Kazakh-British Technical University

Web Application Development

Assignment №3

Django Models, Django Views, Django Templates

Full Name: Tamiris Abildayeva

ID: 23MD0503

Link to GitHub: https://github.com/TamirisK/university-web-application-development

10.11.2024

Table of Contents

Introduction

This report summarizes Assignment 3 Developing a simple blog project using Django. The project employs three core components of Django—Models, Views, and Templates—essential for creating dynamic, data-driven websites.

I will detail the implementation steps for the blog project, illustrating how these components work together to create a functional and user-friendly platform. This includes defining the structure of blog posts, handling user requests, and presenting content.

Django Models are fundamental for data handling and database management. They represent database tables, allowing developers to interact with the database using Python, thus avoiding raw SQL. In this project, I created a Post model with fields like title, content, author, and published date. The model also includes relationships with a Category model (many-to-many) and a Comment model. A custom manager was implemented to return only published posts and those by specific authors.

Views in Django process user requests and return appropriate responses. This project utilizes both Function-Based Views and Class-Based Views. Initial function-based views listed blog posts and displayed individual posts, while class-based views like *ListView* and *DetailView* were used for modular code. Views also handle form submissions for new posts with built-in validation.

Templates define the structure of web pages using Django's templating engine, which allows dynamic content within static HTML. In this project, templates were created for post lists, detailed views, and other components. Template inheritance was used to create a reusable base template, reducing redundancy. Additionally, static files for CSS and media were integrated to enhance the user experience.

Django encourages the separation of data handling, request processing, and presentation, enhancing maintainability and scalability:

1. **Models** provide an abstraction for easier database interaction, reducing complexity and allowing for easy evolution of data structures.
2. **Views** manage business logic, enabling the handling of both simple and complex use cases with cleaner, maintainable code.
3. **Templates** separate design from logic, making it easier to maintain and update the user interface while ensuring dynamic and appealing web pages.

Combining these components, Django offers a useful framework for developing scalable web applications like blogs, e-commerce platforms, and content management systems.

Exercise Descriptions

Django Models

This chapter focuses on implementing the Models component in Django, which provides the data structure for a blog application. I will define models for blog posts, categories, comments, and other related data, allowing us to manage interactions with the database.

I will outline the steps for creating these models, establishing relationships between them, and adding custom managers for improved querying.

Exercise 1: Creating a Basic Model

Objective

This exercise aims to create a Django app called "Blog" and to define a core model named *Post*. This model will represent individual blog posts and will include important fields for managing blog content, such as the title, content, author, published date, and image. It will also have a method to return a simple representation of the post, which helps with readability and management in the Django admin interface.

Description of Implementation Steps

1. **Creating the Post Model:**

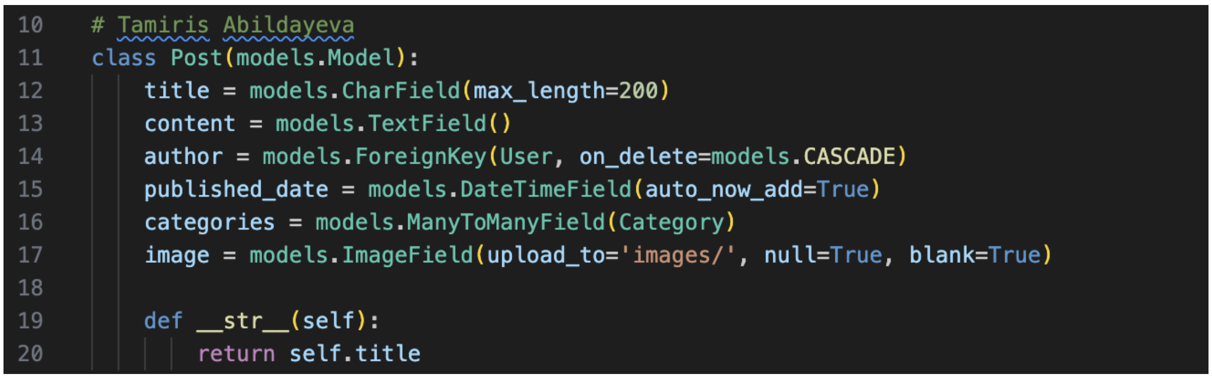
I created the *Post* model within the Django app. This model represents each blog post and includes the following fields:

1. **Title:** A *CharField* with a maximum length of 200 characters, representing the title of the blog post.
2. **Content:** A *TextField* for the main body of the blog post, allowing for long text input.
3. **Author:** A *ForeignKey* relationship with the built-in *User* model, linking the post to the user who wrote it. This establishes a one-to-many relationship between users and their posts.
4. **Published Date:** A *DateTimeField* that automatically records when the post is created. It uses *auto_now_add=True* to fill this field with the current date and time automatically.
5. **Image:** An optional *ImageField* that allows the user to upload an image related to the post. The *upload_to* parameter specifies where to store images. This field can be left blank or set to null if no image is provided.

2. **Defining the __str__ Method:**

I improved the readability of the *Post* model by overriding the *__str__* method. This method returns the title of the post when printed or referenced, making it easier to identify posts in the Django admin interface.

Here is the code for the *Post* model:



```
# Tamiris Abildayeva
class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    published_date = models.DateTimeField(auto_now_add=True)
    categories = models.ManyToManyField(Category)
    image = models.ImageField(upload_to='images/', null=True, blank=True)

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

Fig. 1: Screenshot of the Post model code

In this exercise, I created the Post model, which forms the basis for each blog post in the Django project. It includes essential fields like title, content, author, published date, and an optional image. I also implemented the __*str*__ method to display the post's title in the Django admin interface.

Exercise 2: Model Relationships

Objective

This exercise aims to improve the *Post* model by adding related models. I have created a *Category* model that uses a many-to-many relationship and a *Comment* model that uses a foreign key relationship. This update has helped categorize posts and manage comments better, making a more complete blogging system.

Description of Implementation Steps

1. **Category Model (Many-to-Many Relationship):**

I created a *Category* model to organize blog posts into categories. A category can contain multiple posts, and a post can be in multiple categories. This creates a many-to-many relationship. I used Django's *ManyToManyField* in the *Post* model to link it to the *Category* model. This setup makes it easy to categorize posts by themes or topics.

2. **Comment Model (Foreign Key Relationship):**

I made a *Comment* model to allow users to comment on blog posts. Each comment is connected to a specific post through a foreign key. The *Comment* model also links to the *User* model to keep track of who wrote the comment. This ensures that each comment is connected to a user and a post, making it easy to see who commented on what. The Comment model includes fields for the comment content and the created date.

3. **Code Overview:**

I updated the Post model to include a *ManyToManyField* for the Category model and a *ForeignKey* for the author using Django's User model. It also has a *published_date* field to show when the post was published. The Category model has one field for the category name. The Comment model includes fields for the *post*, *author*, *content*, and *created_date* to show the relationships between the post and the author.

4. **PostManager:**

I created a manager called *PostManager* to handle common queries like getting only published posts and filtering posts by authors. This makes retrieving posts easier and more efficient.

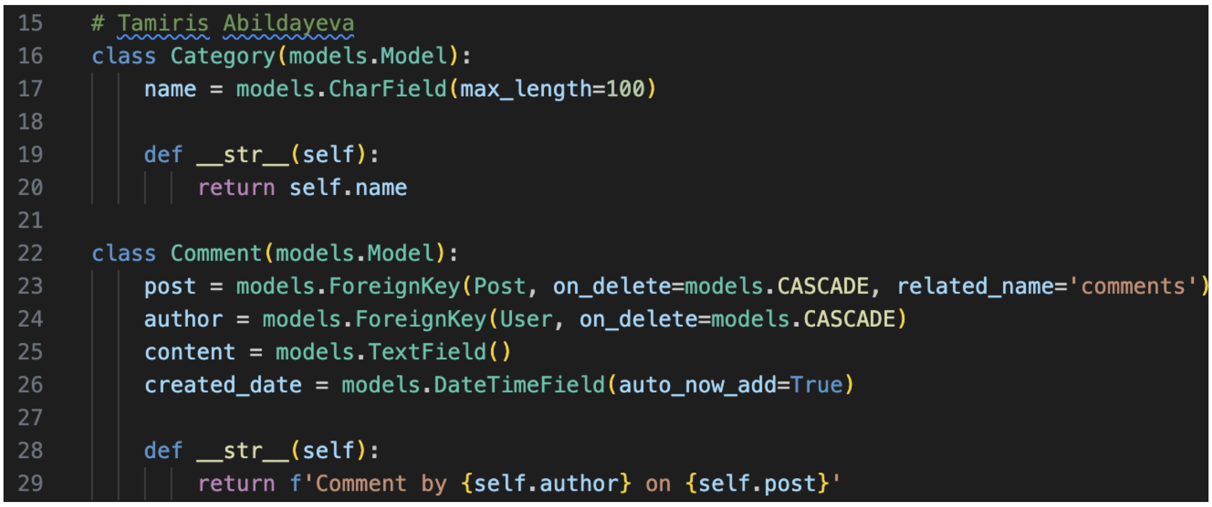
5. **Explanation of Model Relationships:**

   1. **Many-to-Many (Post ↔ Category):**
      The Post model uses ManyToManyField to link multiple categories to a single post. A post can belong to many categories, and a category can hold multiple posts. This structure is flexible for organizing content.

   2. **Foreign Key (Comment ↔ Post):**
      Each Comment links to a specific Post through a ForeignKey, creating a one-to-many relationship. One post can have many comments, but each comment connects to just one post.

   3. **Foreign Key (Comment ↔ User):**
      The Comment model also links to the User model. This shows the relationship between the user who wrote the comment and the comment itself. This makes it clear who authored each comment.

Here is the code for the Category and Comment models:



```python
# Tamiris Abildayeva
class Category(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

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

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

Fig. 2: Screenshot of the Category and Comment models code

In this exercise, I improved the blogging system by establishing relationships between the Post, Category, and Comment models. A many-to-many relationship was created between the Post and Category models, allowing posts to be categorized under multiple themes. I also added a foreign key relationship between the Comment and Post models, enabling users to leave comments linked to specific blog posts and their authors.

These changes enhance post categorization and comment management, essential features for an effective blogging platform. The new PostManager allows for better filtering and retrieval of posts based on criteria like publication status or author.

This exercise wraps up the integration of related models, making the blog more dynamic and user-friendly. Next, I will focus on creating views and templates to display these relationships on the front end of the application.

Exercise 3: Custom Manager

Objective

The goal of this exercise is to create a custom manager for the *Post* model that allows searching for posts based on specific criteria. The manager will include:

1. A method to return only published posts (those with a non-null *published_date*).
2. A method to filter posts by a specific author, making it easy to find posts written by a particular user.

Description of Implementation Steps

1. **Creating the *PostManager* Class:**

I created the *PostManager* class, which extends Django's built-in *models.Manager*. A custom manager helps group-specific query logic for the model. This keeps queries reusable and easy to maintain.

2. **Method for Published Posts:**

I defined the *published* method in the *PostManager* to return only posts with a non-null *published_date*. This method uses Django's filtering syntax (*filter(published_date__isnull=False)*) to ensure that only published posts are included.

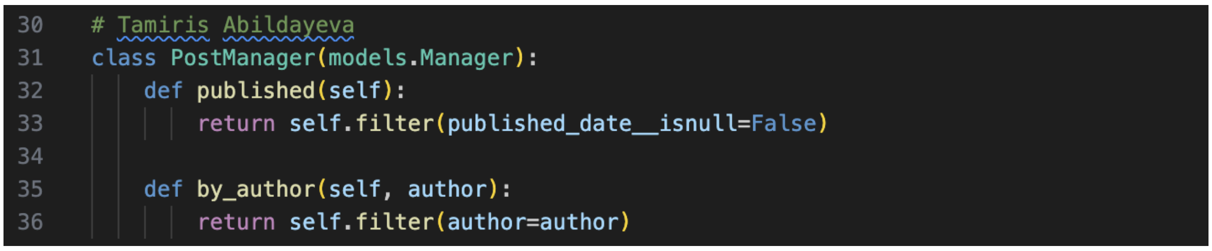
3. **Method to Get Posts by Author:**

I added the *by_author* method to the *PostManager* to filter posts by a specific author. This method takes an *author* parameter (a *User* object) and returns all posts written by that user. The query *filter(author=author)* filters posts based on the author field, which connects to the *User* model.

4. **Assigning the Custom Manager to the *Post* Model:**

I assigned the *PostManager* to the *Post* model by setting it to the *objects* attribute. This allows the custom methods (*published* and *by_author*) to be used like any other methods when searching through the *Post* model.

Here is the complete code for the custom manager:



```python
# Tamiris Abildayeva
class PostManager(models.Manager):
    def published(self):
        return self.filter(published_date__isnull=False)

    def by_author(self, author):
        return self.filter(author=author)
```

Fig. 3: Screenshot of the PostManager code

By integrating the PostManager into the Post model and assigning it to the objects attribute, I optimized the process of querying posts based on their publication status or author. This custom manager improves the flexibility and maintainability of the application.

5. Explanation of the Custom Manager Methods

1. **published()**: This method filters the *Post* model to return only posts with a non-null *published_date*. The query *filter(published_date__isnull=False)* ensures that only posts with a valid *published_date* are shown.
2. **by_author()**: This method filters the *Post* model by the *author* field. It takes an *author* parameter (a *User* instance) and returns posts written by that author. The query *filter(author=author)* makes sure that only posts by the specified *User* are included.

Django Views

In Django, views are essential to an application. They handle requests, get data, and create responses in formats like HTML or JSON. There are two main types of views: function-based views (FBVs) and class-based views (CBVs). Each type offers a different way to manage HTTP requests.

This chapter will focus on function-based views (FBVs). FBVs are simple Python functions that take a request and return a response. They work well for straightforward tasks that need clear logic. I will work on Exercise 4, where I will create two function-based views: one to list all blog posts and another to show a single post by its ID.

