                    .then(response => response.json())
                    .then(data => {
                        if(data.liked){
                            btn.textContent = 'Unlike';
                        } else {
                            btn.textContent = 'Like';
                        }
                        document.getElementById('like-count-' +
postId).textContent = data.total_likes;
                    });
            });
        });
    });
</script>
{% endblock %}
```

Explanation:

- **Template Inheritance (`{% extends 'users/base.html' %}`):** Ensures the feed page includes the navbar and search bar.
- **`{% block content %}`:** Contains the main content of the feed, including posts and their respective like buttons, comments, etc.
- **AJAX for Like Buttons:** Enhances user interaction by allowing likes/unlikes without page reloads.

11.5. Creating the Search Template

- **Create `users/templates/users/search.html`:**

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

{% block title %}Search Results{% endblock %}

{% block content %}
<h2>Search Results for "{{ query }}"</h2>

<h3>Users</h3>
{% if users %}
    <ul class="list-group">
        {% for user in users %}
            <li class="list-group-item">
                <a href="{% url 'profile'
user.id %}">{{ user.username }}</a>
            </li>
        {% endfor %}
    </ul>
{% else %}
    <p>No users found.</p>
{% endif %}

<h3 class="mt-4">Posts</h3>
{% if posts %}
    {% for post in posts %}
        <div class="mb-4">
            <p><strong>{{ post.author.username }}</strong> -
{{ post.created_at }}</p>
            <img src="{{ post.image.url }}" alt="Post Image"
class="img-fluid">
            <p>{{ post.caption }}</p>
            <a href="{% url 'feed' %}#post-{{ post.id }}">View
Post</a>
            <hr>
        </div>
    {% endfor %}
{% else %}
    <p>No posts found.</p>
```

```
{% endif %}
{% endblock %}
```

Explanation:

- **Displays Search Results:** Shows lists of users and posts that match the search query.
- **Conditional Rendering (`{% if users %}`, `{% if posts %}`):** Ensures content is only displayed if there are matching results.

11.6. Enhancing the Profile View URL

To accommodate profiles of other users, update the profile URL to accept an optional `user_id` parameter.

- **In `users/urls.py`:**

```
urlpatterns = [
    # ... existing paths
    path('profile/', views.profile, name='profile'),
    path('profile/<int:user_id>/', views.profile, name='profile'),
    # ... other paths
]
```

11.7. Applying Migrations and Testing

After adding new views and templates, ensure you apply any necessary migrations.

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

Testing Steps:

1. **Accessing the Feed:** Visit http://127.0.0.1:8000/ and ensure you can see the feed with existing posts.
2. **Using the Search Bar:**
   a. **Searching Users:** Enter a username or partial name to find user profiles.
   b. **Searching Posts:** Enter keywords present in post captions or usernames of authors to find relevant posts.
3. **Viewing Search Results:** Click on user links to view their profiles or navigate to specific posts.

13. Detailed Explanations of Methods and Functions

Understanding the underlying methods and functions in your Django application is crucial for effective development and maintenance. Below is a comprehensive breakdown of the key components used throughout the project.

13.1. Models

13.1.1. User Model

- **`django.contrib.auth.models.User`:**
    - **Purpose:** Represents a user in the Django authentication system.
    - **Fields:** `username`, `password`, `email`, `first_name`, `last_name`, etc.
    - **Usage:** Fundamental for user authentication, permissions, and profile management.

