KAZAKH BRITISH TECHNICAL UNIVERSITY

Assignment 3

Building a post management application using views and templates.

Prepared by Azimkhanov Bauyrzhan

Almaty, 10.11.2024

Table of contents

Introduction

Nowadays, it is very important to use or switch to a microservice architecture. Since it allows you to better scale projects, reduces support costs, etc.

Another very common problem is that you often have to either reboot the application or move/run it in another environment. This can cause many problems, ranging from the need to store the order of application startup somewhere to the difference in the architecture of the hardware on which the application is launched. Most of them can be solved by a tool such as Docker and its derivative Docker compose. Docker allows you to prepare an environment and run it on any platform that supports it. That is, the developer and engineer do not need to constantly look back at these aspects.

Django models allow you to access the DBMS in a very simple format using Python language syntax. This is a very important tool that allows you to automate many queries for interaction with the DBMS, which saves a lot of time for developers.

The Django framework for the Python programming language provides us with additional capabilities for interacting with and/or creating the frontend part of a web application. These tools include:

1. views
2. templates
3. static files
4. media files.

My main motivation for making a task manager using Docker and Django was the desire to master a new stack of relevant technologies and the desire to get a good grade at the end of the semester.

Exercise descriptions

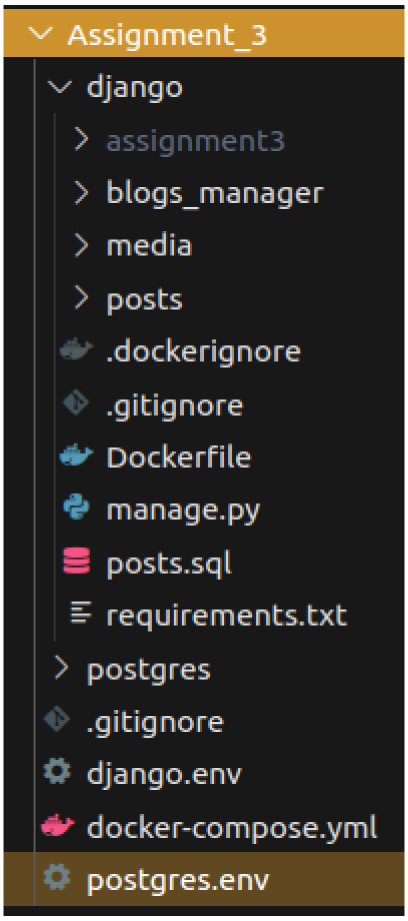
Exercise 1: Creating a Basic Model
Objective:

1. Create a Django app (e.g., blog) and define a model called Post with fields for title, content, author, and published date.
2. Implement a method to return a string representation of the Post model.

Description of the implementation steps:

1. Create app using *python manage.py commands startapp polls* command.
2. Create desired models.
3. Define/override __str__() method to provide string representation of model.
4. Register all new models in admin.py file
5. Make migration.

Expected outcome:



```
Assignment_3
  django
    assignment3
    blogs_manager
    media
    posts
    .dockerignore
    .gitignore
    Dockerfile
    manage.py
    posts.sql
    requirements.txt
  postgres
  .gitignore
  django.env
  docker-compose.yml
  postgres.env
```

Figure 1. Desired project structure

```
class Post(models.Model):
   title = models.CharField(max_length=200)
   content = models.TextField()
   author = models.CharField(max_length=70)
   published_date = models.DateTimeField(auto_now_add=True)

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

Figure 2. Post model

```
from django.contrib import admin
from .models import Post, Comment, Category

admin.site.register(Post)
```

Figure 3. admin.py file

Exercise 2: Model Relationships

Objective:

1. Extend your Post model to include a Category model with a many-to-many relationship.
2. Create a Comment model that has a foreign key relationship with Post.

Description of the implementation steps:

1. Create Category and Comment models.
2. Add new *models.ManyToManyField()* and *models.ForeignKey()* for Post and Comment model correspondingly.
3. Make migration.
4. Register all new models in admin.py file.

Expected outcome:

```python
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200)
    description = models.TextField()

    def __str__(self):
        return self.name

    class Meta:
        verbose_name = "Category"
        verbose_name_plural = "Categories"

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.CharField(max_length=70)
    published_date = models.DateTimeField(auto_now_add=True)
    category = models.ManyToManyField(Category)

    def __str__(self):
        return self.title

    class Meta:
        verbose_name = "Post"
        verbose_name_plural = "Posts"

class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE)
    content = models.TextField()

    def __str__(self):
        return self.post.__str__() + " comment"

    class Meta:
        verbose_name = "Comment"
        verbose_name_plural = "Comments"
```

Figure 4. models.py file with Comment and Category models

```python
from django.contrib import admin
from .models import Post, Comment, Category

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

Figure 5. admin.py file with registered Comment and Category models

Exercise 3: Custom Manager

Objective:

1. Create a custom manager for your Post model that returns only published posts.
2. Implement a method in the manager to get posts by a specific author.