Exercise 4: Function-Based Views

Objective

This exercise focuses on implementing two powerful function-based views in Django:

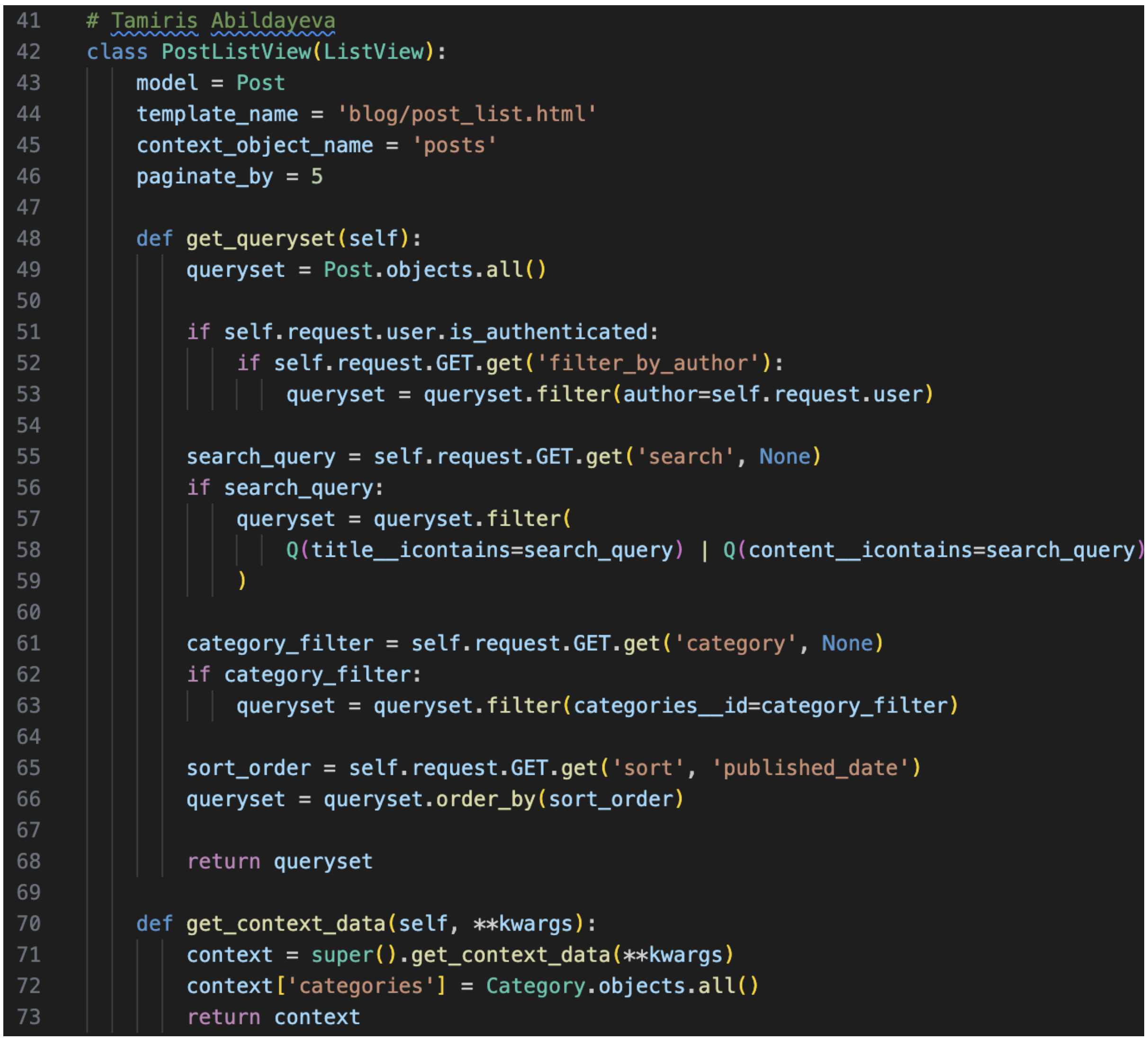
1. A view that effectively lists all blog posts—offering users a clear page displaying all available blog content.
2. A detailed view that showcases a single post by its ID, providing users with an in-depth look at a specific blog entry.

These views will utilize the Post model to retrieve essential data and present it seamlessly through templates. This exercise showcases the fundamental capabilities of Django views, allowing for efficient operations such as resource listing and detail display.

Description of Implementation Steps

1. Implementing the PostListView (List View):

To present all blog posts, I developed the PostListView function-based view. This view actively queries the Post model to retrieve all blog entries from the database and renders them in the *post_list.html* template.



```python
# Tamiris Abildayeva
class PostListView(ListView):
    model = Post
    template_name = 'blog/post_list.html'
    context_object_name = 'posts'
    paginate_by = 5

    def get_queryset(self):
        queryset = Post.objects.all()

        if self.request.user.is_authenticated:
            if self.request.GET.get('filter_by_author'):
                queryset = queryset.filter(author=self.request.user)

        search_query = self.request.GET.get('search', None)
        if search_query:
            queryset = queryset.filter(
                Q(title__icontains=search_query) | Q(content__icontains=search_query)
            )

        category_filter = self.request.GET.get('category', None)
        if category_filter:
            queryset = queryset.filter(categories__id=category_filter)

        sort_order = self.request.GET.get('sort', 'published_date')
        queryset = queryset.order_by(sort_order)

        return queryset

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['categories'] = Category.objects.all()
        return context
```

Fig. 4: Screenshot of the PostListView code

I enhanced user experience by implementing pagination, displaying five posts per page. The *paginate_by = 5* parameter ensures an organized layout, making navigation intuitive. Furthermore, I incorporated robust filtering functionality that enables users to search for posts by title or content (*search_query*), filter by category (*category_filter*), and sort posts by either published date or title (*sort_order*). The view context also includes all available categories, empowering users to filter posts easily.

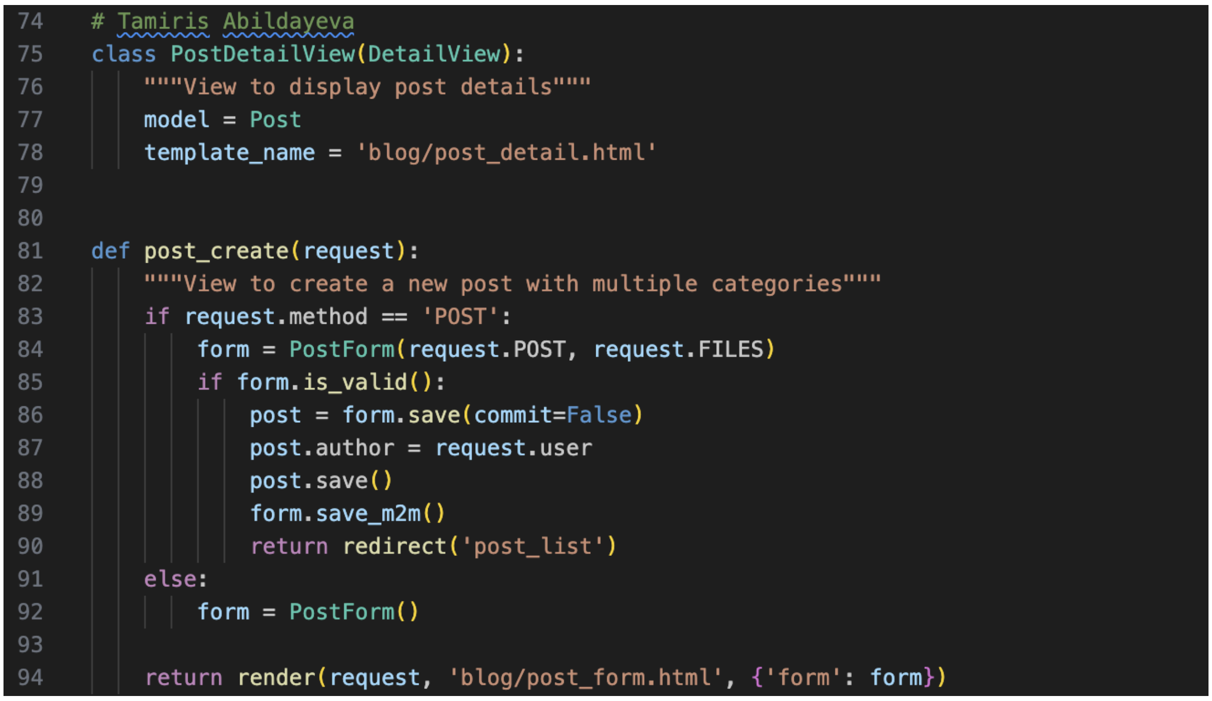
2. **Implementing the PostDetailView (Detail View):**

For displaying individual blog posts, I created the PostDetailView function-based view. This view accepts a *post_id* as a parameter, retrieves the relevant post from the database, and renders the *post_detail.html* template with comprehensive details about the post. Using Django's *get_object_or_404* method guarantees that if the post does not exist, a 404 error is automatically raised, providing a robust fallback.

3. **Rendering the Views:**

These views are linked to specific HTML templates:

1. **post_list.html** effectively showcases a list of posts and integrates pagination, search, and category filters.
2. **post_detail.html** presents the details of a single post, rendering crucial information such as the title, content, author, published date, and associated comments.



```python
# Tamiris Abildayeva
class PostDetailView(DetailView):
    """View to display post details"""
    model = Post
    template_name = 'blog/post_detail.html'


def post_create(request):
    """View to create a new post with multiple categories"""
    if request.method == 'POST':
        form = PostForm(request.POST, request.FILES)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.save()
            form.save_m2m()
            return redirect('post_list')
    else:
        form = PostForm()

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

Fig. 5: Screenshot of the PostDetailView code

In this exercise, I successfully created two function-based views:

1. **post_list()** for efficiently listing all blog posts with filtering, searching, and pagination features.

2. **post_detail()** for displaying detailed information about a single post based on its ID.

Both views interact seamlessly with the Post and Category models to fetch and present relevant data, showcasing the foundational elements necessary for building effective blog content display and filtering in Django.

Exercise 5: Class-Based Views

Objective

The objective of this exercise is to refactor the existing function-based views (FBVs) from Exercise 4 into class-based views (CBVs). I will leverage Django's *ListView* to efficiently list blog posts and *DetailView* to display post details. This transition is designed to streamline and simplify the views, enhancing their reusability and firmly aligning with Django's best practices. Additionally, I will ensure that templates and URL patterns are updated to function seamlessly with the new class-based structure.

Description of Implementation Steps

1. **Refactoring the Post List View (PostListView):**

    1. **Convert the Function-Based View**: I have transformed the function-based *post_list* view into a class-based view using *ListView*.
    2. **Implement Custom Filtering and Sorting**: Since *ListView* inherently provides a query set and pagination, I have implemented any custom filtering and sorting logic within the *get_queryset* method.
    3. **Override Context Data**: The *get_context_data* method is overridden to include the *categories* context, enabling effective filtering of posts by category in the template.
    4. **Enable Pagination**: I have utilized the *paginate_by* attribute to add pagination functionality to the list view.

2. **Refactoring the Post Detail View (PostDetailView):**

    1. **Convert the Function-Based View**: I have converted the function-based *post_detail* view into a class-based view using *DetailView*.
    2. **Simplify Object Retrieval**: The *DetailView* automatically fetches the object to display (a single *Post* object) and passes it to the template as an *object*, eliminating the need for manual object fetching via *get_object_or_404()*.

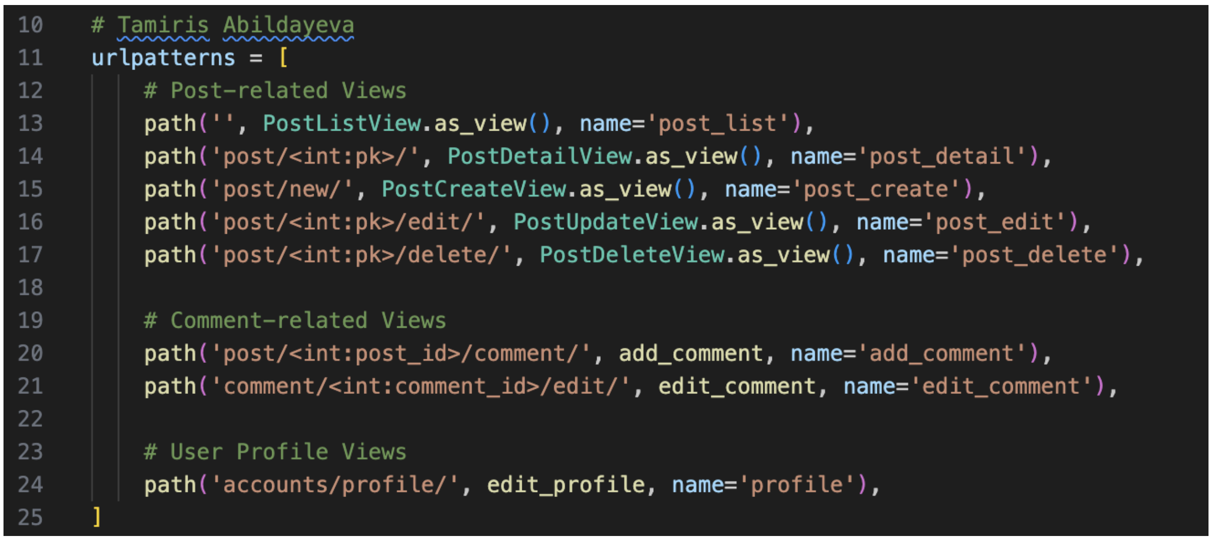
3. **Template Updating:**

    1. **post_list.html**: The template update to fully utilize the context provided by the *PostListView* class. I have accessed the *posts* context variable, which contains the list of posts, along with *page_obj* for pagination.

2. **post_detail.html**: For the *PostDetailView*, the *post* object is automatically passed to the template, and the context variable is accessed using *post* (as defined by *context_object_name*).

4. **URL Patterns Updating:**

I have updated the URL patterns to utilize class-based views by calling *.as_view()* on the relevant CBV classes:



```python
# Tamiris Abildayeva
urlpatterns = [
    # Post-related Views
    path('', PostListView.as_view(), name='post_list'),
    path('post/<int:pk>/', PostDetailView.as_view(), name='post_detail'),
    path('post/new/', PostCreateView.as_view(), name='post_create'),
    path('post/<int:pk>/edit/', PostUpdateView.as_view(), name='post_edit'),
    path('post/<int:pk>/delete/', PostDeleteView.as_view(), name='post_delete'),

    # Comment-related Views
    path('post/<int:post_id>/comment/', add_comment, name='add_comment'),
    path('comment/<int:comment_id>/edit/', edit_comment, name='edit_comment'),

