

MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF
KAZAKHSTAN

KAZAKH-BRITISH TECHNICAL UNIVERSITY

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING

REPORT

Web Application Development

Assignment 3

Presented to Serek A.G.

Done by Kabyl Dauren

Student ID: 23MD0452

Almaty 2024

Table of contents

1.	Introduction	3
2.	Exercise Descriptions	4
2.1.	Django Models	4
2.1.1.	Creating a Basic Model	4
2.1.2.	Model Relationships	4
2.1.3.	Custom Manager	5
2.2.	Django Views	5
2.2.1.	Function-Based Views	5
2.2.2.	Class-Based Views	6
2.2.3.	Handling Forms	7
2.3.	Django Templates	8
2.3.1.	Basic Template Rendering	8
2.3.2.	Template Inheritance	8
2.3.3.	Static Files and Media	9
3.	Code Snippets	11
3.1.	Django Models	11
3.2.	Django Views	12
3.3.	Django Templates	14
4.	Results	17
5.	Conclusion	18
6.	References	19

Introduction

Django is a high-level framework for Python that allows you to easily and quickly create web applications [1]. In the modern world, the Django Framework is widespread and is used in such large projects as YouTube, Discord, Pinterest, etc.

The purpose of this work is to study the basics of creating a client-server architecture for a web application. To create our application we will use MVT (Model-View-Template) architectural style. We will study what Django Models and Views are, and how they are used to render HTML page templates. In section 2, we will go through each point in detail and explain how they work.

Objectives:

1. Understanding what is Django Models;
2. Creating basic Django Models;
3. Learning about Django View;
4. Creating views to our models;
5. Studying about Django Templates;
6. Creating and rendering templates to our views;
7. Creating a simple blog application.

Exercise Description

Django Models

Creating a Basic Models

Objectives:

1. Creating django *blog* application;
2. Understanding Django Models creation;
3. Defining Post model;
4. Migrating model to database.

First, we need to create our django project with command **django-admin startproject <project name>**. This command will create the basic structure of our project. After that to create a django application we need to execute the command **python manage.py startapp <application name>**. This command will create our *blog* application where we would define our models. To define our models we need to go to **models.py** file and create class **Post** with fields *author*, *title*, *content*, *published_at*, *likes* and *dislikes* that extends django class *Model*. Extending the *Model* class shows to Django that our class is a model which needs to be migrated to our database. Without this Django can't create migration to our Post model. To apply our model to the database we need to execute the **python manage.py makemigrations** command to make the migration file and **python manage.py migrate** to migrate it to the database. We use PostgreSQL database.

Model Relationships

Objectives:

1. Learning models relationships;
2. Defining Category model;
3. Creating Many-to-Many relationship between Post and Category;
4. Defining Comment model;
5. Creating a foreign key in Comment to Post.

To understand relationships between models we need to create two new models **Category** with field *name* and **Comment** with fields *author*, *content*, *created_at* and *post*. The *post* field in the *Comment* model is a foreign key to the *Post* model. After that, we need to add new field *categories* to our *Post* model and define it as a Many-to-Many relationship field. After migration with **python manage.py makemigrations** and **python manage.py migrate** commands we can see our new tables and new field *categories* in our *Post* table.

On *Image 1*, we can see not only our created migrations but also default migrations prepared by Django.

```
(web-dev) PS C:\Users\admin\PycharmProjects\assignment3-web-dev> python manage.py makemigrations
Migrations for 'blog':
  blog/migrations\0001_initial.py
    + Create model Category
    + Create model Post
    + Create model Comment
(web-dev) PS C:\Users\admin\PycharmProjects\assignment3-web-dev> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying blog.0001_initial... OK
  Applying sessions.0001_initial... OK
(web-dev) PS C:\Users\admin\PycharmProjects\assignment3-web-dev>
```

Image 1. Initial migration

Custom Manager

Objectives:

1. Learning about Models Manager;
2. Creating Manager for Post model;
3. Defining some methods with Manager.

To implement a custom manager to our Post we created a PostManager class that extends the Manager class from Django. In this class we defined 2 methods:

1. Method **by_author** - returns list of posts by specific author;
2. Method **by_category** - returns list of posts by specific category.

After that, we need to rewrite the default objects field that came from Model class. To do that we simply put our new PostManager class to objects.

Django Views

Function-Based Views

Objectives:

1. Understanding what is function-based views;
2. Implementing function-based views to create posts and post detailed views.