Description of the implementation steps:

1. Create custom manager PostManager.
2. Define 2 methods *get_published_posts()* and *get_posts_from_author()* with specified functionality.

Expected outcome:

```python
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200)
    description = models.TextField()

    def __str__(self):
        return self.name

    class Meta:
        verbose_name = "Category"
        verbose_name_plural = "Categories"

class PostManager(models.Manager):
    def get_published_posts(self):
        return self.filter(published_date__isnull=False)

    def get_posts_from_author(self, author):
        return self.filter(author=author)

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.CharField(max_length=70)
    published_date = models.DateTimeField(auto_now_add=True)
    category = models.ManyToManyField(Category)

    objects = PostManager()

    def __str__(self):
        return self.title

    class Meta:
        verbose_name = "Post"
        verbose_name_plural = "Posts"



class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE)
    content = models.TextField()
    def __str__(self):
        return self.post.__str__() + " comment"

    class Meta:
        verbose_name = "Comment"
        verbose_name_plural = "Comments"
```

Figure 6. Updated models.py with custom PostManager

Exercise 4: Function-Based Views

Objective:

1. Implement a function-based view to list all blog posts.
2. Create a separate view to display a single post using its ID.

Description of the implementation steps:

1. Create FBV function *indexFBV()* in views.py file that will display all post from newest to oldest by *published_date* field.
2. Create FBV function *detailFBV()* in views.py file that will display specified post or 404 error page if there is no such post.
3. Create templates for FBVs index.html and detail.html.
4. Add FBVs to urls.py.

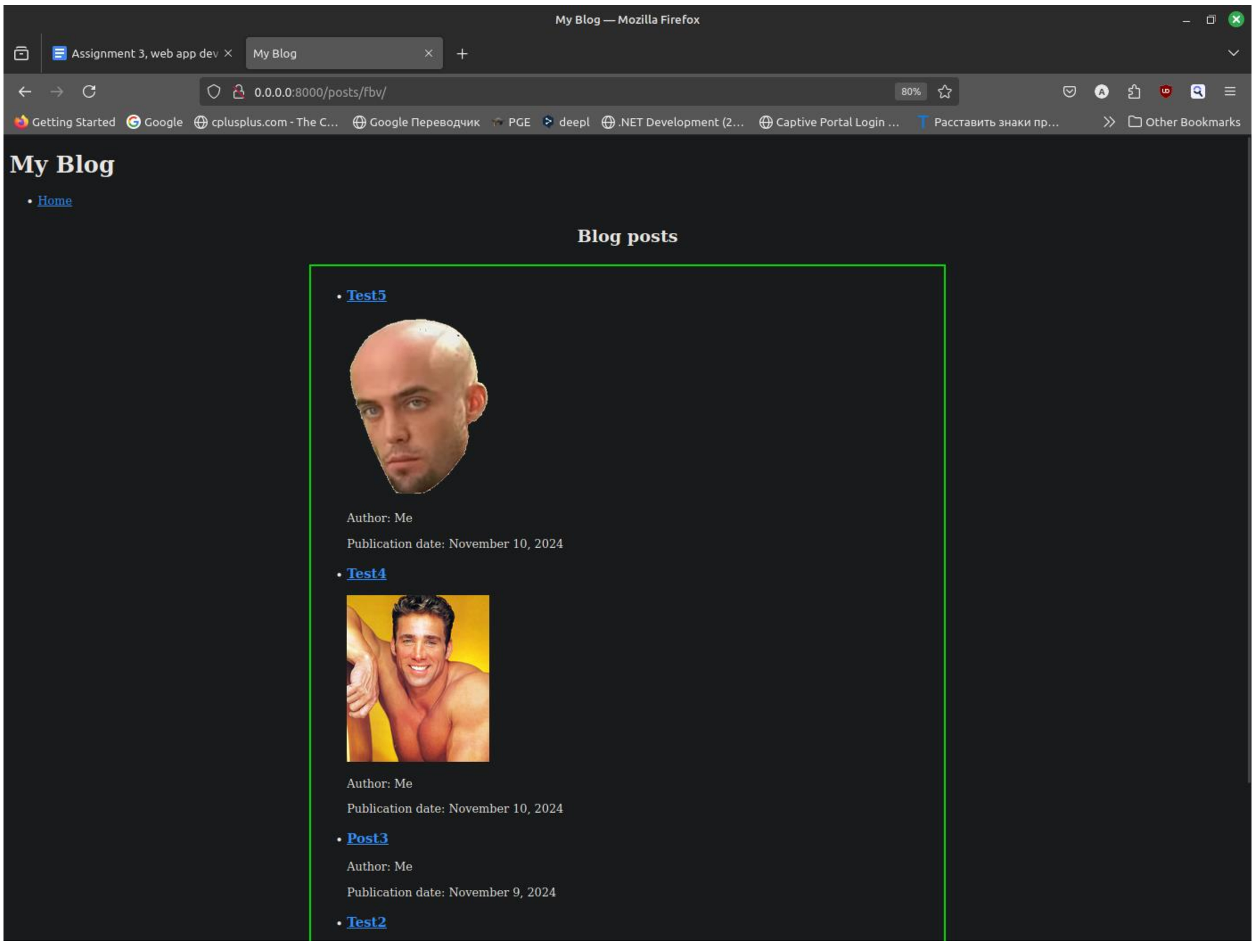
Expected outcome:

```
def indexFBV(request):
    latest_posts_list = Post.objects.order_by("-published_date")
    template = loader.get_template("posts/index.html")
    context = {
        "latest_posts_list": latest_posts_list,
        "sub_domen": "fbv"
    }
    return render(request, "posts/index.html", context)

def detailFBV(request, post_id):
    post = get_object_or_404(Post, pk=post_id)
    context = {
        "post": post,
        "sub_domen": "fbv"
    }
    return render(request, "posts/detail.html", context)
```