13.1.2. Profile Model

- **Definition:**

```
class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(blank=True)
    profile_pic = models.ImageField(default='default.jpg',
upload_to='profile_pics')
```

- **Fields:**
    - **`user`:** Links each profile to a unique `User` instance using a `OneToOneField`.
    - **`bio`:** Stores user biography information. `blank=True` allows this field to be optional.
    - **`profile_pic`:** Stores the profile picture. Images are uploaded to the `profile_pics` directory, and a default image is used if none is provided.
- **Methods:**
    - **`__str__(self)`:** Returns a human-readable representation of the object, here showing the username.
    - **`save(self, *args, **kwargs)`:** Overrides the default save method to resize images to a maximum of 300x300 pixels using Pillow for optimization.
- **Signals:**

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

@receiver(post_save, sender=User)
def save_profile(sender, instance, **kwargs):
    instance.profile.save()
```

    - **Purpose:** Automatically creates and saves a `Profile` instance whenever a `User` instance is created or saved.
    - **`@receiver(post_save, sender=User)`:** Decorator that connects the following function to the `post_save` signal of the `User` model.
    - **`create_profile`:** Creates a `Profile` when a new `User` is created.
    - **`save_profile`:** Saves the related `Profile` whenever the `User` is saved.

13.1.3. Post Model

- **Definition:**

```
class Post(models.Model):
    author = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='posts')
    image = models.ImageField(upload_to='post_images')
    caption = models.TextField(blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
```

- **Fields:**
  - **author:** Links the post to the `User` who created it via a `ForeignKey`, enabling multiple posts per user.
  - **image:** Stores the uploaded image for the post in the `post_images` directory.
  - **caption:** Stores the caption for the post. Optional due to `blank=True`.
  - **created_at:** Automatically records the timestamp when the post is created.
- **Meta Class:**

```
class Meta:
    ordering = ['-created_at']
```

  - **Purpose:** Orders posts in the database by `created_at` in descending order, ensuring the newest posts appear first.
- **Methods:**
  - **__str__(self):** Provides a readable representation, indicating the author and creation date of the post.

13.1.4. Like Model

- **Definition:**

```
class Like(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    post = models.ForeignKey(Post, on_delete=models.CASCADE,
related_name='likes')
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        unique_together = ('user', 'post')
```

- **Fields:**
  - **user:** References the `User` who liked the post.
  - **post:** References the `Post` that was liked.
- **Meta Class:**

```
class Meta:
    unique_together = ('user', 'post')
```

        - **Purpose:** Ensures a user can like a specific post only once by enforcing a unique relationship between the `user` and `post`.
    - **Methods:**
        - **`__str__(self)`:** Returns a readable string indicating which user liked which post.

13.1.5. Comment Model

- **Definition:**

```
class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE,
related_name='comments')
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    text = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
```

- **Fields:**
    - **`post`:** Links the comment to the associated post.
    - **`author`:** References the `User` who made the comment.
    - **`text`:** Contains the comment text.
    - **`created_at`:** Automatically records when the comment was created.
- **Methods:**
    - **`__str__(self)`:** Provides a readable representation indicating the comment's author and associated post.

13.1.6. Follow Model

- **Definition:**

```
class Follow(models.Model):
    follower = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='following')
    following = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='followers')
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        unique_together = ('follower', 'following')
```

- **Fields:**
    - **`follower`:** The user who is following another user.
    - **`following`:** The user being followed.

- **Meta Class:**

```
class Meta:
    unique_together = ('follower', 'following')
```

    - **Purpose:** Ensures a user cannot follow the same user more than once.
- **Methods:**
    - **`__str__(self)`:** Returns a string indicating which user follows whom.

13.2. Forms

13.2.1. UserUpdateForm

- **Definition:**

```
class UserUpdateForm(forms.ModelForm):
    email = forms.EmailField()

    class Meta:
        model = User
        fields = ['username', 'email']
```

- **Purpose:** Allows users to update their `username` and `email`.
- **Fields:**
    - **`username`:** From the `User` model.
    - **`email`:** Explicitly added as an `EmailField` for email validation.

13.2.2. ProfileUpdateForm

- **Definition:**

```
class ProfileUpdateForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['bio', 'profile_pic']
```

- **Purpose:** Enables users to update their `bio` and `profile_pic`.

13.2.3. PostForm

- **Definition:**