    # User Profile Views
    path('accounts/profile/', edit_profile, name='profile'),
]
```

Fig. 6: Screenshot of the URL patterns code

By refactoring the views to use Django's class-based views (*ListView*, *DetailView*, *CreateView*, *UpdateView*, *DeleteView*), I am significantly enhancing the code's maintainability and reusability. The templates and URL patterns be expertly adjusted to work with these views, ensuring that the functionalities for filtering, pagination, and sorting remain robust. This strategy is aligned with Django's best practices and elevates both the readability and scalability of the application.

Exercise 6: Handling Forms

Objective

The objective of this exercise is to confidently create a form for adding new blog posts using Django's *forms.ModelForm*. I will implement a robust view that handles form submissions, validates the data, and saves the new post in the database. Moreover, I will ensure that the form is rendered correctly within the template and provide users with appropriate feedback upon successful submission.

Description of Implementation Steps

1. **Defining the PostForm using ModelForm:**

I have created an effective form class for the *Post* model utilizing Django's *forms.ModelForm*. This process automatically generates fields corresponding to the *Post* model while handling validation, saving, and all necessary form-related logic seamlessly.

2. **Creating the View for Handling Form Submission:**

Next, I have developed a view function or a class-based view that expertly manages the rendering of the form and handles its submission. This critical step involves validating the form data, saving it to the database, and providing success messages or error handling for invalid forms.

3. **Creating the Template for Displaying the Form:**

I have crafted a user-friendly template that renders the form and enables users to input the title, content, and other essential fields for the new post. This template also proficiently displays error messages when the form is not valid, ensuring a smooth user experience.

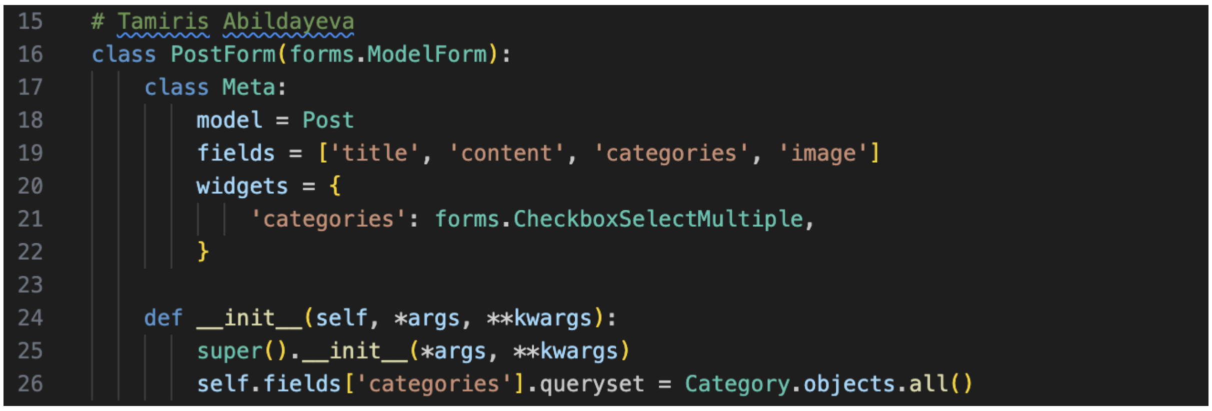
4. **Handling Form Submission and Providing Feedback:**

Upon successful form submission, the new post is seamlessly saved to the database, and users have been redirected to a success page (such as the list of posts). In the case of invalid submissions, the form is re-rendered with clear error messages indicating the necessary corrections.

5. **Implementation Details**

    1. **Creating the *PostForm*:**

In *forms.py*, I have defined the *PostForm* as a *ModelForm*:



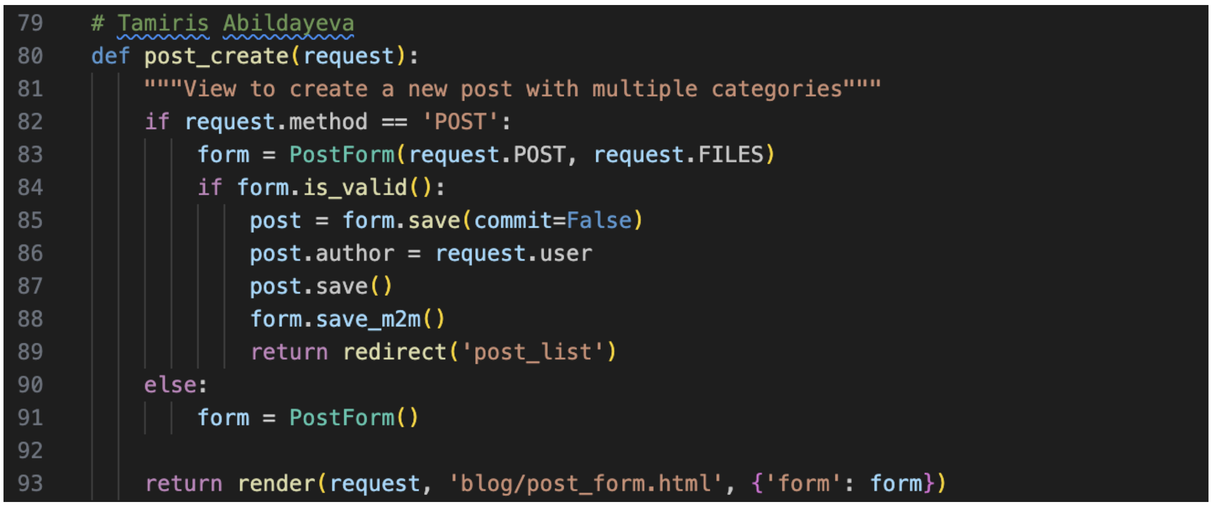
```
# Tamiris Abildayeva
class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['title', 'content', 'categories', 'image']
        widgets = {
            'categories': forms.CheckboxSelectMultiple,
        }

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fields['categories'].queryset = Category.objects.all()
```

Fig. 7: Screenshot of the PostForm code

The *PostForm* class leverages *ModelForm* to automatically generate fields from the *Post* model. I have specified the fields to include in the form (e.g., title, content, image, and categories). Additionally, I have implemented a custom validation method, *clean_title*, to ensure the title is not empty, which enhances the form's reliability.

2. Creating the View to Handle Form Submission:

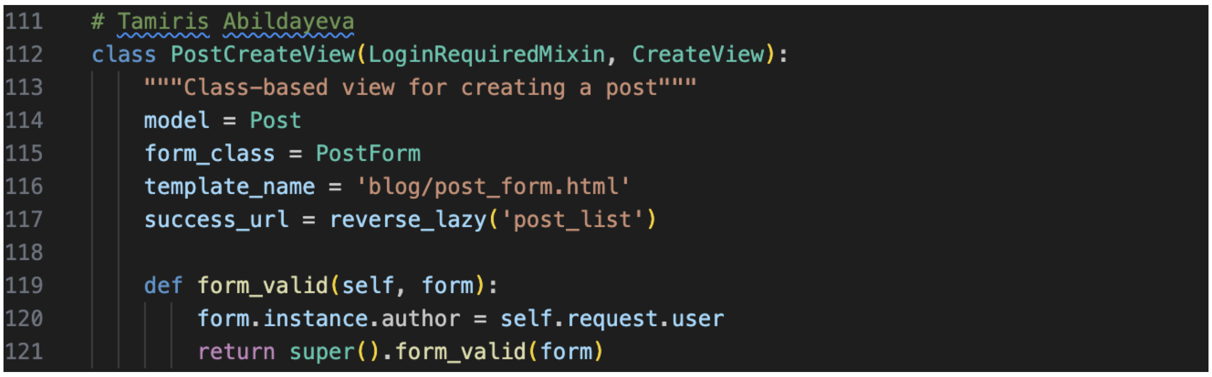
Then, I created a view that renders the form and effectively manages its submission. This view either saves the new post or re-render the form with helpful error messages if validation fails. Users can opt for either a Function-Based View (FBV) or a Class-Based View (CBV), and I have demonstrated both approaches.



```python
# Tamiris Abildayeva
def post_create(request):
    """View to create a new post with multiple categories"""
    if request.method == 'POST':
        form = PostForm(request.POST, request.FILES)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.save()
            form.save_m2m()
            return redirect('post_list')
    else:
        form = PostForm()

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

Fig. 8: Screenshot of the post_create code

For the class-based approach, I have utilized *CreateView*, which expertly handles the form logic.



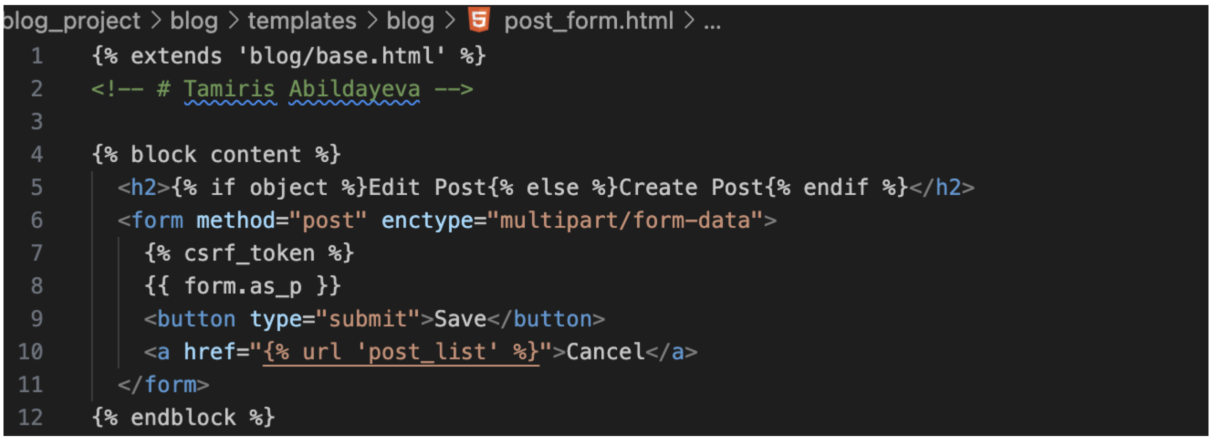
```python
# Tamiris Abildayeva
class PostCreateView(LoginRequiredMixin, CreateView):
    """Class-based view for creating a post"""
    model = Post
    form_class = PostForm
    template_name = 'blog/post_form.html'
    success_url = reverse_lazy('post_list')

    def form_valid(self, form):
        form.instance.author = self.request.user
        return super().form_valid(form)
```

Fig. 9: Screenshot of the PostCreateView code

The *PostCreateView* extends *CreateView*, leveraging *PostForm* for efficient form management. The *success_url* specifies the redirect destination after successful form submission (such as the post list). Moreover, I have overridden the *form_valid* method to assign the current user as the author of the new post before saving the form.

3. Creating the Template to Display the Form:

In *post_form.html*, I have rendered the form while efficiently managing the display of errors and success messages.



blog_project > blog > templates > blog > post_form.html > ...

```
{% extends 'blog/base.html' %}
<!-- # Tamiris Abildayeva -->

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

Fig. 10: Screenshot of the post_form.html code

The form rendered using *{{ form.as_p }}*, which presents the fields as *<p>* elements. The *enctype="multipart/form-data"* attribute is essential for handling file uploads (like post images). I have also ensured that success or error messages are displayed using Django's messaging framework.

4. Updating the URL Patterns:

I have seamlessly added the view to *urls.py*. Depending on the user's choice of a function-based or class-based view, the URL patterns vary slightly.

6. Testing and Validation

Once everything was in place, I thoroughly tested the following aspects:

1. **Form Validation:** By attempting to submit the form with missing or invalid data, it displays errors and prevents submission effectively.
2. **Success Handling:** After successfully submitting the form, the new post be saved, and users be redirected to the post list with a well-deserved success message.
3. **Image Upload:** If the *Post* model includes an *ImageField*, verify that the images upload correctly, confirming the form's functionality.

In this exercise, I have successfully:

1. Created a *PostForm* using Django's *forms.ModelForm* to expertly handle blog post submissions.
2. Implemented a view to efficiently process the form, validate the data, and save the new post to the database.
3. Developed a user-friendly template to render the form and display relevant error or success messages.
4. Updated the URL patterns to ensure they route correctly to the designated view.

This comprehensive approach empowers to handle form submissions, validation, and object creation cleanly and maintainably while providing an exceptional user experience complete with validation feedback.

Django Templates

In Django, templates play a crucial role in effectively separating the presentation layer from the business logic. They empower you to dynamically generate HTML content that is seamlessly delivered to users. With templates, programmers can easily create reusable structures for web pages, insert dynamic content, and apply logic such as loops and conditions to manipulate the incoming data.

In this chapter, I will implement the essential concepts of Django templates, including rendering dynamic data, utilizing template tags and filters, and mastering template inheritance. I will also use features of templates, including conditional statements, loops, and the inclusion of other templates, all of which significantly enhance code reuse.

By the end of this chapter, I will create clean, maintainable, and dynamic views using Django's templating system.

Exercise 7: Basic Template Rendering

Objective

In this exercise, I will confidently create a robust template to display a list of blog posts. Each post will prominently feature the title, author, and publication date. I will utilize Django’s built-in template tags to format the publication date effectively. Moreover, I will ensure the list is dynamic, incorporating powerful search, sorting, and pagination functionalities.

Description of Implementation Steps

1. **Creating the Template for Post Listing:**

I have extended the *base.html* template to craft a new *post_list.html* template that efficiently renders the list of blog posts. Each post, displays its title, author, and publication date, utilizing the *date* filter to present the publication date in a clear, human-readable format (e.g., "January 1, 2024").

2. **Implementing the Search feature:**

I have implemented a search for posts by their title through an intuitive GET form, filtering the list based on the entered search term.

```
     <form method="GET" action="{% url 'post_list' %}" class="search-form">
       <div class="search-bar">
           <input type="text" name="search" placeholder="Search posts..." value="{{
request.GET.search }}" class="search-input">
         <button type="submit" class="btn btn-search">Search</button>
       </div>
     </form>
```

Fig. 11: Code of the Search feature

3. Adding Category Filtering:

Users could filter posts by category using a straightforward dropdown menu, resulting in a seamless re-rendering of the page with the appropriate posts.

```
     <form method="GET" action="{% url 'post_list' %}" class="category-form">
       <div class="category-filter">
               <select id="category" name="category" onchange="this.form.submit()"
class="category-select">
           <option value="">Select Category</option>
           {% for category in categories %}
                        <option value="{{ category.id }}" {% if category.id ==
request.GET.category %} selected {% endif %}>
               {{ category.name }}
             </option>
           {% endfor %}
         </select>
       </div>
     </form>
```

Fig. 12: Code of the Category filtering feature

4. Sorting the Posts:

I have offered users the flexibility to sort posts by either title or publication date, achieved by modifying the query that retrieves the posts.

```
     <div class="sort-options">
       <a href="?sort=published_date" class="btn btn-sort">Sort by Date</a>
       <a href="?sort=title" class="btn btn-sort">Sort by Title</a>
     </div>
```

Fig. 13: Code of the Sorting feature

5. Handling Pagination:

To enhance usability and avoid overwhelming users with too much information, I have implemented pagination. This feature displays a manageable number of posts per page, complete with navigation links for a smooth user experience.

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

```
    {% endif %}

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

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

Fig. 14: Code of the Pagination feature

6. Conditional Rendering Based on Authentication:

I have intelligently conditioned the rendering of specific elements, such as the "Create New Post" button and post management actions, based on whether the user is logged in and their permissions as an author or admin.

```
{% if user.is_authenticated %}
  <h2>Blog Posts by {{ user.username }}</h2>
{% else %}
  <h2>Blog Posts</h2>
{% endif %}
```

Fig. 15: Code of the “Create New Post” feature

7. Key Template Features:

1. **Inheritance from Base Template:** By using *{% extends 'blog/base.html' %}*, *post_list.html* inherits the established structure defined in the *base.html* template, ensuring consistency.
2. **Search and Category Filtering:** The search bar effectively allows users to search for posts by title, while the category dropdown facilitates straightforward filtering by category.
3. **Pagination:** The pagination section expertly manages navigation between pages of posts, enhancing user navigation.
4. **Conditional Rendering:** Leveraging *{% if user.is_authenticated %}*, dynamically displays the "Create New Post" button and post management options (edit and delete) based on the user’s authentication status and permissions.
5. **Template Filter (date):** Utilizing Django’s *date* filter allows to format the publication date of posts clearly, such as "January 1, 2024".

In this exercise, I have successfully created a dynamic template to showcase blog posts, integrating essential features such as search, category filtering, sorting, and pagination.