Figure 7. FBV views.

```
urlpatterns = [
    # ex: /posts/fbv/
    path("fbv/", views.indexFBV, name="index"),
    # ex: /posts/fbv/5/
    path("fbv/<int:post_id>/", views.detailFBV, name="detail"),
```

Figure 8. urls.py with FBVs



My Blog

- Home

Blog posts

- Test5

Author: Me

Publication date: November 10, 2024

- Test4

Author: Me

Publication date: November 10, 2024

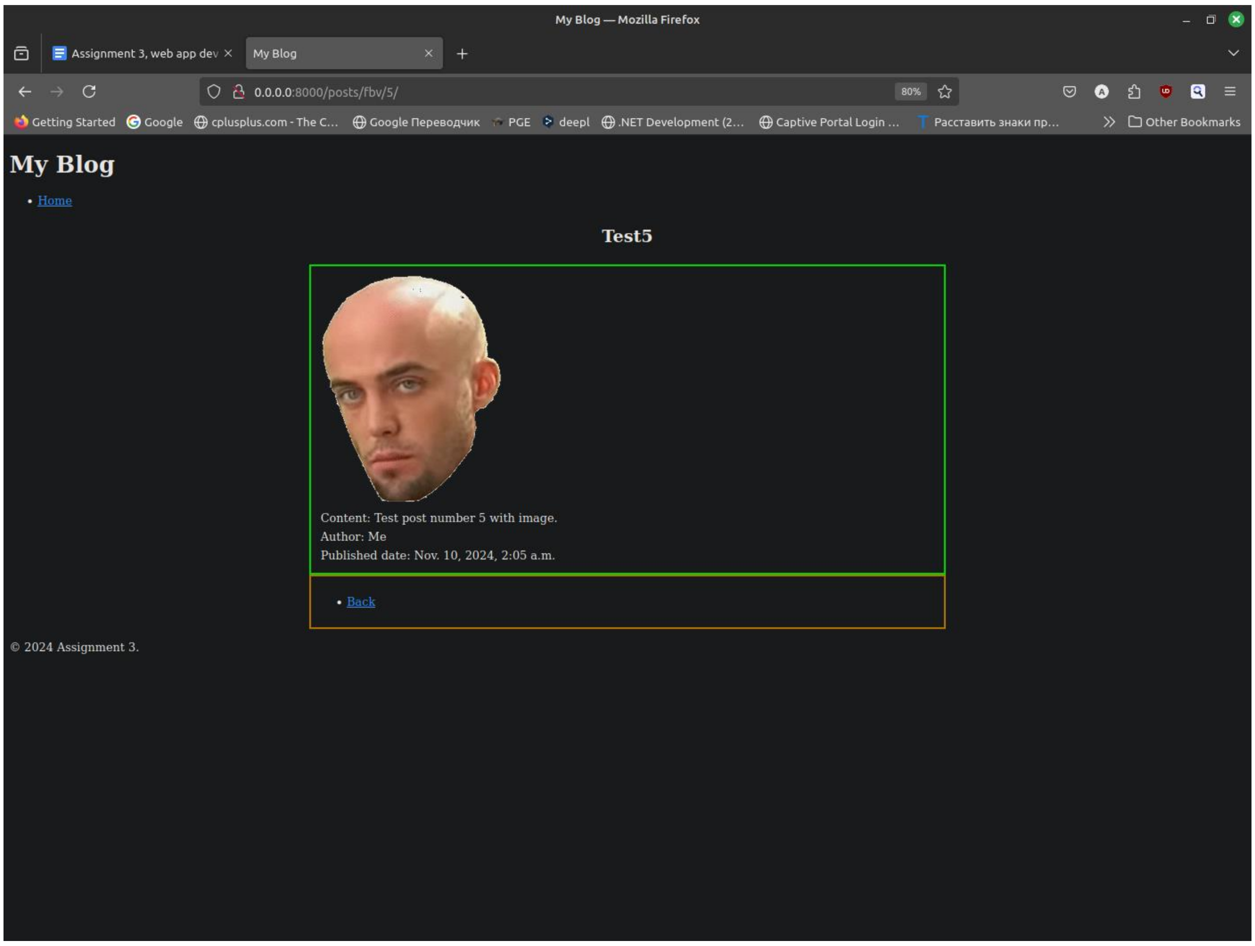
- Post3

Author: Me

Publication date: November 9, 2024

- Test2

Figure 9. index.html using FBV


My Blog — Mozilla Firefox
Assignment 3, web app dev
My Blog
0.0.0.0:8000/posts/fbv/5/