```
class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['image', 'caption']
```

- **Purpose:** Facilitates the creation of new posts by allowing users to upload an image and add an optional caption.

13.2.4. CommentForm

- **Definition:**

```
class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ['text']
        widgets = {
            'text': forms.Textarea(attrs={'rows': 2, 'placeholder':
'Add a comment...'}),
        }
```

- **Purpose:** Allows users to add comments to posts.
- **Widgets:**
    - **TextArea:** Enhances the comment input with multiple rows and a placeholder for better UX.

13.3. Views

13.3.1. User Authentication Views

13.3.1.1. Register View

- **Definition:**

```
def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            username = form.cleaned_data.get('username')
            password = form.cleaned_data.get('password1')
            user = authenticate(username=username, password=password)
            login(request, user)
            return redirect('feed')
    else:
        form = UserCreationForm()
    return render(request, 'users/register.html', {'form': form})
```

- **Functionality:**
    - **GET Request:** Presents an empty user creation form.
    - **POST Request:** Processes the submitted form data. If valid:
        - Saves the new user.
        - Authenticates the user.
        - Logs the user in.

        - Redirects to the feed page.
- **Functions Used:**
    - **`UserCreationForm`:** Django's built-in form for creating new users.
    - **`authenticate`:** Verifies the user's credentials.
    - **`login`:** Logs the user in by creating a session.

13.3.1.2. User Login View

- **Definition:**

```
def user_login(request):
    if request.method == 'POST':
        form = AuthenticationForm(request, data=request.POST)
        if form.is_valid():
            username = form.cleaned_data.get('username')
            password = form.cleaned_data.get('password')
            user = authenticate(username=username, password=password)
            if user is not None:
                login(request, user)
                return redirect('feed')
    else:
        form = AuthenticationForm()
    return render(request, 'users/login.html', {'form': form})
```

- **Functionality:**
    - **GET Request:** Presents an empty authentication form.
    - **POST Request:** Processes the login data. If credentials are correct:
        - Authenticates the user.
        - Logs the user in.
        - Redirects to the feed.
- **Functions Used:**
    - **`AuthenticationForm`:** Django's built-in form for authenticating users.
    - **`authenticate`:** Validates the provided username and password.
    - **`login`:** Creates a user session.

13.3.1.3. User Logout View

- **Definition:**

```
def user_logout(request):
    logout(request)
    return redirect('login')
```

- **Functionality:**
    - Logs the user out by invalidating the session.
    - Redirects to the login page.

- **Functions Used:**
    - **`logout`:** Django function that removes the authenticated user's ID from the request and flushes the session data.

13.3.2. Profile Views

13.3.2.1. View Profile

- **Definition:**

```
def profile(request, user_id=None):
    if user_id:
        user = get_object_or_404(User, id=user_id)
    else:
        user = request.user
    is_following = False
    if request.user != user:
        is_following = Follow.objects.filter(follower=request.user,
following=user).exists()
    posts = user.posts.all()
    return render(request, 'users/profile.html', {'profile_user':
user, 'is_following': is_following, 'posts': posts})
```

- **Functionality:**
    - **Parameters:**
        - **`user_id`:** Optional parameter. If provided, displays the profile of the specified user.
    - **Logic:**
        - If `user_id` is provided, fetch the corresponding `User`; otherwise, use the currently logged-in user.
        - Determines if the current user is following the profile user.
        - Retrieves all posts authored by the profile user.
    - **Rendering:** Displays the profile with user details, follow/unfollow options, and their posts.
- **Functions Used:**
    - **`get_object_or_404`:** Retrieves an object or returns a 404 error if not found.
    - **`Follow.objects.filter(...).exists()`:** Checks if a follow relationship exists between two users.

13.3.2.2. Edit Profile

- **Definition:**

```
def edit_profile(request):
    if request.method == 'POST':
        u_form = UserUpdateForm(request.POST, instance=request.user)
        p_form = ProfileUpdateForm(request.POST,
                                   request.FILES,
                                   instance=request.user.profile)
```