By employing Django's template language, I have developed reusable components and logic that significantly enhance the user experience. This strategic approach ensures a clear separation of concerns, maintaining a distinct distinction between presentation logic (HTML) and business logic (views and models).

Exercise 8: Template Inheritance

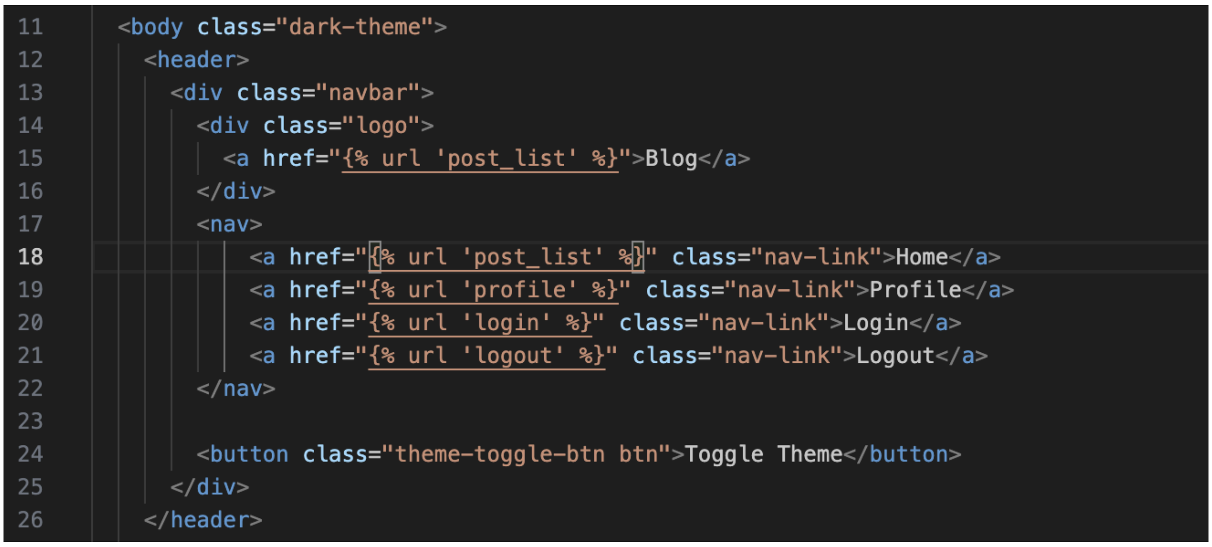
Objective

In this exercise, I will confidently explore template inheritance in Django by creating a robust base template that incorporates a header and footer. I will effectively extend this base template to render the content for both the list and detailed views of blog posts. Template inheritance is a powerful technique that enables to elimination of duplicative structures across web pages (such as headers, footers, and navigation). Defining these common elements in a base template can seamlessly override or extend the content in individual templates.

Description of Implementation Steps

1. **Creating a *base.html* Template:**

I have created a *base.html* template that establishes the fundamental structure of pages, including the header, navigation, footer, and a dedicated placeholder for dynamic content. This template utilizes blocks such as *{% block title %}* for the page title and *{% block content %}* for page-specific content.



```html
<body class="dark-theme">
  <header>
    <div class="navbar">
      <div class="logo">
        <a href="{% url 'post_list' %}">Blog</a>
      </div>
      <nav>
        <a href="{% url 'post_list' %}" class="nav-link">Home</a>
        <a href="{% url 'profile' %}" class="nav-link">Profile</a>
        <a href="{% url 'login' %}" class="nav-link">Login</a>
        <a href="{% url 'logout' %}" class="nav-link">Logout</a>
      </nav>

      <button class="theme-toggle-btn btn">Toggle Theme</button>
    </div>
  </header>
```

Fig. 16: Screenshot part of the base.html code

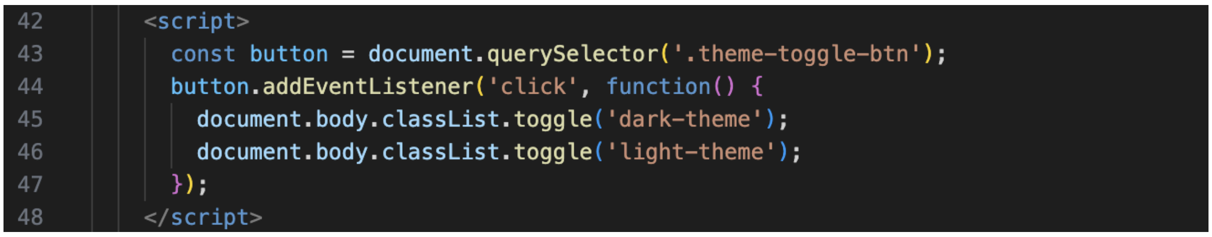
2. **Extending *base.html* in the Post List and Detail Templates:**

The *post_list.html* and *post_detail.html* templates extended *base.html*, allowing to reuse of its header and footer efficiently. These templates simplify the need to populate the

{% block content %} section with their specific content, showcasing the beauty of inheritance.

3. Adding Styling and JavaScript:

The *base.html* also incorporates static file loading for styles and scripts. I implemented a theme toggle feature, enabling users to switch effortlessly between light and dark modes, which is accessible on all pages that extend *base.html*.



```
<script>
  const button = document.querySelector('.theme-toggle-btn');
  button.addEventListener('click', function() {
    document.body.classList.toggle('dark-theme');
    document.body.classList.toggle('light-theme');
  });
</script>
```

Fig. 17: Screenshot of the JavaScript part in the base.html code

4. Implementation *base.html* Template:

This template defines the shared structure of web pages.

1. **Title Block:** The *{% block title %}* tag empowers each page to customize its title, defaulting to "Blog by Tamiris" when not overridden.
2. **Main Content Block:** The *{% block content %}* tag serves as the area where each child template defines its specific content.
3. **Navigation:** The header features a navigation bar with links to core pages like Home, Profile, Login, and Logout.
4. **Theme Toggle:** A straightforward JavaScript function is implemented for toggling between dark and light themes.

5. Implementation *post_list.html* Template:

This template inherits from *base.html* to present the blog post list.

1. **Title Block:** It overrides the base template's title, displaying "Blog Posts."
2. **Content Block:** This section implements the listing of posts with details such as titles, summaries, authors, and publication dates.
3. **Pagination:** I add pagination to facilitate navigation through the post list.

6. Implementation *post_detail.html* Template:

This template also inherits from *base.html*, providing content for individual post details.

1. **Title Block**: It overrides the base template's title with the specific post's title.

2. **Content Block**: This section renders detailed information about the post, including the title, content, categories, and comments.

Template inheritance in Django is a formidable feature that enhances code reuse and maintainability. By defining a *base.html* template, I centralize common page elements (like headers, footers, navigation, etc.) and utilize *{% block %}* tags to inject unique content for each page. Through this exercise, I have effectively created and extended templates to structure our blog views with confidence, significantly simplifying the management of shared components across multiple pages.

Exercise 9: Static Files and Media

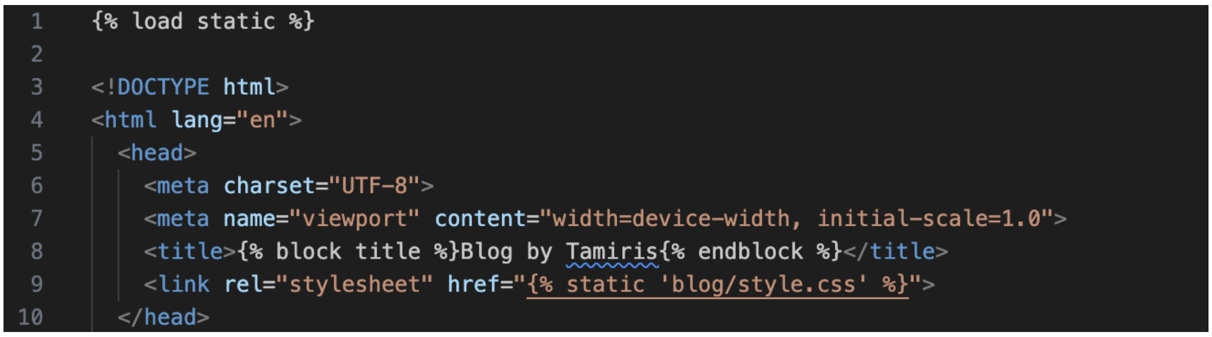
Objective

In this exercise, I will implement static files for CSS styles and configure media file handling, enabling users to effortlessly upload images for their posts. Below, I outline the key changes and configurations I made, complete with relevant code snippets.

Description of Implementation Steps

1. **Adding CSS Styles Using Static Files**

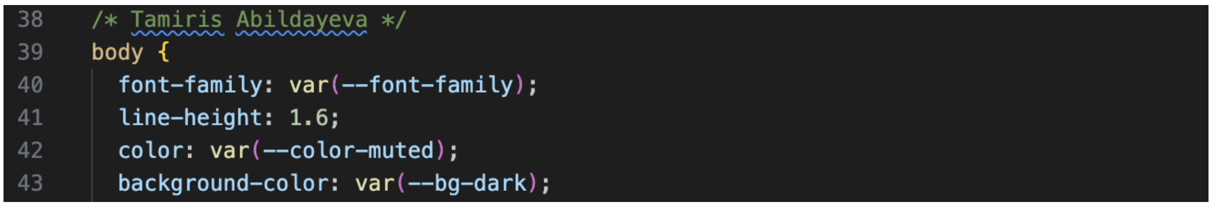
    1. **Directory Structure:** I established a *static/css* folder to house the *styles.css* file.
    2. **Linked CSS in Templates:** In the base template (*base.html*), I linked the static CSS file using *{% load static %}*:



```
{% load static %}

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Blog by Tamiris{% endblock %}</title>
    <link rel="stylesheet" href="{% static 'blog/style.css' %}">
  </head>
```

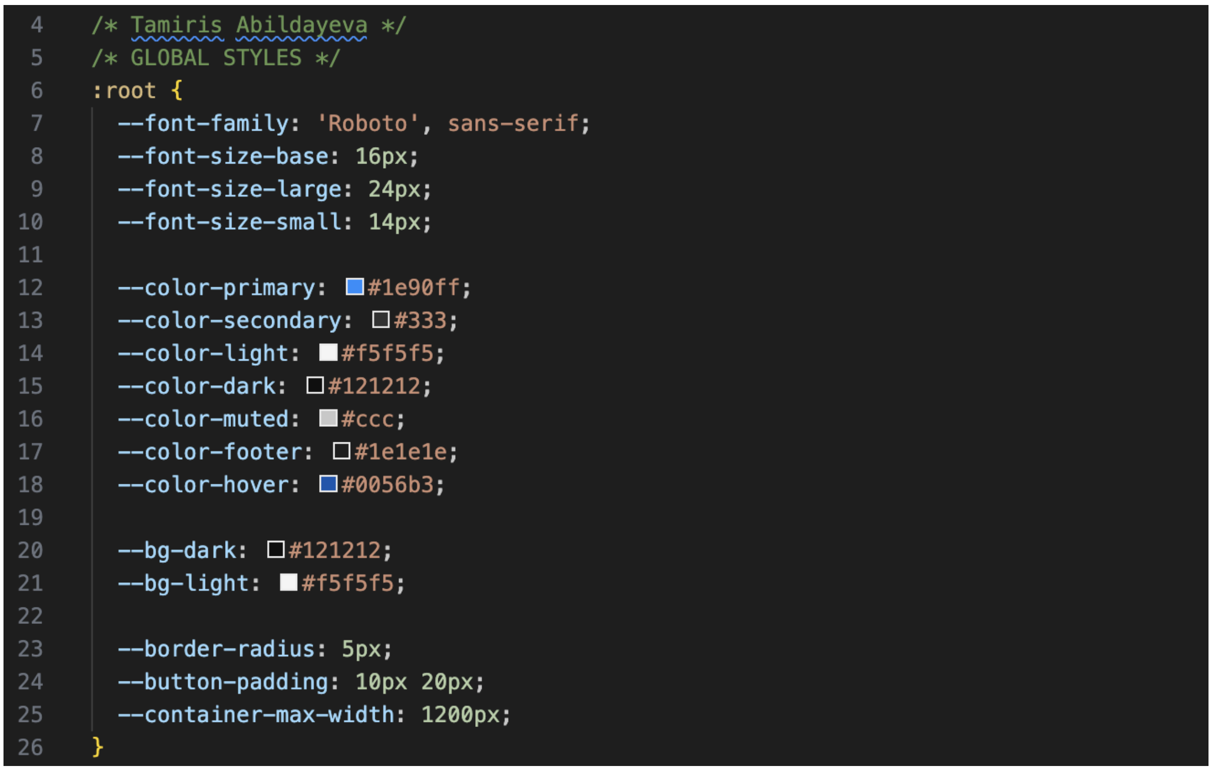
Fig. 18: Screenshot of the Style linked part in the base.html code

    3. **Global Styles:** I defined custom root variables and additional styles that significantly enhance the design and user experience:



```css
/* Tamiris Abildayeva */
body {
  font-family: var(--font-family);
  line-height: 1.6;
  color: var(--color-muted);
  background-color: var(--bg-dark);
```

Fig. 19: Screenshot of the body styling part in the style.css code

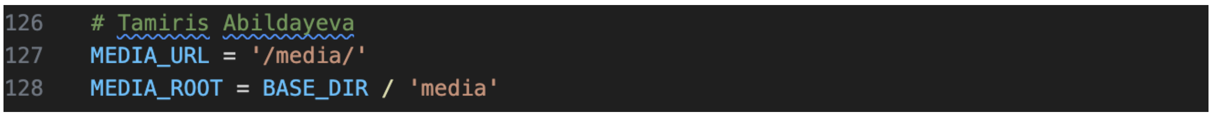


```css
/* Tamiris Abildayeva */
/* GLOBAL STYLES */
:root {
  --font-family: 'Roboto', sans-serif;
  --font-size-base: 16px;
  --font-size-large: 24px;
  --font-size-small: 14px;

  --color-primary: #1e90ff;
  --color-secondary: #333;
  --color-light: #f5f5f5;
  --color-dark: #121212;
  --color-muted: #ccc;
  --color-footer: #1e1e1e;
  --color-hover: #0056b3;

  --bg-dark: #121212;
  --bg-light: #f5f5f5;

  --border-radius: 5px;
  --button-padding: 10px 20px;
  --container-max-width: 1200px;
}
```