My Blog

- Home

Test5

Content: Test post number 5 with image.
Author: Me
Published date: Nov. 10, 2024, 2:05 a.m.

- Back

© 2024 Assignment 3.

Figure 10. detail.html using FBV

Exercise 5: Class-Based Views

Objective:

1. Refactor the views from Exercise 4 using Django's class-based views (ListView and DetailView).
2. Ensure the templates render correctly for both views.

Description of the implementation steps:

1. Create CBV class *IndexCBV()* in views.py file using *indexFBV()* as example.
2. Create CBV class *DetailCBV()* in views.py file using *detailFBV()* as example.
3. Update urls.py by adding new CBVs.

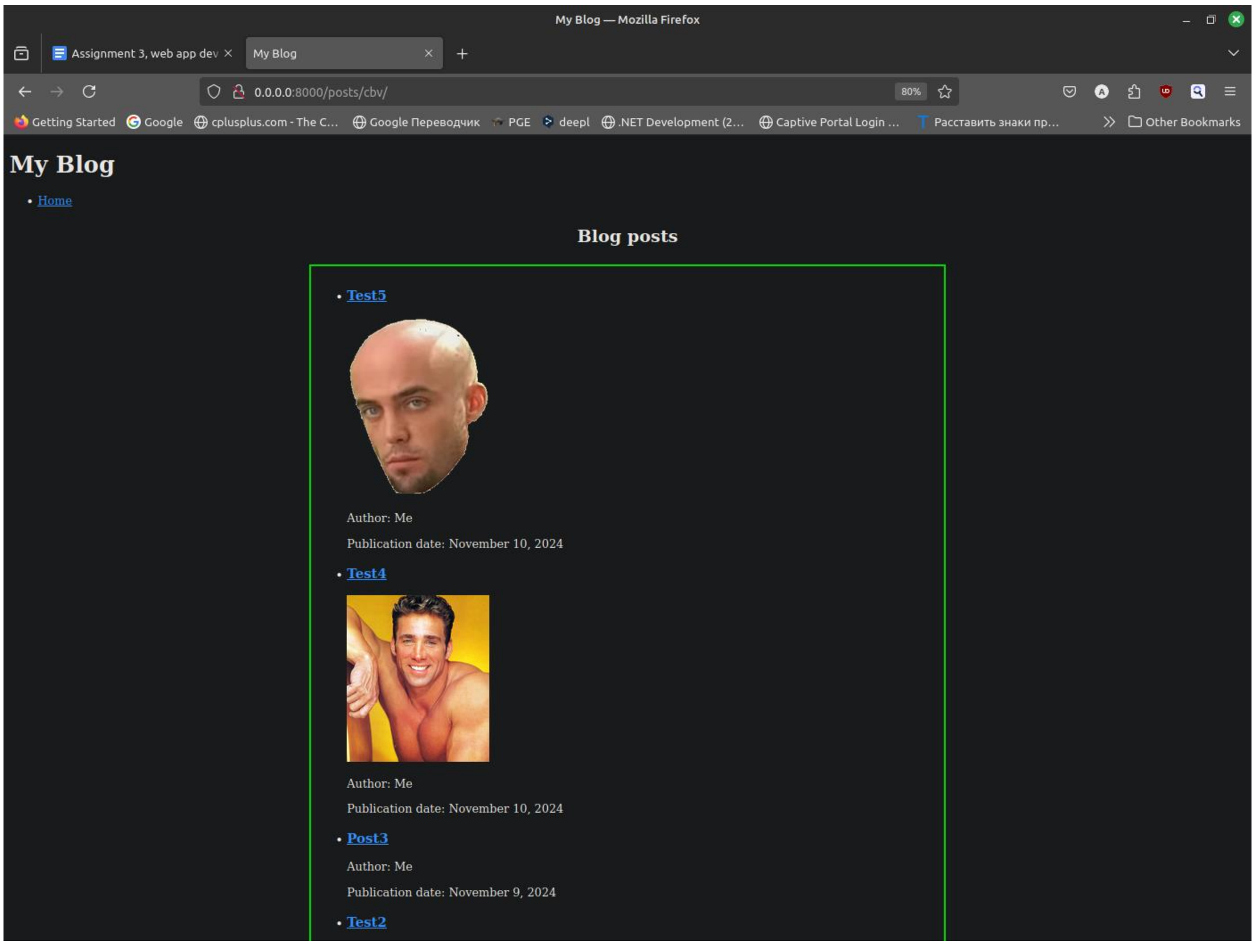
Expected outcome:

```
class IndexCBV(ListView):
    model = Post
    template_name = "posts/index.html"
    context_object_name = "latest_posts_list"
    ordering = ["-published_date"]

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["sub_domen"] = "cbv"
        return context

class DetailCBV(DetailView):
    model = Post
    template_name = "posts/detail.html"
    context_object_name = "post"
    pk_url_kwarg = "post_id"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['sub_domen'] = 'cbv'
        return context
```