```
        if u_form.is_valid() and p_form.is_valid():
            u_form.save()
            p_form.save()
            return redirect('profile')
    else:
        u_form = UserUpdateForm(instance=request.user)
        p_form = ProfileUpdateForm(instance=request.user.profile)

    context = {
        'u_form': u_form,
        'p_form': p_form
    }

    return render(request, 'users/edit_profile.html', context)
```

- **Functionality:**
  - **GET Request:** Presents pre-filled forms with the user's current information.
  - **POST Request:** Processes the submitted forms. If valid:
    - Saves the updated user and profile information.
    - Redirects to the profile page.
- **Functions Used:**
  - **`UserUpdateForm`:** Form for updating username and email.
  - **`ProfileUpdateForm`:** Form for updating bio and profile picture.
  - **`save()`:** Commits the form data to the database.

13.3.3. Post Views

13.3.3.1. Create New Post

- **Definition:**

```
def create_post(request):
    if request.method == 'POST':
        form = PostForm(request.POST, request.FILES)
        if form.is_valid():
            new_post = form.save(commit=False)
            new_post.author = request.user
            new_post.save()
            return redirect('feed')
    else:
        form = PostForm()
    return render(request, 'users/create_post.html', {'form': form})
```

- **Functionality:**
    - **GET Request:** Presents an empty form for creating a new post.
    - **POST Request:** Processes the form data. If valid:
        - Assigns the current user as the author.
        - Saves the post to the database.
        - Redirects to the feed.
- **Functions Used:**
    - **`PostForm`:** Form for creating a new post.
    - **`form.save(commit=False)`:** Creates an unsaved `Post` instance, allowing assignment of the author before saving.
    - **`save()`:** Commits the post to the database.

13.3.3.2. Feed View

- **Definition:**

```
from django.core.paginator import Paginator

@login_required
def feed(request):
    post_list = Post.objects.all()
    paginator = Paginator(post_list, 10)  # Show 10 posts per page

    page_number = request.GET.get('page')
    posts = paginator.get_page(page_number)
    form = CommentForm()
    return render(request, 'users/feed.html', {'posts': posts, 'form':
form})
```

- **Functionality:**
    - **Retrieves All Posts:** Fetches all posts from the database.
    - **Pagination:**
        - Uses Django's `Paginator` to split posts into pages, with 10 posts per page.
        - Retrieves the current page number from GET parameters.
        - Gets the corresponding page of posts.
    - **Provides Comment Form:** Passes an empty `CommentForm` to the template for adding new comments.
- **Functions Used:**
    - **`Paginator`:** Handles splitting of querysets into manageable pages.
    - **`get_page`:** Retrieves the specified page; automatically handles invalid page numbers.

13.3.4. Like View

- **Definition:**

```
def like_post(request, post_id):
    post = get_object_or_404(Post, id=post_id)
    liked = False
    like, created = Like.objects.get_or_create(user=request.user,
post=post)
    if not created:
        like.delete()
    else:
        liked = True
    return JsonResponse({'liked': liked, 'total_likes':
post.likes.count()})
```

- **Functionality:**
  - Fetches the post by `post_id`. If not found, returns a 404 error.
  - Attempts to create a `Like` instance for the current user and post.
    - **If created:** The post has been liked. `liked` is set to `True`.
    - **If not created:** The user has already liked the post, so the existing `Like` is deleted (unlike action).
  - Returns a JSON response indicating:
    - **`liked`:** Whether the post is now liked (`True`) or unliked (`False`).
    - **`total_likes`:** The updated total number of likes for the post.
- **Functions Used:**
  - **`get_object_or_404`:** Retrieves the post or returns a 404 error.
  - **`Like.objects.get_or_create(...)`:** Attempts to retrieve an existing `Like` or create a new one if it doesn't exist.
  - **`delete()`:** Removes the `Like` instance from the database.
  - **`JsonResponse`:** Sends a JSON-formatted HTTP response.

13.3.5. Comment View

- **Definition:**