Fig. 20: Screenshot of the root styling part in the style.css code

2. Setting Up Media Folder for User Uploads

In the *settings.py* file, I confidently configured Django to handle media files, allowing users to upload images seamlessly:



```python
# Tamiris Abildayeva
MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

Fig. 21: Screenshot of the settings.py code

This setup guarantees that all uploaded files are stored securely in the *media/* directory.

3. Updating Post Model for Image Uploads

In the *Post* model, I added an *ImageField* to allow users to easily upload images for their posts:

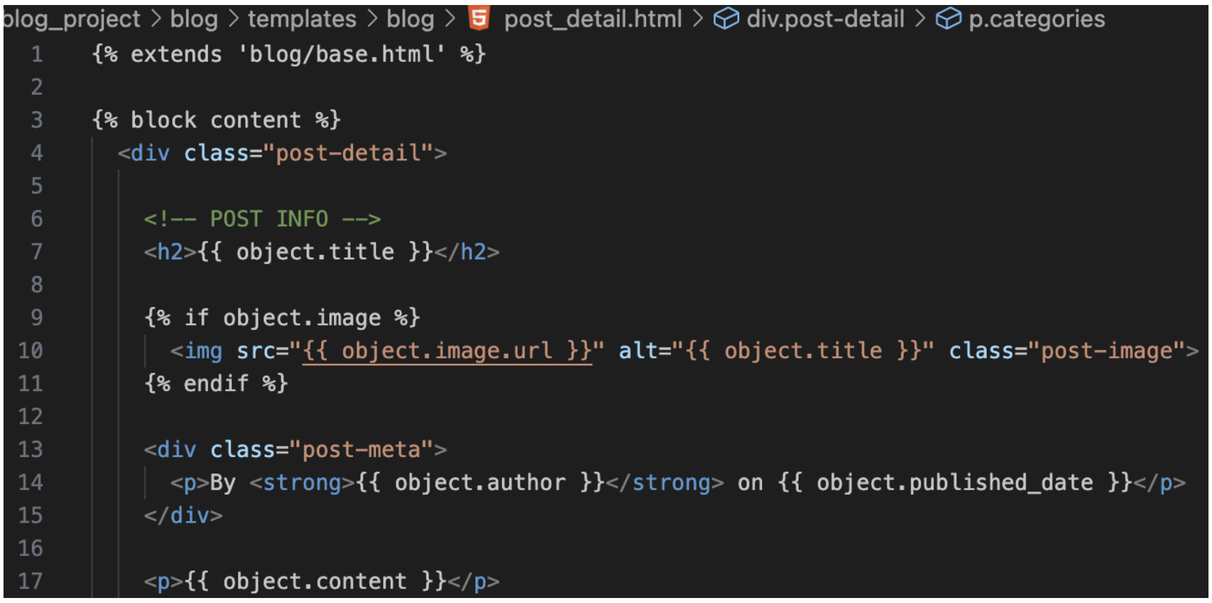
4. **Updating Form and Views for Image Uploads**

In the post creation form, I integrated the *image* field:

In the view that handles post creation, the form can handle image uploads by utilizing *request.FILES*.

5. **Displaying Images in Templates**

In the templates, I implemented logic to display the uploaded images, ensuring the *image* field renders correctly in the post detail view:



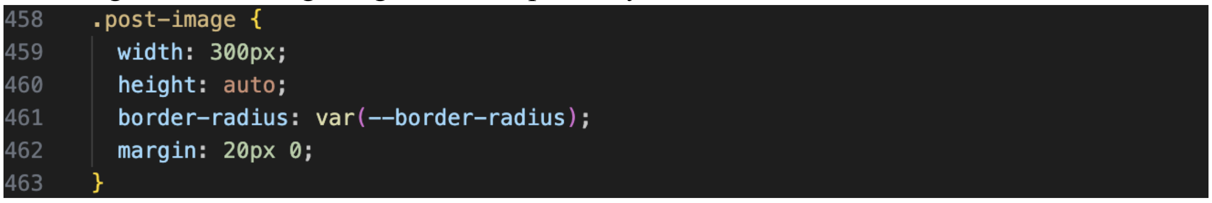
```
blog_project > blog > templates > blog > post_detail.html > div.post-detail > p.categories
{% extends 'blog/base.html' %}

{% block content %}
  <div class="post-detail">

    <!-- POST INFO -->
    <h2>{{ object.title }}</h2>

    {% if object.image %}
      <img src="{{ object.image.url }}" alt="{{ object.title }}" class="post-image">
    {% endif %}

    <div class="post-meta">
      <p>By <strong>{{ object.author }}</strong> on {{ object.published_date }}</p>
    </div>

    <p>{{ object.content }}</p>
```

Fig. 21: Screenshot of the post_detail.html code

This ensures that each post's image is prominently displayed on its detail page when available.

6. **Responsive Design for Media Content**

To guarantee that images are fully responsive across all screen sizes, I added the following CSS, allowing images to scale perfectly:



```
.post-image {
  width: 300px;
  height: auto;
  border-radius: var(--border-radius);
  margin: 20px 0;
}
```

Fig. 22: Screenshot of the post-image styling part in the style.css code

This ensures that the images maintain their aspect ratio while adjusting to the width of their container.

Through these strategic implementations, I successfully enabled user-uploaded images to be showcased in blog posts, seamlessly integrated CSS for enhanced styling and ensured a responsive design for an outstanding user experience. This robust approach makes the blog not only more interactive and visually appealing but also efficiently handles media uploads in Django.

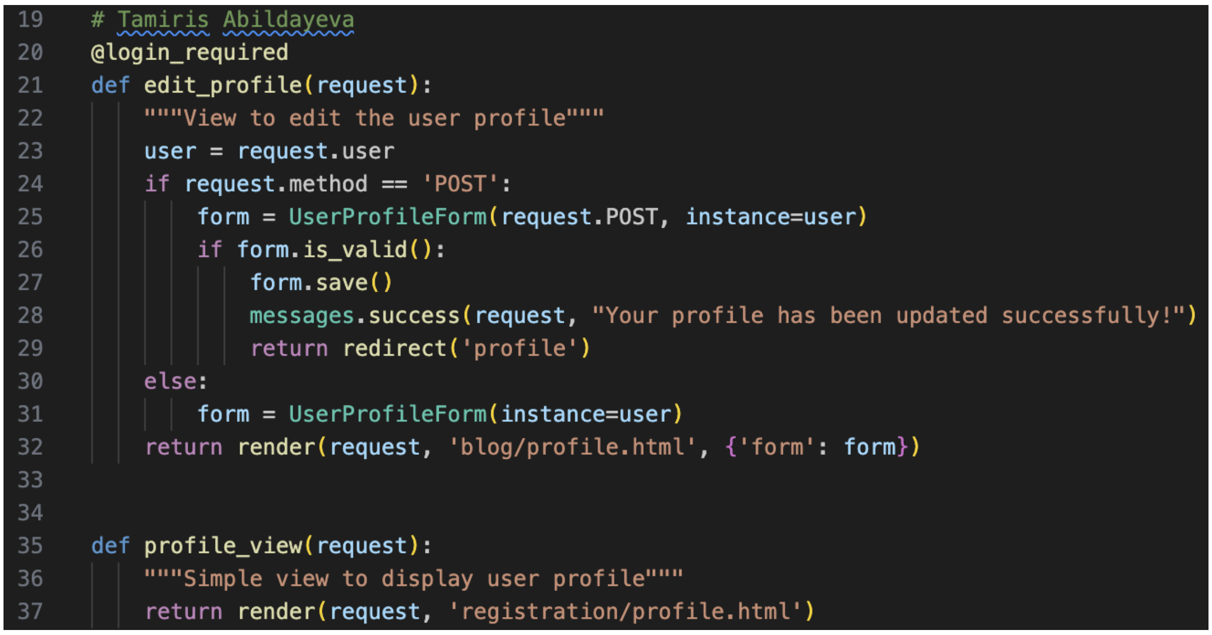
Code Snippets

Throughout this project, I have provided detailed code snippets for each exercise in the report, which reflect the steps and implementations I followed to complete the Django blog application. However, I previously mentioned that I would avoid repeating the code for every single exercise, as I have already included the relevant snippets when describing each task in detail. This serves as a reminder that I am not unnecessarily duplicating code in this section.

I have implemented several impactful views to improve user profile management and comment functionality. These views empower users to manage their profiles and engage with posts effectively.

1. **User Profile Management**

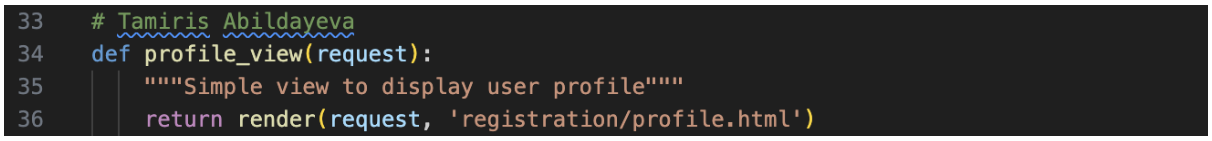
    1. **Edit Profile:** The *edit_profile* view allows logged-in users to update their profile information easily using the *@login_required* decorator. Upon submitting a valid form, users receive a success message, enhancing their experience.



```python
# Tamiris Abildayeva
@login_required
def edit_profile(request):
    """View to edit the user profile"""
    user = request.user
    if request.method == 'POST':
        form = UserProfileForm(request.POST, instance=user)
        if form.is_valid():
            form.save()
            messages.success(request, "Your profile has been updated successfully!")
            return redirect('profile')
    else:
        form = UserProfileForm(instance=user)
    return render(request, 'blog/profile.html', {'form': form})


def profile_view(request):
    """Simple view to display user profile"""
    return render(request, 'registration/profile.html')
```

Fig. 23: Screenshot of the edit_profile part in the views.py code

    2. **Profile View:** The *profile_view* function displays the user’s profile, providing a dedicated space for managing account details.

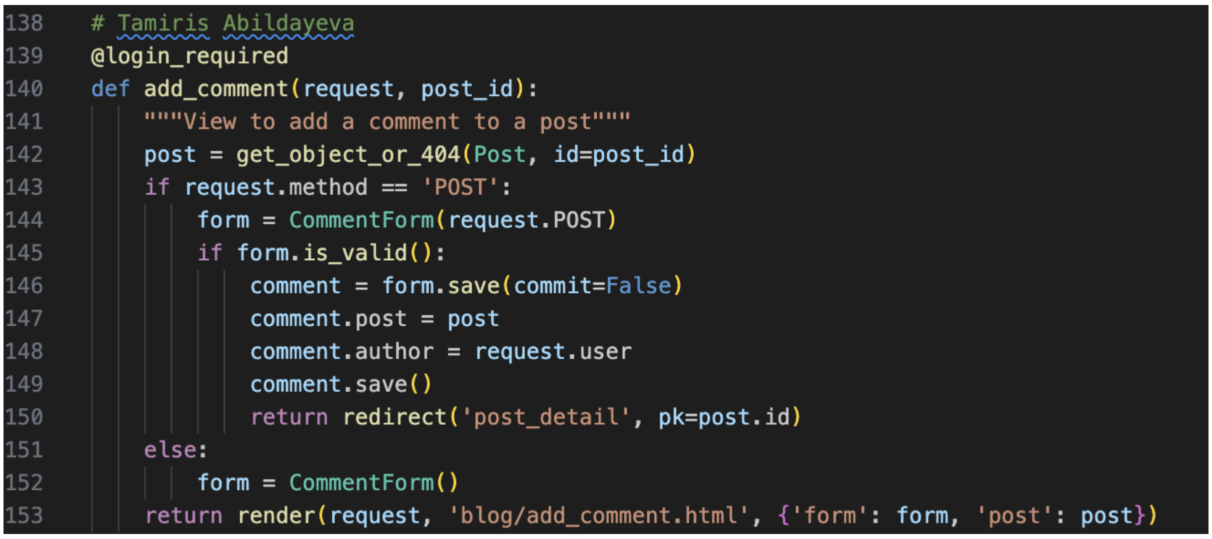


```python
# Tamiris Abildayeva
def profile_view(request):
    """Simple view to display user profile"""
    return render(request, 'registration/profile.html')
```

Fig. 24: Screenshot of the profile_view part in the views.py code

2. **Comment Management**

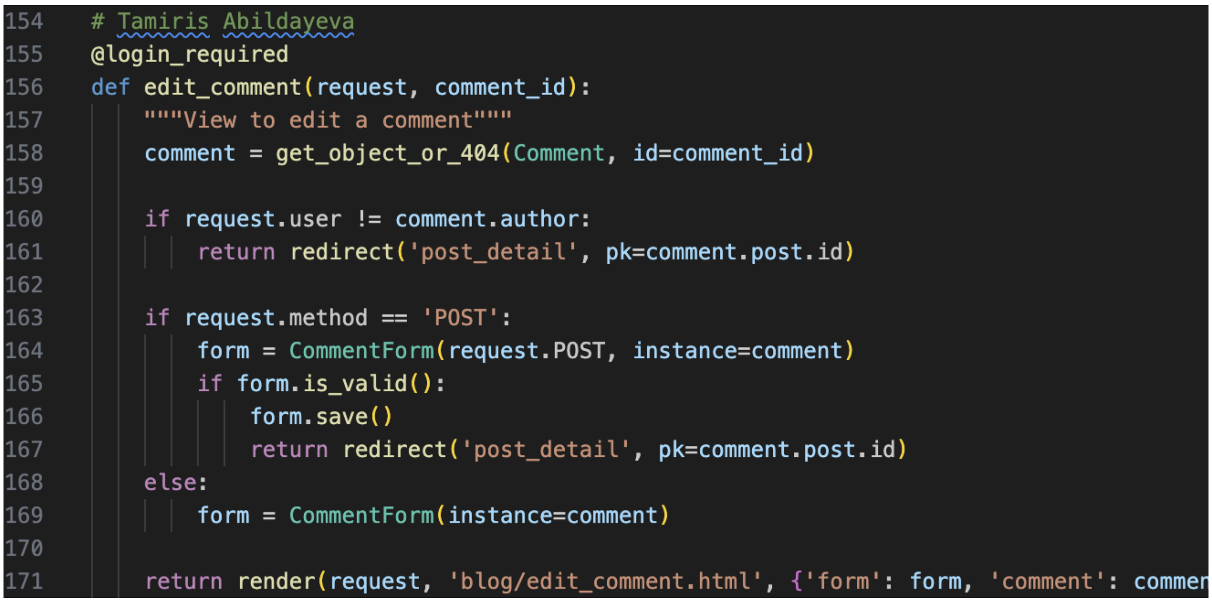
a. **Add Comment:** The *add_comment* view enables authenticated users to contribute comments to blog posts. Comments are validated and saved seamlessly, and users are redirected to the post detail page.



```python
# Tamiris Abildayeva
@login_required
def add_comment(request, post_id):
    """View to add a comment to a post"""
    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.post = post
            comment.author = request.user
            comment.save()
            return redirect('post_detail', pk=post.id)
    else:
        form = CommentForm()
    return render(request, 'blog/add_comment.html', {'form': form, 'post': post})
```

Fig. 25: Screenshot of the add_comment part in the views.py code

b. **Edit Comment:** The *edit_comment* view allows users to edit their comments with proper validation to maintain integrity.