To implement function-based views we need to create functions in our **views.py** file. Each function that we created necessarily needs to take a **request** as the first parameter. Our function-based view for list of all posts takes all views from the database and renders **posts.html** page with them. In this page we just show titles of posts. Each title is a hyperlink and when we click on it we will be redirected to the details page of this post. Our function-based view for post details takes **post_id** as the second argument and gets this post and comments on this post by post id from the database. In the **post_details.html** page we can see the title, authors name, published date, post content and all comments of this post. Next, we need to add new urlpatterns to our **urls.py** file. To get initial data for our database we added *initial_data.json* file with data for our models and executed the command **python manage.py loaddata blog/fixtures/initial_data.json**. After that we can see our rendered pages from the browser.

Class-Based Views

Objectives:

1. Understanding what is class-based views;
2. Refactoring our function-based views to class-based views.
3. Learning differences between function-based and class-based views.

To refactor our previously created function-based views to class-based views we need to create two classes in our **views.py** file:

1. **PostListView** - view class that shows the page of all posts. Extends **ListView** class from Django. In the class fields we need to show that the model is Post and template is **posts.html** and context name is posts.
2. **PostDetailView** - view class that shows the page of post detail. Extends **DetailView** class from Django. In the class fields we need to show that the model is Post and template is **post_details.html** and context name is post. To add comments to context we need to override **get_context_data** method and add comments context to it.

Blog Posts

- [Getting Started with Django](#)
- [Introduction to Python](#)
- [Django Models](#)
- [Python Functions](#)
- [Weather application with Django](#)
- [Calculator on Python](#)

Image 2. All posts page

Getting Started with Django

Author: [Alikhan Bokeikhanuly](#)

Categories: Python Django Framework

Published on: November 01, 2024

Likes: 24351, Dislikes: 334

Django is a high-level Python web framework that encourages rapid developm source.

Comments:

Shoqan Ualikhanov - November 01, 2024 13:00

Great article on Django!

Image 3. Post details page

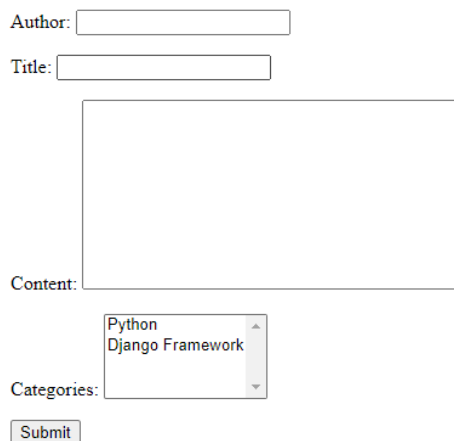
Handling Forms

Objectives:

1. Learning how to create forms in Django;
2. Creating simple page for form;
3. Implementing form for creating new posts.

To create new posts for our blog we need a form to input new data. To create a form we created a **PostForm** class in **forms.py** file. Inside it we created an inner **Meta** class where we defined model and input fields. In our case input fields are author, title, content and categories of post. After that we created a **post_form.html** page to show our form and **create_post** view to render it. Finally, we added a new url to this view and link to the form page in the posts page. You can see the form on *Image 4*.

Create New Post



Author:

Title:

Content:

Categories:

Image 4. Form for creating new post

Django Templates

Basic Template Rendering

Objectives:

1. Learning about Django Templates;
2. Understanding how to add data to templates;
3. Rendering templates to pages.

Django Templates works with HTML pages, where with code we can add some data. Django will automatically render the page and require data to it. We updated our **posts.html** page with adding post author and formatted publication date. The result you can see on *Image 5*.

Blog Posts

- [Getting Started with Django](#)

By Alikhan Bokeikhanuly | Published on November 01, 2024

- [Introduction to Python](#)

By Akhmet Baitursynuly | Published on November 05, 2024

- [Django Models](#)

By Alikhan Bokeikhanuly | Published on November 01, 2024

- [Python Functions](#)

By Akhmet Baitursynuly | Published on November 05, 2024

- [Weather application with Django](#)

By Khalel Dosmukhamedov | Published on June 01, 2024

- [Calculator on Python](#)

By Turar Rysqulov | Published on June 05, 2024

- [Django Views](#)

By Abai Qunanbaiuly | Published on November 08, 2024

[Create New Post](#)

Image 5. Updated Posts Page

Template Inheritance

Objectives:

1. Understanding about template inheritance;
2. Creating base page with header and footer;
3. Extending all previously created pages with the base page.