Figure 11. CBV views

```
urlpatterns = [
    # ex: /posts/fbv/
    path("fbv/", views.indexFBV, name="index"),
    # ex: /posts/fbv/5/
    path("fbv/<int:post_id>/", views.detailFBV, name="detail"),
    # ex: /posts/cbv/
    path("cbv/", views.IndexCBV.as_view(), name="index"),
    # ex: /posts/cbv/5/
    path("cbv/<int:post_id>/", views.DetailCBV.as_view(), name="detail"),
```

Figure 12. urls.py with CBVs


My Blog — Mozilla Firefox

Assignment 3, web app dev | My Blog

0.0.0.0:8000/posts/cbv/

My Blog

- Home

Blog posts

- Test5

Author: Me

Publication date: November 10, 2024

- Test4

Author: Me

Publication date: November 10, 2024

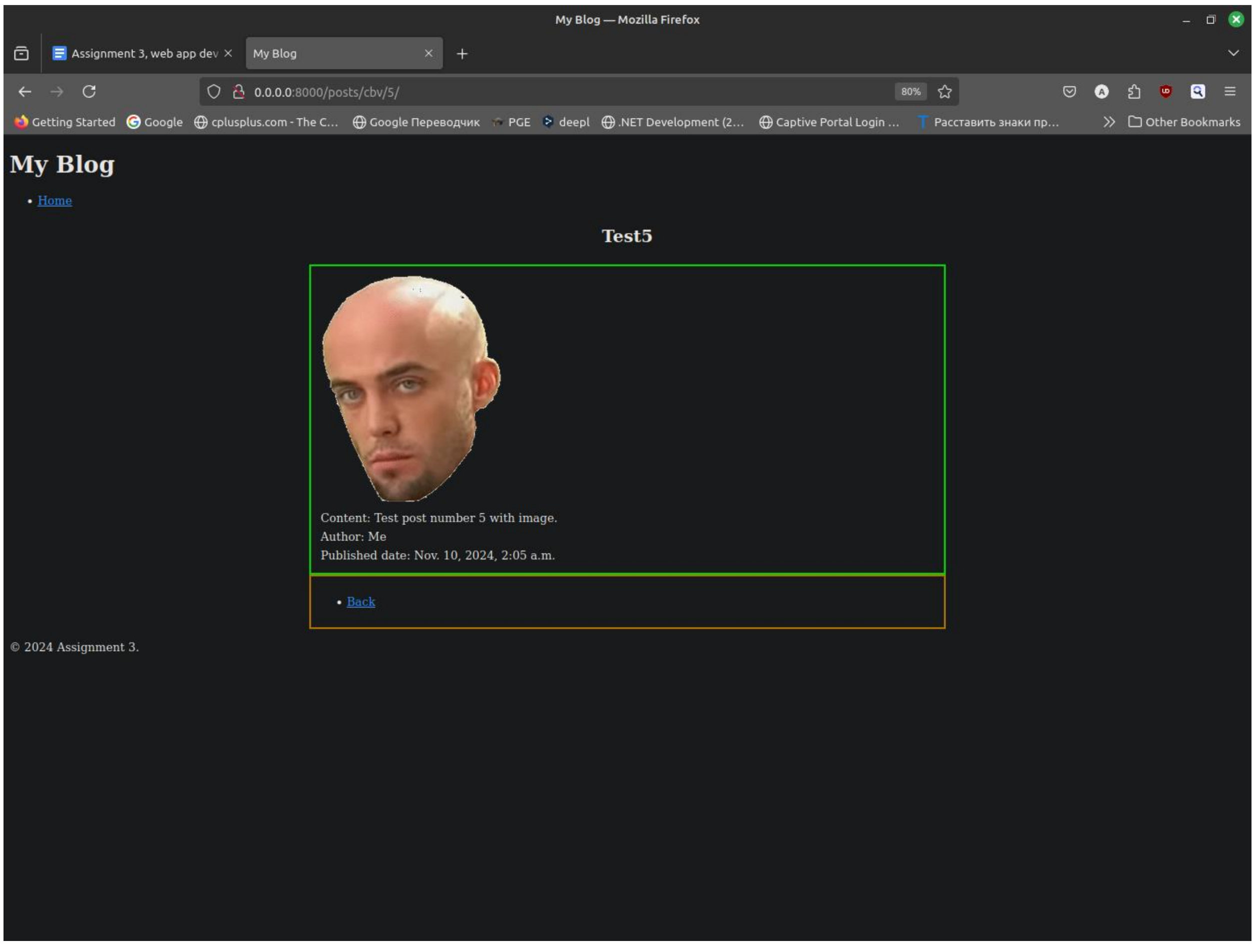
- Post3

Author: Me

Publication date: November 9, 2024

- Test2

Figure 13. index.html using CBV



My Blog

- Home

Test5

Content: Test post number 5 with image.
Author: Me
Published date: Nov. 10, 2024, 2:05 a.m.

- Back

© 2024 Assignment 3.

Figure 14. detail.html using CBV

Exercise 6: Handling Forms

Objective:

1. Create a form to add new blog posts using Django's forms.ModelForm.
2. Implement a view that handles form submission, validates the data, and saves the new post.

Description of the implementation steps:

1. Create forms.py file in post app directory and define *PostForm()* class with fields from Post model.
2. Create *PostCreateCBV()* in views.py. Use FormView as basement for this class.
3. Add new path to urls.py.
4. Create new create_post.html template.

Expected outcome:

```
from django.forms import ModelForm
from . import models

class PostForm(ModelForm):
    class Meta:
        model = models.Post
        fields = ["title", "content", "author", "category", "image"]
```

Figure 15. forms.py file

```python
from django.shortcuts import render, get_object_or_404
from django.urls import reverse_lazy
from django.template import loader
from django.views.generic import ListView, DetailView, FormView

from .forms import PostForm
from .models import Post

def indexFBV(request):
    latest_posts_list = Post.objects.order_by("-published_date")
    template = loader.get_template("posts/index.html")
    context = {
        "latest_posts_list": latest_posts_list,
        "sub_domen": "fbv"
    }
    return render(request, "posts/index.html", context)

def detailFBV(request, post_id):
    post = get_object_or_404(Post, pk=post_id)
    context = {
        "post": post,
        "sub_domen": "fbv"
    }
    return render(request, "posts/detail.html", context)

class IndexCBV(ListView):
    model = Post
    template_name = "posts/index.html"
    context_object_name = "latest_posts_list"