```python
# Tamiris Abildayeva
@login_required
def edit_comment(request, comment_id):
    """View to edit a comment"""
    comment = get_object_or_404(Comment, id=comment_id)

    if request.user != comment.author:
        return redirect('post_detail', pk=comment.post.id)

    if request.method == 'POST':
        form = CommentForm(request.POST, instance=comment)
        if form.is_valid():
            form.save()
            return redirect('post_detail', pk=comment.post.id)
    else:
        form = CommentForm(instance=comment)

    return render(request, 'blog/edit_comment.html', {'form': form, 'comment': commen
```

Fig. 26: Screenshot of the edit_comment part in the views.py code

These views enhance user engagement by allowing them to manage their profiles and interact through comments. By utilizing Django's authentication and form-handling features, I have created a seamless experience that promotes user interaction and data integrity within the blog.

Results

In this section, I will summarize the results of each exercise, including achievements, challenges, and solutions encountered during the development process.

1. **Exercise 1: Creating a Basic Model**

Result: I created the *Post* model with fields for *title*, *content*, *author*, and *published_date*, implementing the *__str__()* method to return the post title. The model was registered with the admin interface for easy management.
Challenges & Solution:: The main challenge was understanding Django models and migrations, which I resolved by reviewing my migration files and consulting the documentation.

2. **Exercise 2: Model Relationships**

Result: I implemented a many-to-many relationship between *Post* and *Category*, as well as a foreign key relationship between *Post* and *Comment*.
Challenges: Initially, I struggled with cascading deletes for comments.
Solution: After further research, I added the *on_delete=models.CASCADE* option to ensure comments were deleted when the related post was removed.

3. **Exercise 3: Custom Manager**

Result: I created a custom manager for the *Post* model to filter for published posts and by author, which enhanced querying.
Challenges & Solution: The challenge was linking the manager to the model, which I resolved by assigning it to the *objects* attribute, as outlined in the Django documentation.

4. **Exercise 4: Function-Based Views**

Result: I created function-based views to list all blog posts and display individual posts by ID, successfully rendering the data in the templates.
Challenges: I struggled to pass the correct context to the templates.
Solution: I reviewed the context dictionary and made sure to include the necessary data (*post* and *posts*) in the correct format.

5. **Exercise 5: Class-Based Views**

Result: I refactored the views to use class-based views (CBVs) with *ListView* for the post list and *DetailView* for individual post details, resulting in cleaner code.
Challenges: Transitioning from function-based to class-based views was difficult, especially with context handling.
Solution: I studied the Django documentation and experimented with CBVs, which helped me understand *get_context_data* and its integration with CBVs.

6. **Exercise 6: Handling Forms**

Result: I implemented a form using Django's *ModelForm* to create new blog posts, ensuring it validated user input and saved data to the database.

Challenges: I had issues with correct form submission and redirection after submission.

Solution: I added validation for the form and ensured it re-rendered with error messages if invalid, while also fixing the redirect to the post list page after successful submission.

7. **Exercise 7: Basic Template Rendering**

Result: I created a template that displays a list of blog posts, including titles, authors, and publication dates. I also used a template tag to format the publication date.

Challenges: The main challenge was achieving the correct date formatting, as Django's template system employs a specific syntax for this purpose.

Solution: I referred to the Django documentation on template filters and utilized the *date* filter to format the publication date correctly. Once I applied this filter, the dates were displayed as expected.

8. **Exercise 8: Template Inheritance**

Result: I successfully implemented template inheritance by creating a base template with a consistent header and footer. I then extended this base template in both my list and detail views, which helped maintain a consistent look and feel throughout the site.

Challenges: One challenge was understanding the syntax for extending templates and how to pass content to different blocks.

Solution: I reviewed Django's template inheritance system and experimented with the *{% block %}* and *{% extends %}* tags until I became comfortable with their usage.

9. **Exercise 9: Static Files and Media**

Result: I added static files (CSS) to the project to style the templates and configured the media folder to allow image uploads for posts. This enhanced the blog's visual appeal and interactivity.

Challenges: The primary challenge was setting up media handling for images. Initially, I faced issues with serving media files during development due to incorrect settings.

Solution: After consulting the documentation on serving media files in development, I added the appropriate *MEDIA_URL* and *MEDIA_ROOT* settings to *settings.py* and updated *urlpatterns* in *urls.py* to serve media files. This resolved the issue, and the images were displayed properly.