```
def add_comment(request, post_id):
    post = get_object_or_404(Post, id=post_id)
    if request.method == 'POST':
        form = CommentForm(request.POST)
        if form.is_valid():
            comment = form.save(commit=False)
            comment.author = request.user
            comment.post = post
            comment.save()
            return redirect('feed')
    else:
```

```
        form = CommentForm()
    return render(request, 'users/add_comment.html', {'form': form})
```

- **Functionality:**
    - Retrieves the specific post by `post_id`.
    - **GET Request:** Presents an empty comment form (optional; in the current setup, comments are added directly in the feed).
    - **POST Request:** Processes the submitted comment. If valid:
        - Assigns the current user as the comment's author.
        - Assigns the comment to the relevant post.
        - Saves the comment.
        - Redirects back to the feed.
- **Functions Used:**
    - **`get_object_or_404`:** Fetches the post or returns a 404 if not found.
    - **`CommentForm`:** Form for adding new comments.
    - **`save(commit=False)`:** Creates an unsaved `Comment` instance for further assignment before saving.
    - **`save()`:** Commits the comment to the database.

13.3.6. Follow View

- **Definition:**

```
def follow_user(request, user_id):
    target_user = get_object_or_404(User, id=user_id)
    if target_user != request.user:
        follow, created =
Follow.objects.get_or_create(follower=request.user,
following=target_user)
        if not created:
            follow.delete()
    return redirect('profile', user_id=target_user.id)
```

- **Functionality:**
    - Retrieves the target user to follow/unfollow.
    - Prevents users from following themselves.
    - Attempts to create a `Follow` instance indicating that `request.user` is following `target_user`.
        - **If created:** The user is now following the target user.
        - **If not created:** The `Follow` already exists, so it deletes it to unfollow.
    - Redirects back to the target user's profile.
- **Functions Used:**
    - **`get_object_or_404`:** Retrieves the target user or returns a 404 error.

- **`Follow.objects.get_or_create(...)`:** Attempts to establish a follow relationship or remove it if it already exists.
- **`delete()`:** Removes the existing follow relationship.
- **`redirect`:** Redirects the user to the specified URL.

13.3.7. Search View

- **Definition:**