    ordering = ["-published_date"]

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["sub_domen"] = "cbv"
        return context

class DetailCBV(DetailView):
    model = Post
    template_name = "posts/detail.html"
    context_object_name = "post"
    pk_url_kwarg = "post_id"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['sub_domen'] = 'cbv'
        return context

class PostCreateCBV(FormView):
    template_name = "posts/create_post.html"
    form_class = PostForm
    success_url = reverse_lazy("index")
    def form_valid(self, form):
        form.save()
        return super().form_valid(form)
```

Figure 16. Final version of views.py

```
urlpatterns = [
    # ex: /posts/fbv/
    path("fbv/", views.indexFBV, name="index"),
    # ex: /posts/fbv/5/
    path("fbv/<int:post_id>/", views.detailFBV, name="detail"),
    # ex: /posts/cbv/
    path("cbv/", views.IndexCBV.as_view(), name="index"),
    # ex: /posts/cbv/5/
    path("cbv/<int:post_id>/", views.DetailCBV.as_view(), name="detail"),
    # ex: /posts/create/
    path("create/", views.PostCreateCBV.as_view(), name="create_post"),
]
```

Figure 17. urls.py with *PostCreateCBV()*.

Exercise 7: Basic Template Rendering

Objective:

1. Create a template to display the list of posts. Include titles, authors, and publication dates.
2. Use a template tag to format the publication date.

Description of the implementation steps:

1. Adapt index.html to satisfy requirements.

Expected outcome:

```
<h2 style="text-align: center; margin: 1em;">Blog posts</h1>
<div style="margin: auto; width: 50%; border: 3px solid green; padding: 10px;">
    {% if latest_posts_list %}
        <ul>
        {% for post in latest_posts_list %}
            <li>
                <h3><a href="/posts/{{ sub_domen }}/{{ post.id }}/">{{ post.title }}</a></h2>
                {% if post.image %}
                    <img src="{{ post.image.url }}" alt="{{ post.title }}" style="max-width:200px;">
                {% endif %}
                <p>Author: {{ post.author }}</p>
                <p>Publication date: {{ post.published_date|date:"F j, Y" }}</p>
            </li>
        {% endfor %}
        </ul>
    {% else %}
        <p>No posts are available.</p>
    {% endif %}
</div>
```

Figure 18. Adapted index.html

Exercise 8: Template Inheritance

Objective:

1. Create a base template that includes a header and footer.
2. Extend this base template for your list and detail views.

Description of the implementation steps:

1. Create base.html template with footer and footer.
2. Adapt index.html, detail.html, create_post.html templates to extend base.html.