One template can inherit structure from another template and just define his own content block. This helps to reduce copy code from HTML pages. We defined our **base.html** structure and added two blocks for title and content. We can redefine these blocks inside other HTML pages. To apply this base structure to our posts page we need to extend posts.html with base.html. After that we need to redefine two blocks to apply the content of posts. Now we can see changes in our posts pages. As shown on *Image 6*, header and footer from base.html applied to posts page. We applied this base page to all of our pages.

Blog

[Home](#)

- [Getting Started with Django](#)

By Alikhan Bokeikhanuly | Published on November 01, 2024

- [Introduction to Python](#)

By Akhmet Baitursynuly | Published on November 05, 2024

- [Django Models](#)

By Alikhan Bokeikhanuly | Published on November 01, 2024

- [Python Functions](#)

By Akhmet Baitursynuly | Published on November 05, 2024

- [Weather application with Django](#)

By Khalel Dosmukhamedov | Published on June 01, 2024

- [Calculator on Python](#)

By Turar Rysqulov | Published on June 05, 2024

- [Django Views](#)

By Abai Qunanbauly | Published on November 08, 2024

[Create New Post](#)

© 2024 Blog

Image 6. Posts page with header and footer

Static Files and Media

Objectives:

1. Creating CSS styles file for HTML pages;
2. Applying styles.css to header and footer;
3. Creating media directory;
4. Adding opportunity to adding image to post.

Django can search static files in the project by **static** directory name. So, to apply all static files like CSS we need to add it inside this directory. After that Django will automatically determine these files. We created **styles.css** file in `/static/css/` directory. Inside it we defined styles for header

and footer. After that we applied these styles to our base.html file. The results you can see on *Image 6*.

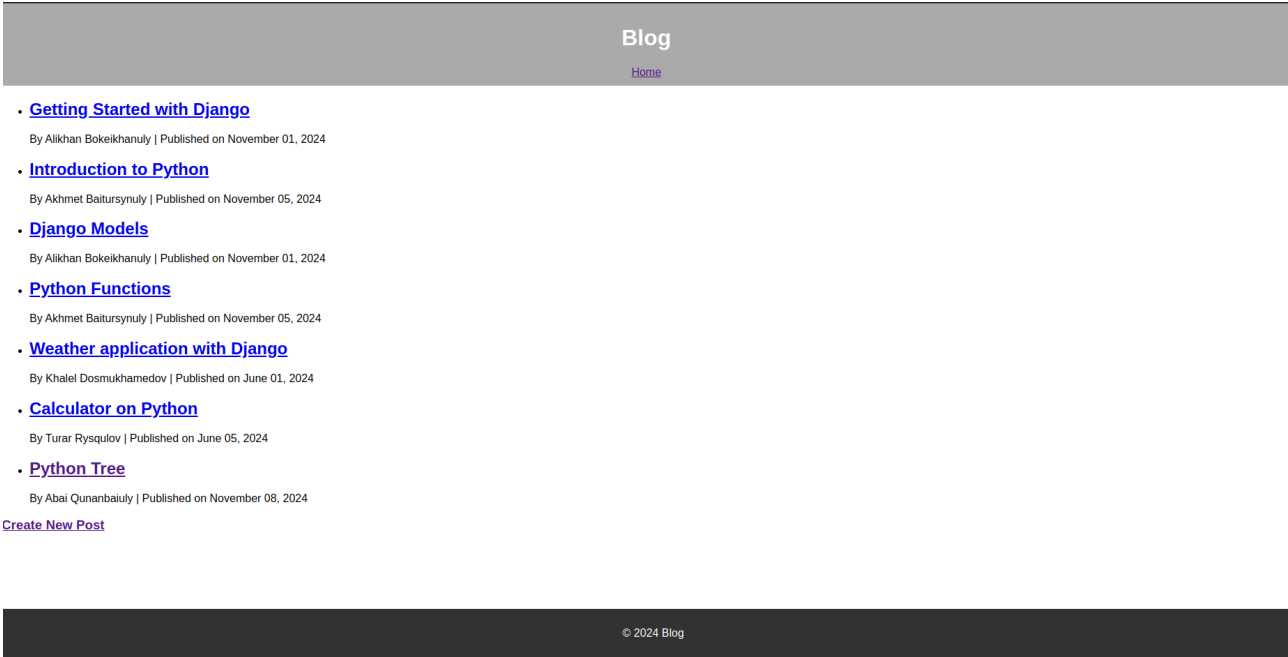


Image 6. Posts page with applied CSS styles

To add an opportunity to the user to include some image to the post we add a new field **image** to Post model and in the form and views we added sending/handling **multipart/form_data**. All files will be saved inside **/media/blog/** directory. The results you can see on *Image 7*.