```
@login_required
def search(request):
    query = request.GET.get('q', '')
    users = User.objects.filter(username__icontains=query) if query
else []
    posts = Post.objects.filter(Q(caption__icontains=query) |
Q(author__username__icontains=query)) if query else []
    return render(request, 'users/search.html', {'query': query,
'users': users, 'posts': posts})
```

- **Functionality:**
  - **Retrieves Query:** Extracts the search term from the GET parameters.
  - **Searches Users:**
    - **`username__icontains=query`:** Performs a case-insensitive search for users whose usernames contain the query string.
  - **Searches Posts:**
    - **`caption__icontains=query`:** Finds posts with captions containing the query.
    - **`author__username__icontains=query`:** Finds posts authored by users with usernames containing the query.
  - **Renders Results:** Passes the found users and posts to the `search.html` template.
- **Functions Used:**
  - **`@login_required`:** Decorator ensuring only authenticated users can access the search functionality.
  - **`Q`:** Allows complex queries combining multiple conditions using OR logic.
  - **`filter`:** Retrieves objects matching the given conditions.

13.4. Template Tags and Filters

13.4.1. `{% csrf_token %}`

- **Usage:** Included in HTML forms to provide Cross-Site Request Forgery protection.
- **Purpose:** Generates a hidden form field that contains a token. Django verifies this token on form submission to ensure the request originates from the same site.

13.4.2. `{{ form.as_p }}` and `{{ form.as_p }}`

- **Usage:** Renders Django forms as HTML paragraphs (`<p>` elements).
- **Purpose:** Simplifies form rendering by automatically generating form fields with appropriate labels and inputs.

13.4.3. `{% url 'route_name' %}`

- **Usage:** Generates the URL corresponding to the given route name.
- **Purpose:** Facilitates URL management by avoiding hardcoding URLs in templates. Enhances maintainability, especially when URLs change.

13.4.4. `{% for ... %}` and `{% endfor %}`

- **Usage:** Iterates over a queryset or list to render repeated elements.
- **Purpose:** Dynamically generates HTML content based on database records, such as listing posts or comments.

13.4.5. `{% if ... %}` and `{% endif %}`

- **Usage:** Implements conditional logic in templates.
- **Purpose:** Controls the rendering of HTML elements based on certain conditions, such as displaying the "Unlike" button only if the user has liked the post.

13.4.6. `{{ variable }}`

- **Usage:** Outputs the value of a variable.
- **Purpose:** Displays dynamic content in templates, such as usernames, captions, or image URLs.

13.5. Decorators

13.5.1. `@login_required`

- **Usage:** Placed above view functions to restrict access to authenticated users only.
- **Purpose:** Prevents unauthorized access to certain views, ensuring that only logged-in users can access functionalities like creating posts or liking content.
- **Behavior:** Redirects unauthenticated users to the login page.

13.6. Django's ORM Methods

13.6.1. `objects.all()`

- **Usage:** Retrieves all instances of a model from the database.
- **Example:** `Post.objects.all()`
- **Purpose:** Fetches all records, often used for displaying all posts in the feed.

13.6.2. `objects.filter(...)`

- **Usage:** Retrieves instances that match certain conditions.
- **Example:** `User.objects.filter(username__icontains=query)`
- **Purpose:** Performs database queries to find specific records, such as searching for users or posts.

13.6.3. `objects.get_or_create(...)`

- **Usage:** Attempts to retrieve an object matching the given parameters; if none exists, creates it.
- **Example:** `Like.objects.get_or_create(user=request.user, post=post)`

- **Purpose:** Simplifies operations where you need to ensure a record exists without manually checking for its existence first.

13.6.4. `objects.exists()`

- **Usage:** Checks if any records match the given query.
- **Example:** `Follow.objects.filter(...).exists()`
- **Purpose:** Efficiently determines the existence of records without retrieving them, useful for conditionals.

14. Conclusion

Congratulations! You've successfully extended your Instagram clone by adding robust **search functionality** and gained a deeper understanding of the intricate **methods and functions** that power your Django application. Here's a recap of what you've accomplished:

- **Search Functionality:**
  - Integrated a search bar into the navigation for user and post searches.
  - Created a dedicated search view to handle and process queries.
  - Designed a search template to display results elegantly.
- **Comprehensive Understanding:**
  - Explored Django models, forms, views, and templates in-depth.
  - Learned about Django's ORM methods for efficient database operations.
  - Understood the importance of decorators and template tags in Django.
  - Gained insights into how signals automate tasks like profile creation.

Next Steps

To further enhance your Instagram clone, consider implementing the following features:

1. **Direct Messaging:** Allow users to send private messages to each other.
2. **User Notifications:** Notify users when someone likes or comments on their posts.
3. **Stories:** Implement ephemeral content similar to Instagram Stories.
4. **Hashtags:** Enable users to tag posts with hashtags and search by them.
5. **Pagination Enhancements:** Improve the user experience with infinite scrolling.
6. **Responsive Design:** Ensure the application is mobile-friendly across all devices.
7. **API Integration:** Create RESTful APIs using Django REST Framework to allow for mobile app integration or third-party services.

Resources

- **Django Documentation:** Official Django documentation for comprehensive guidance.
- **Django REST Framework:** Toolkit for building Web APIs in Django.
- **Bootstrap:** Frontend framework for designing responsive websites.
- **Pillow Documentation:** Python Imaging Library for image processing.
- **Heroku Django Deployment:** Guide for deploying Django applications to Heroku.