Expected outcome:

```
<!-- templates/base.html -->
<!DOCTYPE html>
<html>
   <head>
      <title>My Blog</title>
      {% load static %}
      <link rel="stylesheet" type="text/css" href="{% static 'posts/css/styles.css' %}">
   </head>
   <body>
      <header>
         <h1>My Blog</h1>
         <nav>
            <ul>
               <li><a href="/posts/{{ sub_domen }}/">Home</a></li>
            </ul>
         </nav>
      </header>

      <div class="content">
         {% block content %}
         {% endblock %}
      </div>

      <footer>
         <p>&copy; 2024 Assignment 3.</p>
      </footer>
   </body>
</html>
```

Figure 19. base.html file

```
<!-- templates/posts/index.html -->
{% extends "posts/base.html" %}

{% block content %}
<h2 style="text-align: center; margin: 1em;">Blog posts</h1>
<div style="margin: auto; width: 50%; border: 3px solid green; padding: 10px;">
  {% if latest_posts_list %}
    <ul>
    {% for post in latest_posts_list %}
      <li>
        <h3><a href="/posts/{{ sub_domen }}/{{ post.id }}/">{{ post.title }}</a></h2>
        {% if post.image %}
          <img src="{{ post.image.url }}" alt="{{ post.title }}" style="max-width:200px;">
        {% endif %}
        <p>Author: {{ post.author }}</p>
        <p>Publication date: {{ post.published_date|date:"F j, Y" }}</p>
      </li>
    {% endfor %}
    </ul>
  {% else %}
    <p>No posts are available.</p>
  {% endif %}
</div>
{% endblock %}
```

Figure 20. index.html final version

```
<!-- templates/posts/detail.html -->
{% extends "posts/base.html" %}

{% block content %}
<h2 style="text-align: center; margin: 1em;">{{ post.title }}</h1>
<div style="margin: auto; width: 50%; border: 3px solid green; padding: 10px;">
  {% if post.image %}
    <img src="{{ post.image.url }}" alt="{{ post.title }}" style="max-width:600px;">
  {% endif %}
  <div style="padding: 0.2em;">Content: {{ post.content }}</div>
  <div style="padding: 0.2em;">Author: {{ post.author }}</div>
  <div style="padding: 0.2em;">Published date: {{ post.published_date }}</div>
</div>
<div style="margin: auto; width: 50%; border: 3px solid orange; padding: 10px;">
  <ul>
    <li><a href="/posts/{{ sub_domen }}/">Back</a></li>
  </ul>
</div>
{% endblock %}
```

Figure 21. detail.html final version

```
<!-- templates/posts/create_post.html -->
{% extends "posts/base.html" %}

{% block content %}
<h2 style="text-align: center; margin: 1em;">Create post</h1>
<div style="margin: auto; width: 50%; border: 3px solid green; padding: 10px;">
   <form method="POST">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit">Create post</button>
   </form>
</div>
{% endblock %}
```

Figure 22. create_post final version

Exercise 9: Static Files and Media

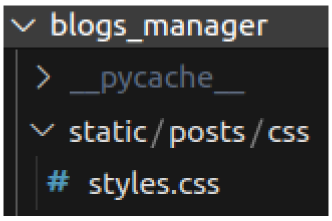
Objective:

1. Add CSS styles to your templates using static files.
2. Set up a media directory to allow users to upload images for their posts and display these images in your templates.

Description of the implementation steps:

1. Create static/posts/css/styles.html file in blogs_manager app.
2. Fill it with CSS.
3. Define STATIC_URL in settings.py file.
4. Create media/post_images directory in project root directory.
5. Define image field in Post model with post_images/ as *upload_to* argument.
6. Make migration.
7. Add static link to urlpatterns in both urls.py for blogs_manager and posts apps.
8. Define MEDIA_ROOT in settings.py file.

Expected outcome:

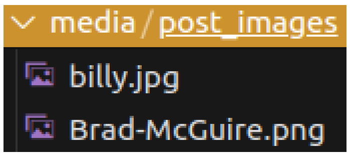


```
∨ blogs_manager
  > __pycache__
  ∨ static / posts / css
    # styles.css
```

Figure 23. static directory in blogs_manager app

```css
/* static/posts/css/styles.css */
body {
   font-family: Arial, sans-serif;
   background-color: #f9f9f9;
   margin: 0;
   padding: 0;
}

header, footer {
   background-color: #333;
   color: white;
   text-align: center;
   padding: 1em 0;
}

.content {
   padding: 2em;
}

h1, h2 {
   color: #333;
}
```

Figure 24. styles.css file



```
media / post_images
  billy.jpg
  Brad-McGuire.png
```