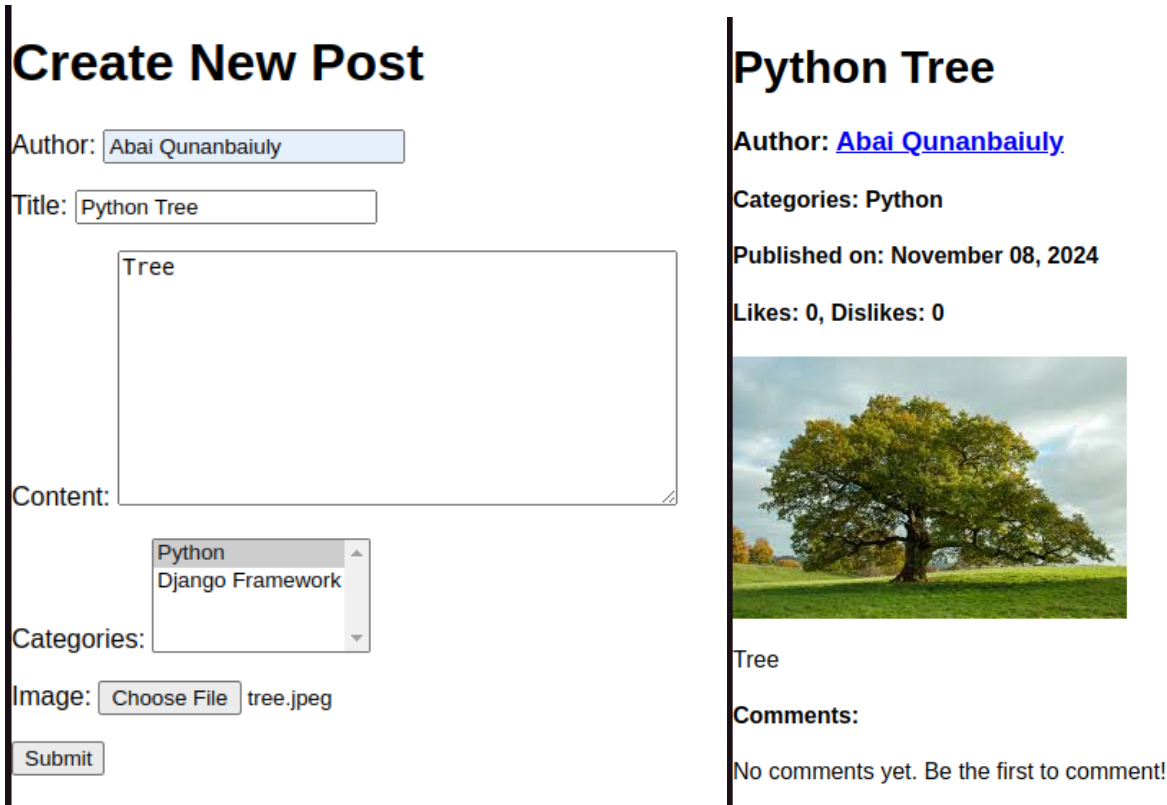


Image 7. Adding image to post

Code Snippets

Django Models

On the *Images 8-10* you can see the models that we created in this project. Post model has author, title, content, published_at, likes & dislikes, categories and image fields. Categories field is Many-to-Many relation field to Category model. Category model has only a name field. Comment model has author, content, created_at and post fields. Post field is foreign key that defines the Many-to-One relationship to the Post model. Each model has a method that returns a string representation of this model.

```
class Post(models.Model):
    author = models.CharField(max_length=255)
    title = models.CharField(max_length=255)
    content = models.TextField()
    published_at = models.DateTimeField(auto_now_add=True)
    likes = models.IntegerField(default=0)
    dislikes = models.IntegerField(default=0)
    categories = models.ManyToManyField(Category, related_name='posts')
    image = models.ImageField(upload_to='blog/%Y/%m/%d', blank=True, null=True)

    objects = PostManager()

    def __str__(self):
        return f'Author: {self.author}, Title: {self.title}'
```

Image 8. Post model

```
class Category(models.Model):
    name = models.CharField(max_length=255)

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

Image 9. Category model

```
class Comment(models.Model):
    author = models.CharField(max_length=255)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    post = models.ForeignKey(Post, on_delete=models.CASCADE, related_name='comments')

    def __str__(self):
        return f'Author: {self.author}, Post: {self.post}'
```

Image 10. Comment model

On *Image 11*, you can see the manager class for posts. It extends the `Manager` class and defines two methods. **by_author** method returns posts by author name and **by_category** method returns posts by category.