10. **General Challenges and Solutions**

Throughout the development of this project, I encountered several common challenges:

1. **Understanding Django's ORM and Migrations:** Initially, I found Django's ORM system and migration commands somewhat confusing. The issue with migrations not being applied correctly was a recurring problem. However, by carefully reviewing the migration process and executing the necessary commands, I was able to resolve it.
2. **Form Handling and Validation:** Handling forms and validating user input posed challenges, particularly ensuring that the forms were validated before

saving the data. I overcame this by referring to the Django documentation and utilizing the built-in *ModelForm* validation methods.

3. **Working with Class-Based Views:** Transitioning from function-based views to class-based views took time to understand, especially regarding how context is managed. By reviewing the Django documentation and experimenting with various class-based views (CBVs), I gained a better understanding of their structure.

These exercises enabled me to build a fully functioning Django blog with dynamic features, including model relationships, user profile management, comment functionality, and a polished user interface with templates. While I faced challenges along the way, such as understanding migrations, working with class-based views, and handling forms, I overcame each obstacle by consulting Django's comprehensive documentation and applying the concepts I learned through trial and error. The final project not only functions as expected but also demonstrates a strong understanding of Django's core features, including models, views, forms, and templates.

Conclusion

This report outlines the development of a simple blog project using Django, showcasing the effectiveness of its core components—Models, Views, and Templates—in building dynamic, data-driven web applications. By creating models, managing user interactions through views, and using templates for content presentation, I built a fully functional blog.

Django Models provided a structured approach to database management, enabling easy interaction without raw SQL queries. I established flexible relationships between models, such as many-to-many between posts and categories, and simplified data queries with a custom manager.

In *Django Views*, I used both function-based and class-based views, highlighting the reusability of class-based views. User authentication limited access to specific features, enhancing the application's security.

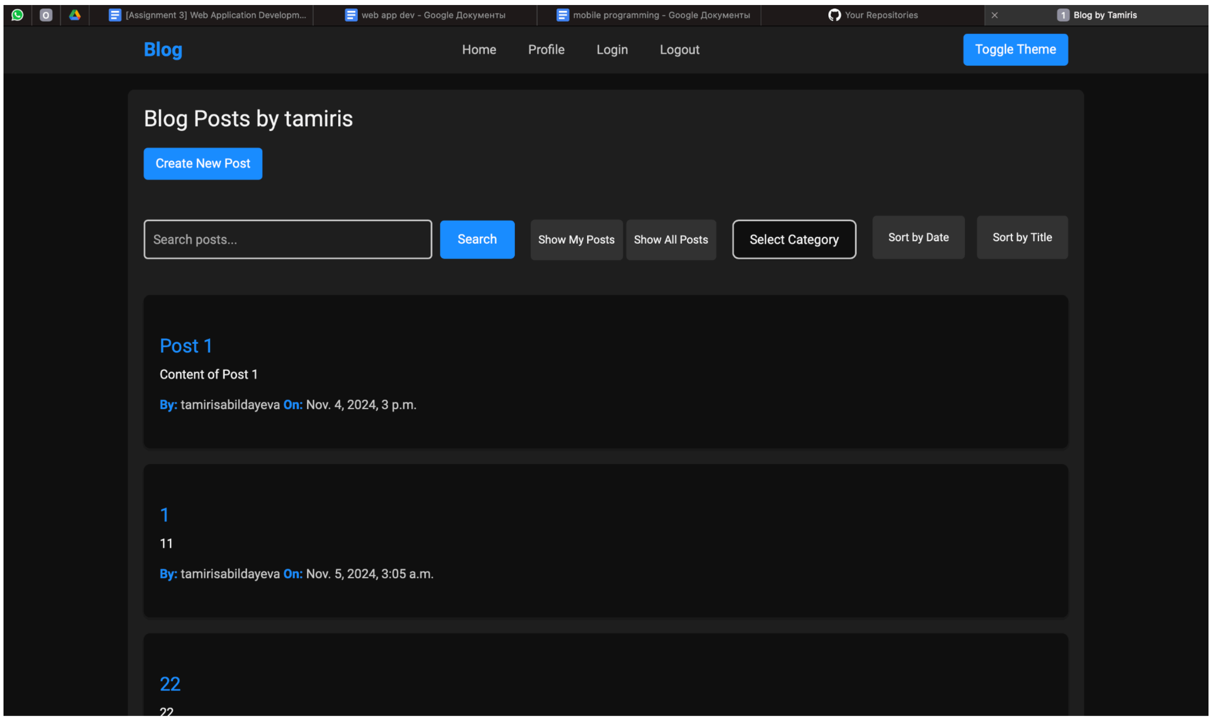
The *Django Templates* system helped me separate content presentation from business logic, making updates easier. Template inheritance allowed for a consistent design, while static files improved the blog's visual appeal.

This project reinforced the importance of Django's design philosophy, which separates concerns for better maintainability and scalability. Overall, this experience has deepened my understanding of Django and prepared me to build more complex web applications in the future.

References

1. https://docs.djangoproject.com/en/stable/
2. https://docs.djangoproject.com/en/stable/topics/db/models/
3. https://docs.djangoproject.com/en/stable/topics/http/views/
4. https://docs.djangoproject.com/en/stable/topics/templates/
5. https://docs.djangoproject.com/en/stable/topics/forms/
6. https://docs.djangoproject.com/en/stable/topics/auth/
7. https://stackoverflow.com/
8. https://stackoverflow.com/questions/tagged/django
9. https://docs.djangoproject.com/en/stable/ref/debugging/
10. https://docs.djangoproject.com/en/stable/topics/migrations/#migration-issues
11. https://docs.djangoproject.com/en/stable/ref/templates/builtins/#django.template.Library.simple_tag
12. https://forum.djangoproject.com/
13. https://docs.djangoproject.com/en/stable/topics/auth/default/
14. https://docs.djangoproject.com/en/stable/misc/design-philosophies/
15. https://docs.python.org/3/tutorial/errors.html
16. & others

Appendix


Blog Posts by tamiris

Post 1

Content of Post 1

By: tamirisabildayeva On: Nov. 4, 2024, 3 p.m.

1

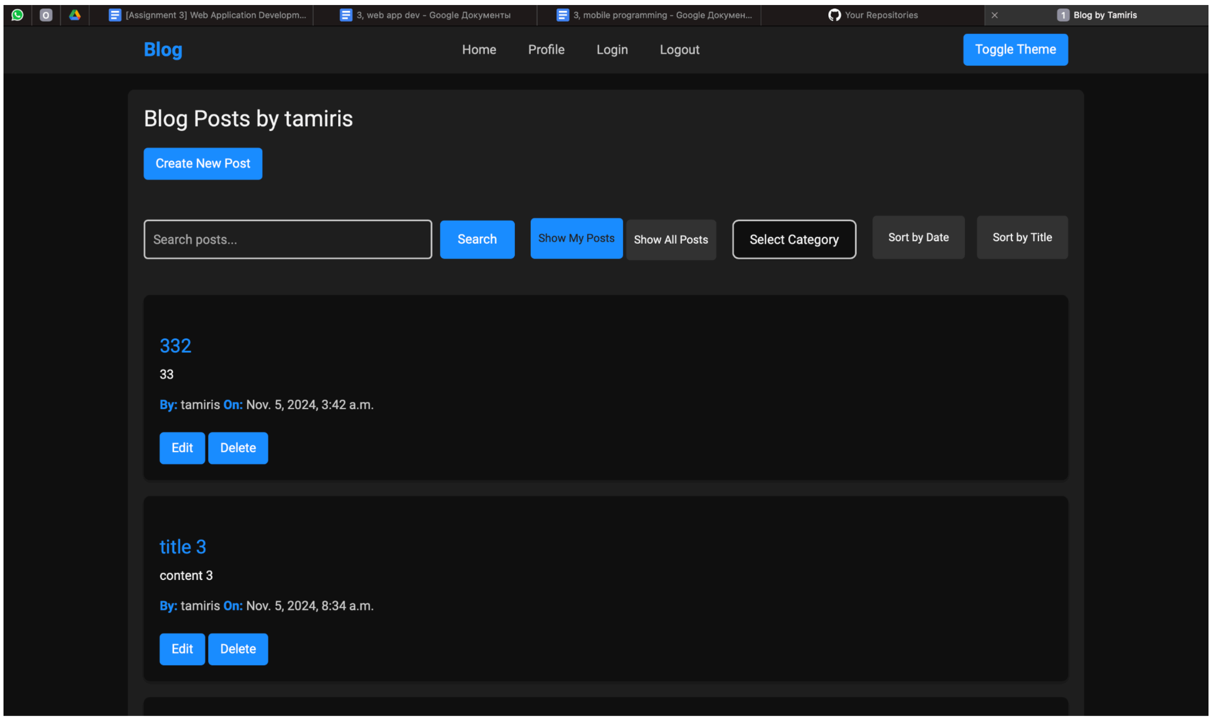
11

By: tamirisabildayeva On: Nov. 5, 2024, 3:05 a.m.

22

22

Fig. 27: Screenshot of the main page


Blog Posts by tamiris

332

33

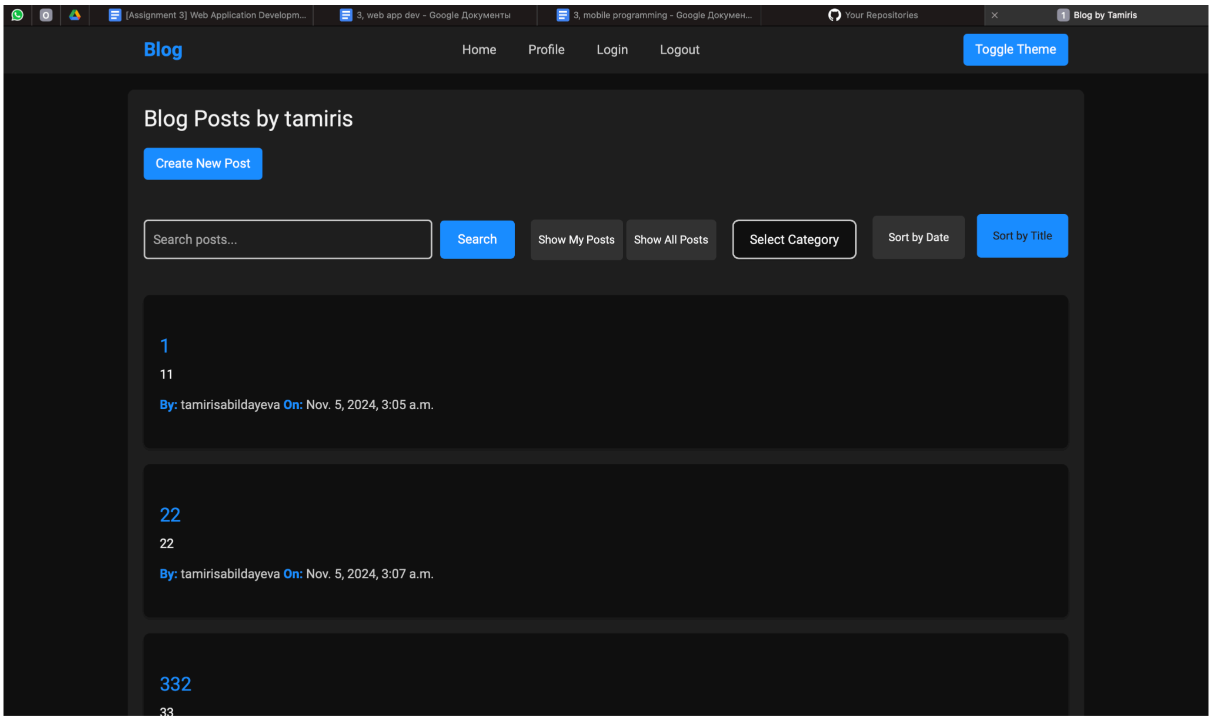
By: tamiris On: Nov. 5, 2024, 3:42 a.m.

title 3

content 3

By: tamiris On: Nov. 5, 2024, 8:34 a.m.

Fig. 28: Screenshot of the main page when clicking on “Show My Posts”


Blog Home Profile Login Logout Toggle Theme

Blog Posts by tamiris

Create New Post

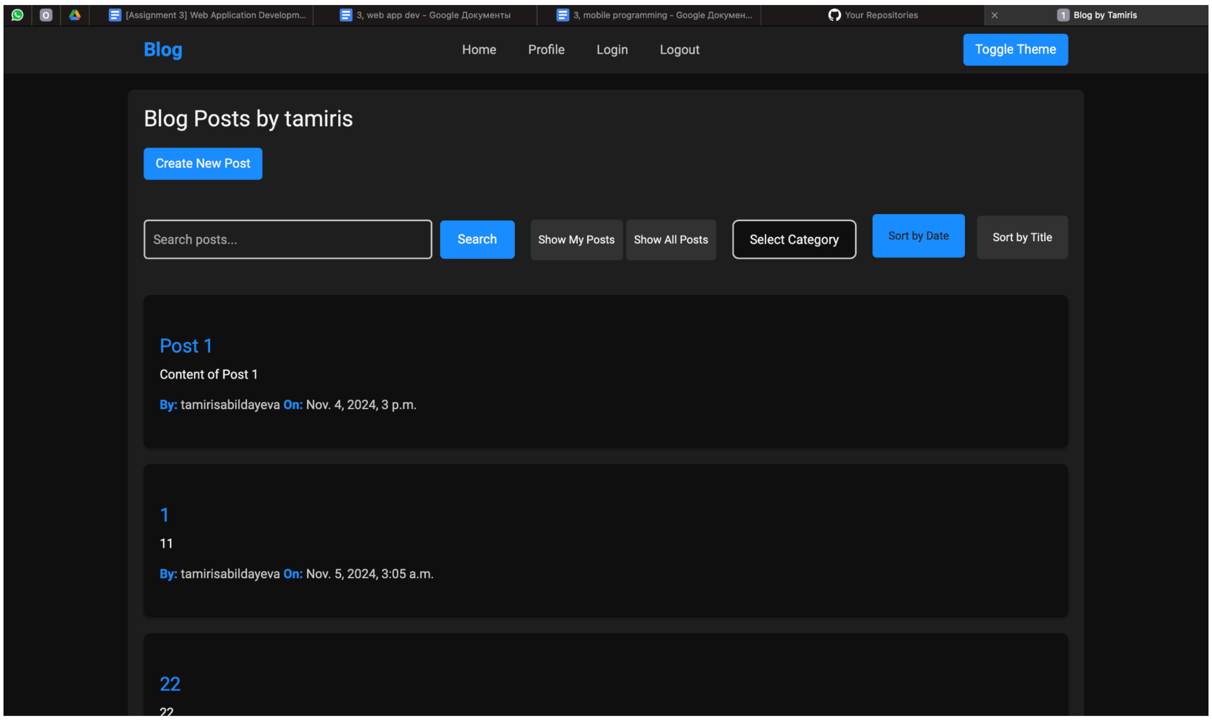
Search posts... Search Show My Posts Show All Posts Select Category Sort by Date Sort by Title

1
11
By: tamirisabildayeva On: Nov. 5, 2024, 3:05 a.m.

22
22
By: tamirisabildayeva On: Nov. 5, 2024, 3:07 a.m.

332
33

Fig. 29: Screenshot of the main page when clicking on “Sort by Title”


Blog Home Profile Login Logout Toggle Theme

Blog Posts by tamiris

Create New Post

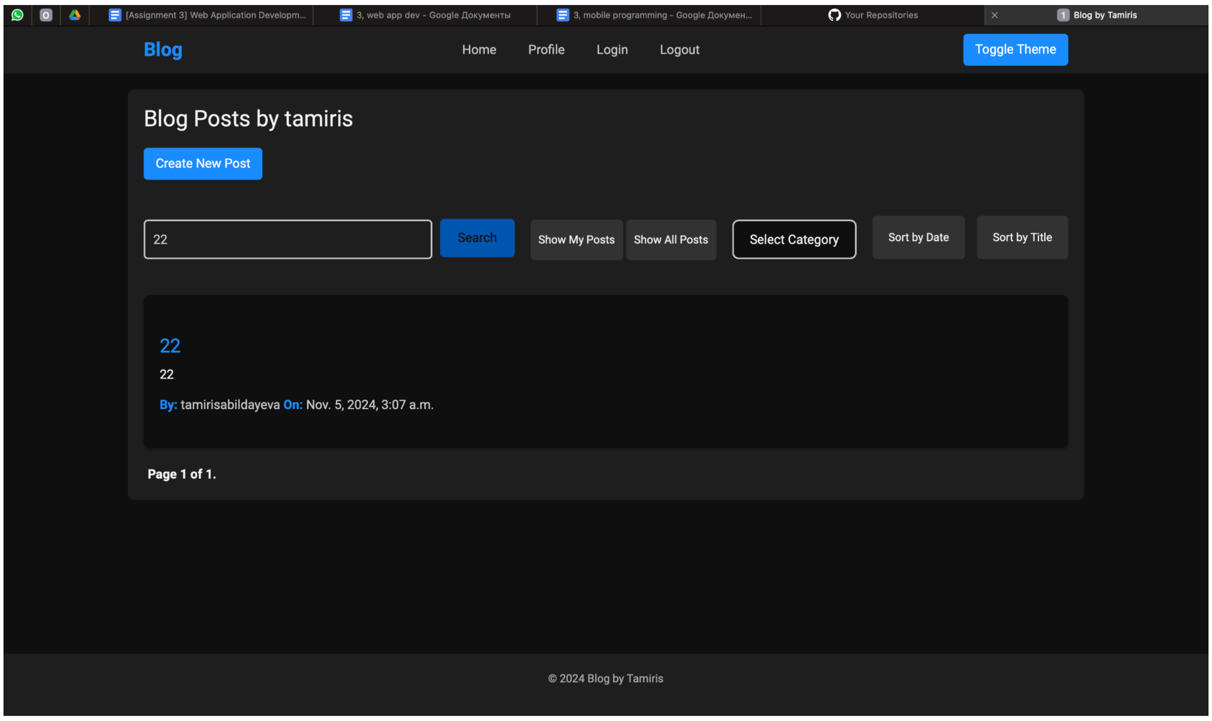
Search posts... Search Show My Posts Show All Posts Select Category Sort by Date Sort by Title

Post 1
Content of Post 1
By: tamirisabildayeva On: Nov. 4, 2024, 3 p.m.

1
11
By: tamirisabildayeva On: Nov. 5, 2024, 3:05 a.m.

22
22

Fig. 30: Screenshot of the main page when clicking on “Sort by Date”