Figure 25. media directory in project root

```python
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200)
    description = models.TextField()

    def __str__(self):
        return self.name

    class Meta:
        verbose_name = "Category"
        verbose_name_plural = "Categories"

class PostManager(models.Manager):
    def get_published_posts(self):
        return self.filter(published_date__isnull=False)

    def get_posts_from_author(self, author):
        return self.filter(author=author)

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.CharField(max_length=70)
    published_date = models.DateTimeField(auto_now_add=True)
    category = models.ManyToManyField(Category)
    image = models.ImageField(upload_to="post_images/", blank=True, null=True)

    objects = PostManager()

    def __str__(self):
        return self.title

    class Meta:
        verbose_name = "Post"
        verbose_name_plural = "Posts"


class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE)
    content = models.TextField()

    def __str__(self):
        return self.post.__str__() + " comment"

    class Meta:
        verbose_name = "Comment"
        verbose_name_plural = "Comments"
```

Figure 26. Final version of models.py

```
urlpatterns = [
   …
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```
Figure 27. Static link to media in both urls.py

```
STATIC_URL = '/static/'

MEDIA_URL = '/media/'

MEDIA_ROOT = BASE_DIR / 'media'
```
Figure 28. Media and static related variables in settings.py

Code snippets

I have already briefly described most of the important points in the previous section.

Here I would like to highlight the part about creating CBVs and why they have an additional sub_domen field in the context object when calling the render function in views.py. The thing is that I did not want to duplicate the html templates for CBV and FBV. And in order for the "back" button and links to each post in index.html to redirect to the desired view, I added an additional sub_domen field to the context object. To do this, I had to override the get_context_data() method in the corresponding IndexCBV and DetailCBV classes.

Next, each template receives a value from the sub_domain field and correctly creates links for transitions.

```python
class DetailCBV(DetailView):
    model = Post
    template_name = "posts/detail.html"
    context_object_name = "post"
    pk_url_kwarg = "post_id"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['sub_domen'] = 'cbv'
        return context
```

Figure 29. DetailCBV with overriden method.

```html
<div style="margin: auto; width: 50%; border: 3px solid orange; padding: 10px;">
    <ul>
        <li><a href="/posts/{{ sub_domen }}/">Back</a></li>
    </ul>
</div>
```

Figure 30. Excerpt from detail.html using the sub_domen field.

Results

As a result, we received an application that meets the requirements and goals of this project. Namely, an application with:

1. templates
2. Class Based Views
3. Function Based Views
4. static files
5. media files
6. and reuse of templates.

The main problems were small blots related to the structure and declaration of directories for storing static and media files. I had to carefully reread the official documentation several times.
Figure 3. Django structure

Conclusion

During my work I received and consolidated knowledge on how to use Django templates, views, static and media files. How to competently redefine and reuse them.

All the above-described tools play a key role in the popularity of this framework, providing high-quality automation of routine processes and significantly reducing the development time and knowledge threshold for writing complex web applications.

References

Links to Django tutorials from official documentation:

1. https://docs.djangoproject.com/en/5.1/intro/tutorial01/
2. https://docs.djangoproject.com/en/5.1/intro/tutorial02/
3. https://docs.djangoproject.com/en/5.1/intro/tutorial03/
4. https://docs.djangoproject.com/en/5.1/intro/tutorial04/
5. https://docs.djangoproject.com/en/5.1/intro/tutorial02/

Link to official Docker documentation:

- https://docs.docker.com/

Link to official Docker image registry where from I took base images for compose:

- https://hub.docker.com/

Links to some topics in StackOverflow:

1. https://stackoverflow.com/questions/13164048/text-box-input-height
2. https://stackoverflow.com/questions/3681627/how-to-update-fields-in-a-model-without-creating-a-new-record-in-django
3. https://stackoverflow.com/questions/42614172/how-to-redirect-from-a-view-to-another-view-in-django
4. https://stackoverflow.com/questions/3805958/how-to-delete-a-record-in-django-models
5. https://stackoverflow.com/questions/14719883/django-can-not-get-a-time-function-timezone-datetime-to-work-properly-gett
6. https://stackoverflow.com/questions/3289601/null-object-in-python
7. https://stackoverflow.com/questions/11336548/how-to-get-post-request-values-in-django
8. https://stackoverflow.com/questions/11714721/how-to-set-the-space-between-lines-in-a-div-without-setting-line-height
9. https://stackoverflow.com/questions/3430432/django-update-table
10. https://stackoverflow.com/questions/15128705/how-to-insert-a-row-of-data-to-a-table-using-djangos-orm
11. https://stackoverflow.com/questions/67155473/how-can-i-get-values-from-a-view-in-django-every-time-i-access-the-home-page
12. https://stackoverflow.com/questions/3500859/django-request-get
13. https://stackoverflow.com/questions/50346326/programmingerror-relation-django-session-does-not-exist
14. https://stackoverflow.com/questions/18713086/virtualenv-wont-activate-on-windows

Link to my GitHub repository (screenshots, logs and SQL structure included):

- https://github.com/BauyrzhanAzimkhanov/Web-application-development-MSc