```
class PostManager(models.Manager):
    def by_author(self, author):
        return self.get_queryset().filter(author=author)

    def by_category(self, category):
        return self.get_queryset().filter(categories=category)
```

Image 11. PostManager custom manager

Django Views

On *Image 12*, you can see the function-based views that we created for our project.

1. **get_posts** function returns the rendered page of all posts.
2. **get_post_details** function returns the rendered page of the post details page with comments.
3. **create_post** function renders the post form page for GET requests and handles data from form for POST requests and redirects to the home page.

```
def get_posts(request):
    posts = Post.objects.all()
    return render(request, template_name="posts.html", context={"posts": posts})

def get_post_details(request, post_id):
    post = get_object_or_404(Post, pk=post_id)
    comments = Comment.objects.filter(post=post)
    return render(request, template_name="post_details.html", context={"post": post, "comments": comments})

<</blog/create/
def create_post(request):
    if request.method == "POST":
        form = PostForm(request.POST, request.FILES)
        if form.is_valid():
            form.save()
            return redirect(get_posts)
    else:
        form = PostForm()
    return render(request, template_name="post_form.html", context={"form": form})
```

Image 12. Function-Based Views

On *Image 13*, you can see the class-based views of our project.

1. **PostListView** - refactored class-based type of `get_posts`.
2. **PostDetailView** - refactored class-based type of `get_post_details`. To add comments to the context it overrides the **get_context_data** method.
3. **PostsByAuthor** - returns a rendered page with all posts of the specific author. To search posts by author it overrides the **get_queryset** method and applies **by_author** method from our `PostManager` class.

```

class PostListView(ListView):
    model = Post
    template_name = "posts.html"
    context_object_name = "posts"

<@ /blog/{pk}/
class PostDetailView(DetailView):
    model = Post
    template_name = "post_details.html"
    context_object_name = "post"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["comments"] = Comment.objects.filter(post=self.object).order_by("-created_at")
        return context

<@ /blog/author/{author}/
class PostsByAuthorView(ListView):
    model = Post
    template_name = "posts_by_author.html"
    context_object_name = "posts"

    def get_queryset(self):
        author = self.kwargs.get("author")
        return Post.objects.by_author(author)

```

Image 13. Class-Based Views

On *Image 14*, you can see the form class of the Post model. It extends the ModelForm class from django forms. Inside it there is an inner Meta class where we define form. **model** field indicates Model that will be used and **fields** field defines which field of the model will be included to the form. In our case it's *author*, *title*, *content*, *categories* and *image* fields.

```

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['author', 'title', 'content', 'categories', 'image']

```

Image 14. Post creating form

On *Image 15*, you can see all the urls that we created in the project. We have a basic url for the all posts page, a url with path pk for post details page, url with author to posts by author page and create url to add post page. Each url pattern has an url address, view that will be rendered and the name of the pattern that can be used in hyperlinks inside the HTML templates.

```
urlpatterns = [
    path('blog/', views.PostListView.as_view(), name='get_posts'),
    path('blog/{pk}/', views.PostDetailView.as_view(), name='get_post_details'),
    path('blog/author/{author}/', views.PostsByAuthorView.as_view(), name='posts_by_author'),
    path('create/', views.create_post, name='create_post'),
]
```

Image 15. Urls of the project

Django Templates

As shown on *Image 16*, our **base.html** file has the basic structure of our pages. It has the header, content block and footer. Inside the header we have the application name and navigation link to the home page. Content block it's the part that will be overridden with each extended page. Inside the footer we just have the copyright sign, current year and application name. This file also loads the status styles.css file and applies styles from it.

```
{% load static %}

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

<main>
    {% block content %}{% endblock %}
</main>

<footer>
    <p>© {% now "Y" %} Blog</p>
</footer>
</body>
</html>
```

Image 16. Base template

As shown on *Image 17*, the content block includes a list for all posts with post title, author name and published date and link to create a post page. Each post title is a link to this post details page.

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

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

{% block content %}
    <ul>
        {% for post in posts %}
            <li>
                <h2><a href="{% url 'get_post_details' post.id %}">{{ post.title }}</a></h2>
                <p>By {{ post.author }} | Published on {{ post.published_at|date:"F d, Y" }}</p>
            </li>
        {% endfor %}
    </ul>

    <h3><a href="{% url 'create_post' %}">Create New Post</a></h3>
{% endblock %}
```

Image 17. All posts page template

On *Image 18*, we can see the content of the post details page. It includes post title, author, categories, published date, likes & dislikes, post image, post content and comments section. Each comment has the author's name, published date and content.