[Assignment 3] Web Application Developm...
3, web app dev - Google Документы
3, mobile programming - Google Докумен...
Your Repositories
Blog by Tamiris
Blog
Home
Profile
Login
Logout
Toggle Theme

Blog Posts by tamiris

22

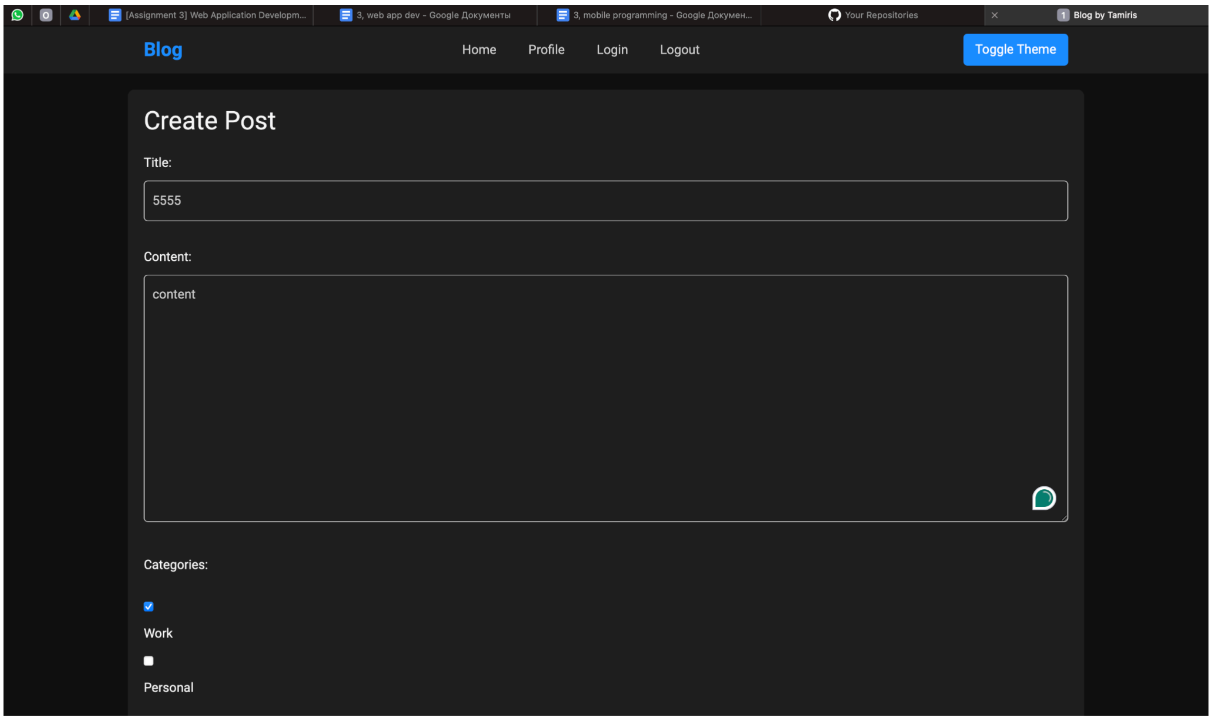
22

By: tamirisabildayeva On: Nov. 5, 2024, 3:07 a.m.

Page 1 of 1.

© 2024 Blog by Tamiris

Fig. 31: Screenshot of the main page when searching “22”


[Assignment 3] Web Application Developm...
3, web app dev - Google Документы
3, mobile programming - Google Докумен...
Your Repositories
Blog by Tamiris
Blog
Home
Profile
Login
Logout
Toggle Theme

Create Post

Title: 5555

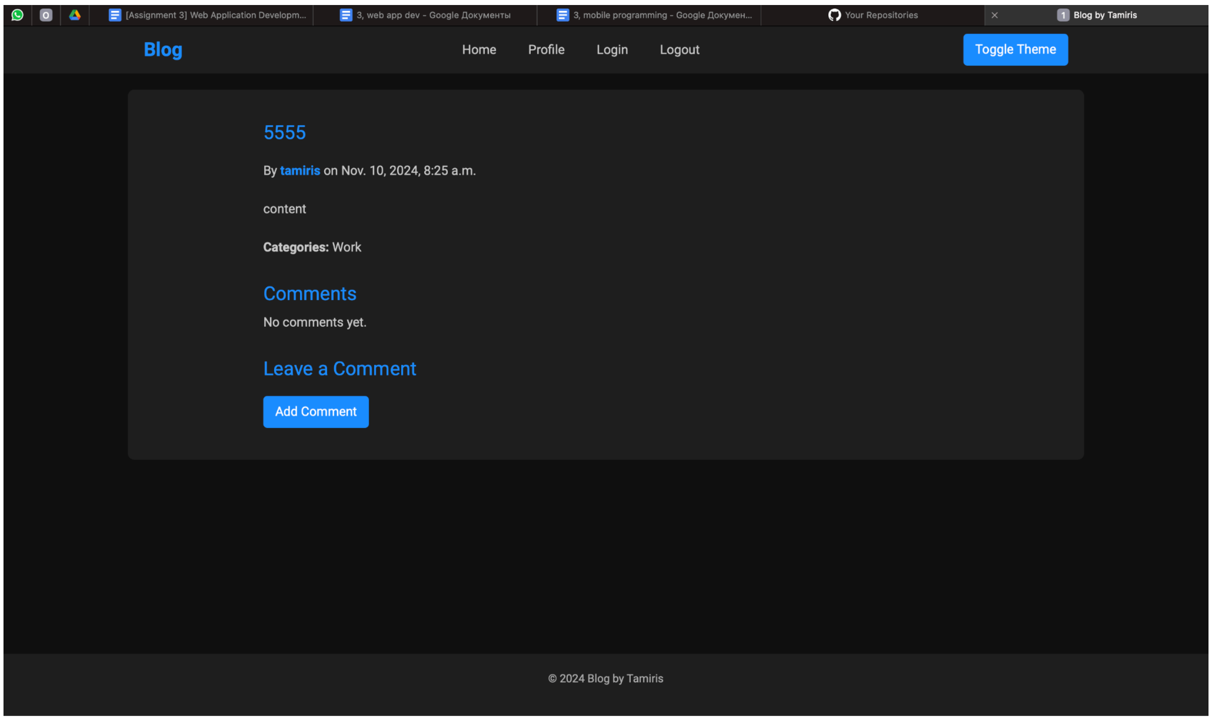
Content: content

Categories:

[✓] Work

[ ] Personal

Fig. 32: Screenshot of the “Create Post” page


[Assignment 3] Web Application Developm...
3, web app dev - Google Документы
3, mobile programming - Google Докумен...
Your Repositories
Blog by Tamiris
Blog
Home
Profile
Login
Logout
Toggle Theme

5555

By tamiris on Nov. 10, 2024, 8:25 a.m.

content

Categories: Work

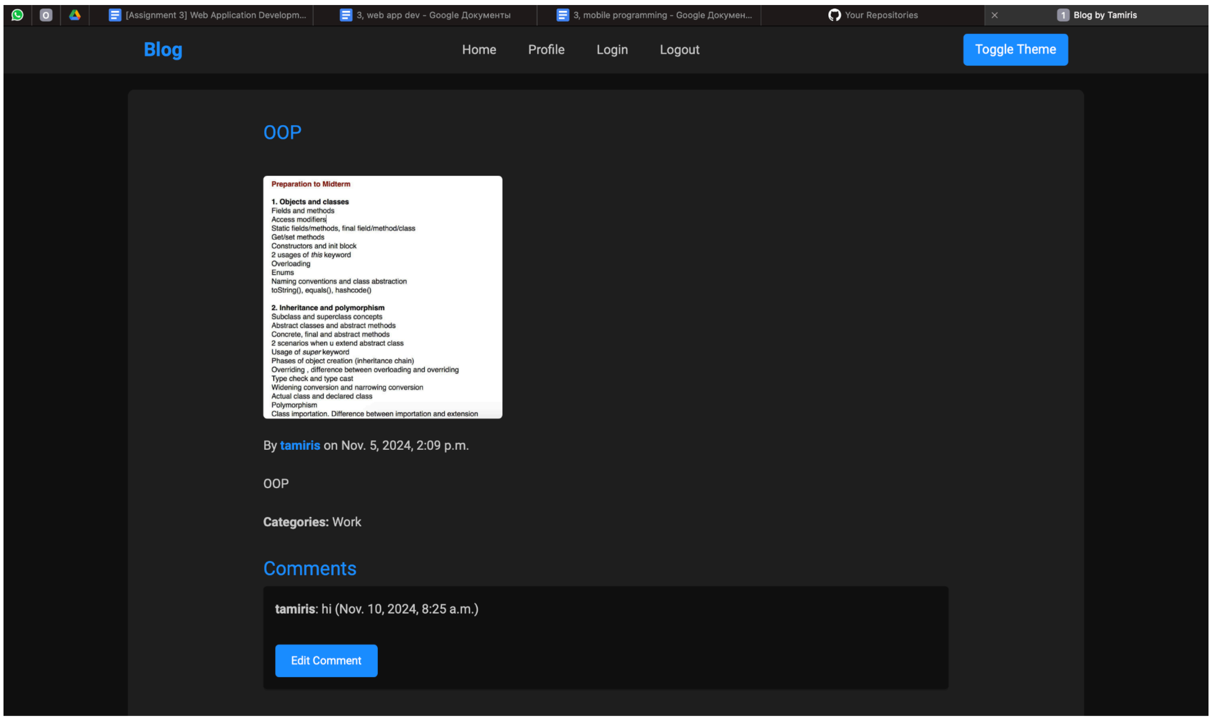
Comments

No comments yet.

Leave a Comment

© 2024 Blog by Tamiris

Fig. 33: Screenshot of the result after creating a new post


[Assignment 3] Web Application Developm...
3, web app dev - Google Документы
3, mobile programming - Google Докумен...
Your Repositories
Blog by Tamiris
Blog
Home
Profile
Login
Logout
Toggle Theme

OOP

Preparation to Midterm

1. Objects and classes
Fields and methods
Access modifiers
Static fields/methods, final field/method/class
Get/set methods
Constructors and init block
2 usages of *this* keyword
Overloading
Enums
Naming conventions and class abstraction
toString(), equals(), hashcode()

2. Inheritance and polymorphism
Subclass and superclass concepts
Abstract classes and abstract methods
Concrete, final and abstract methods
2 scenarios when u extend abstract class
Usage of *super* keyword
Phases of object creation (inheritance chain)
Overriding , difference between overloading and overriding
Type check and type cast
Widening conversion and narrowing conversion
Actual class and declared class
Polymorphism
Class importation. Difference between importation and extension

By tamiris on Nov. 5, 2024, 2:09 p.m.

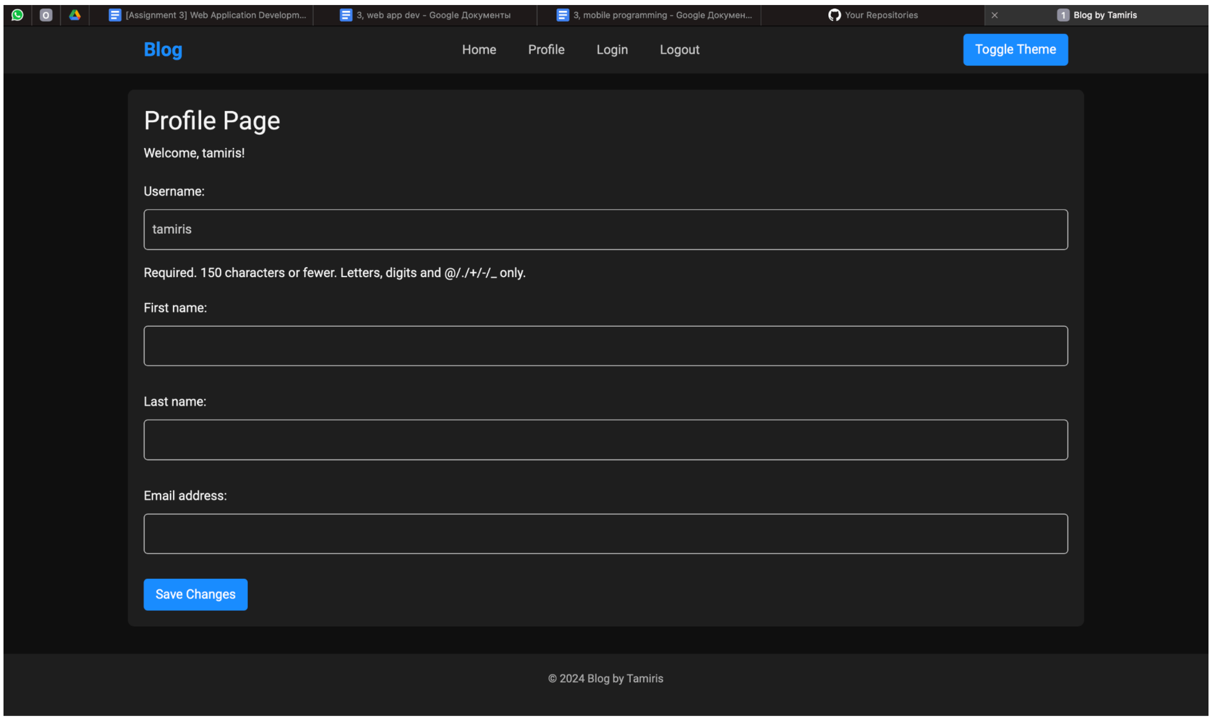
OOP

Categories: Work

Comments

tamiris: hi (Nov. 10, 2024, 8:25 a.m.)

Fig. 34: Screenshot of the "Post" page after adding new comment


[Assignment 3] Web Application Developm...
3, web app dev - Google Документы
3, mobile programming - Google Докумен...
Your Repositories
Blog by Tamiris
Blog
Home
Profile
Login
Logout
Toggle Theme

Profile Page

Welcome, tamiris!

Username:

tamiris

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

First name:

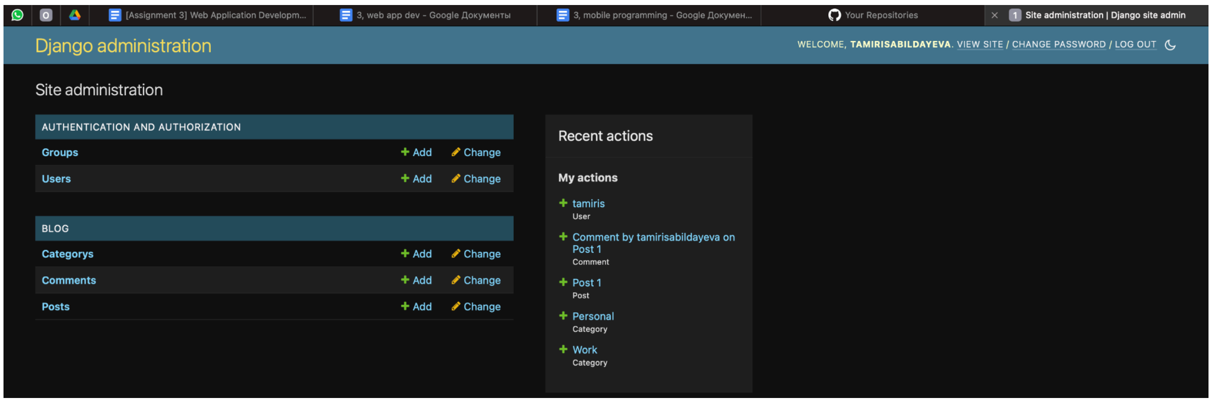
Last name:

Email address:

Save Changes

© 2024 Blog by Tamiris

Fig. 35: Screenshot of the “Profile” page


[Assignment 3] Web Application Developm...
3, web app dev - Google Документы
3, mobile programming - Google Докумен...
Your Repositories
Site administration | Django site admin
Django administration
WELCOME, TAMIRISABILDAYEVA. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

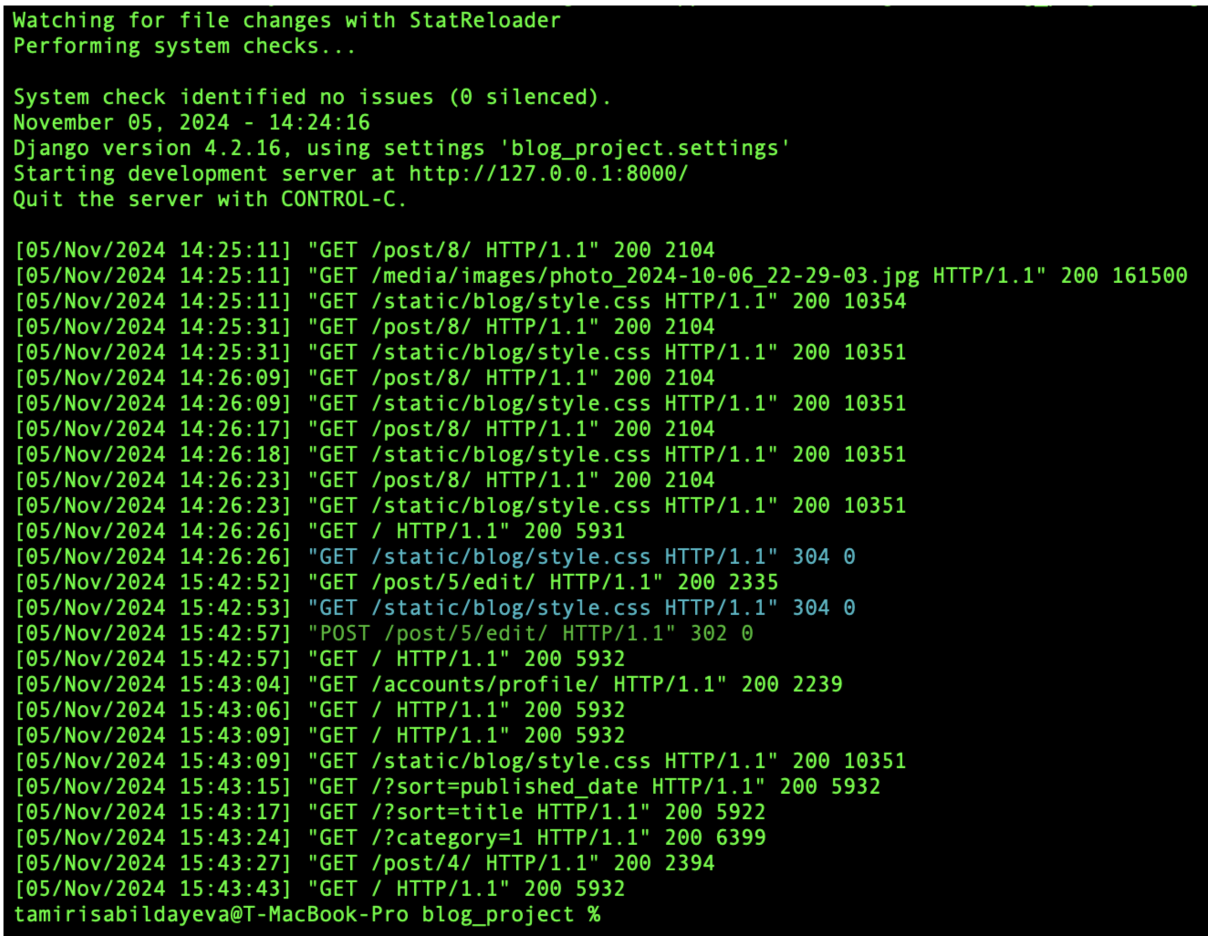
BLOG		
Categorys	+ Add	Change
Comments	+ Add	Change
Posts	+ Add	Change

Recent actions

My actions

+ tamiris
  User
+ Comment by tamirisabildayeva on Post 1
  Comment
+ Post 1
  Post
+ Personal
  Category
+ Work
  Category

Fig. 36: Screenshot of the admin panel page



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

System check identified no issues (0 silenced).
November 05, 2024 - 14:24:16
Django version 4.2.16, using settings 'blog_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

[05/Nov/2024 14:25:11] "GET /post/8/ HTTP/1.1" 200 2104
[05/Nov/2024 14:25:11] "GET /media/images/photo_2024-10-06_22-29-03.jpg HTTP/1.1" 200 161500
[05/Nov/2024 14:25:11] "GET /static/blog/style.css HTTP/1.1" 200 10354
[05/Nov/2024 14:25:31] "GET /post/8/ HTTP/1.1" 200 2104
[05/Nov/2024 14:25:31] "GET /static/blog/style.css HTTP/1.1" 200 10351
[05/Nov/2024 14:26:09] "GET /post/8/ HTTP/1.1" 200 2104
[05/Nov/2024 14:26:09] "GET /static/blog/style.css HTTP/1.1" 200 10351
[05/Nov/2024 14:26:17] "GET /post/8/ HTTP/1.1" 200 2104
[05/Nov/2024 14:26:18] "GET /static/blog/style.css HTTP/1.1" 200 10351
[05/Nov/2024 14:26:23] "GET /post/8/ HTTP/1.1" 200 2104
[05/Nov/2024 14:26:23] "GET /static/blog/style.css HTTP/1.1" 200 10351
[05/Nov/2024 14:26:26] "GET / HTTP/1.1" 200 5931
[05/Nov/2024 14:26:26] "GET /static/blog/style.css HTTP/1.1" 304 0
[05/Nov/2024 15:42:52] "GET /post/5/edit/ HTTP/1.1" 200 2335
[05/Nov/2024 15:42:53] "GET /static/blog/style.css HTTP/1.1" 304 0
[05/Nov/2024 15:42:57] "POST /post/5/edit/ HTTP/1.1" 302 0
[05/Nov/2024 15:42:57] "GET / HTTP/1.1" 200 5932
[05/Nov/2024 15:43:04] "GET /accounts/profile/ HTTP/1.1" 200 2239
[05/Nov/2024 15:43:06] "GET / HTTP/1.1" 200 5932
[05/Nov/2024 15:43:09] "GET / HTTP/1.1" 200 5932
[05/Nov/2024 15:43:09] "GET /static/blog/style.css HTTP/1.1" 200 10351
[05/Nov/2024 15:43:15] "GET /?sort=published_date HTTP/1.1" 200 5932
[05/Nov/2024 15:43:17] "GET /?sort=title HTTP/1.1" 200 5922
[05/Nov/2024 15:43:24] "GET /?category=1 HTTP/1.1" 200 6399
[05/Nov/2024 15:43:27] "GET /post/4/ HTTP/1.1" 200 2394
[05/Nov/2024 15:43:43] "GET / HTTP/1.1" 200 5932
tamirisabildayeva@T-MacBook-Pro blog_project %
```

Fig. 37: Logs in the terminal when the Django application runs.