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

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

{% block content %}
    <h1>{{ post.title }}</h1>
    <h3>Author: <a href="{% url 'posts_by_author' post.author %}"> {{ post.author }} </a></h3>
    <h4>Categories:
        {% for category in post.categories.all %}
            {{ category.name }}
        {% endfor %}
    </h4>
    <h4>Published on: {{ post.published_at|date:"F d, Y" }}</h4>
    <h4>Likes: {{ post.likes }}, Dislikes: {{ post.dislikes }}</h4>
    {% if post.image %}
        
    {% endif %}

    <p>{{ post.content }}</p>
    <h4>Comments:</h4>
    {% if comments %}
        {% for comment in comments %}
            <p>
                <strong>{{ comment.author }}</strong> - {{ comment.created_at|date:"F d, Y H:i" }}
            </p>
            <p>{{ comment.content }}</p>
        {% endfor %}
    {% else %}
        <p>No comments yet. Be the first to comment!</p>
    {% endif %}

    <p></p>
{% endblock %}
```

Image 18. Post details page template

On *Image 19*, we can see content of the posts by author page. It includes the author's name and list of posts. Each post has a title, published date and shortened content.

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

{% block title %}Posts by Author{% endblock %}

{% block content %}
    <h1>Posts by {{ posts.0.author }}</h1>
    <ul>
        {% for post in posts %}
            <li>
                <h2><a href="{% url 'get_post_details' post.id %}">{{ post.title }}</a></h2>
                <p>Published on {{ post.published_at|date:"F d, Y" }}</p>
                <p>{{ post.content|truncatewords:30 }}</p>
            </li>
        {% endfor %}
    </ul>
{% endblock %}
```

Image 19. Posts by author page template

On *Image 20*, we can see the content of the create post page. It includes a form for creating a post. We defined the form method as POST and encryption type as multipart/form-data that can send files with json. **csrf_token** inside it is needed for security encryption of this form. Submit button sends input data from form to our server side.

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

{% block title %}Create post{% endblock %}


{% block content %}
     <h1>Create New Post</h1>
    <form method="post" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Submit</button>
    </form>
{% endblock %}
```

Image 20. Create post form page template

Results

Django Models

In the exercises of the Django Models section we created 3 models for our application and defined relationships between them. We created a basic easy to understand structure of our application with these models and migrated them to the database. Also, we created a custom manager for the Post model to easily implement search on them.

Django Views

In the exercises of the Django Views section we defined views for our models. We learned how to create function-based and class-based views. This helps us to understand how we can easily create views for our application. Also, we defined a form for creating posts and created a view for it. Creating a form in Django is not a difficult process.

Django Templates

In the exercises of the Django Templates section we created base page structure and pages for all our views. We learned how to inherit templates and how to add information from context to page. We learned how to render HTML pages in Django and how to add CSS styles to them. Also, we learned to upload and save media from requests.

Conclusion

In this work we learned about Django Models and Models Relationships, how to create custom manager class, how to create Function-Based and Class-Based views, how to handle input data from forms, how to create and render HTML pages and how to apply CSS styles to them and how to save uploaded media. In the end we get our simple Blog application where we can see the blog posts with authors, see the blog details with comments, see the list of posts by author and add a new blog post etc. We learned many new things about Django Framework. This new knowledge will help us in our future work.

References

1. Meet Django, URL: <https://www.djangoproject.com/>
2. Django Models, URL: https://tutorial.djangogirls.org/en/django_models/
3. How to Define Relationships Between Django Models. URL: <https://www.freecodecamp.org/news/django-model-relationships/>
4. A Comprehensive Guide to Django Views: Understanding Types and Use Cases. URL: <https://medium.com/@satyarepala/a-comprehensive-guide-to-django-views-understanding-types-and-use-cases-4a02a078ced>
5. Django Templating: A Beginner's Guide to Django Templates. URL: <https://medium.com/@devsumitg/django-templating-a-beginners-guide-to-django-templates-5bc1c558574e>
6. Mastering Django Forms: A Comprehensive Guide for Beginners. URL: <https://medium.com/@devsumitg/mastering-django-forms-a-comprehensive-guide-for-beginners-193665f1408d>