KBTU KAZAKH-BRITISH TECHNICAL UNIVERSITY

Assignment 3

Web Application Development

Building a Blog Application Using Django

Prepared by:
Seitbekov S.
Checked by:
Serek A.

Almaty, 2024

Table of Contents

Introduction

Web application development often involves extensive knowledge of how data is structured, managed, and presented to users. In the context of Django, a highly powerful Python web framework, this typically involves effectively using Models, Views, and Templates. This report describes a step-by-step process for setting up a blog application, starting with the very basic setup right through to the implementation of key functionality-model creation, handling views, rendering templates, and processing forms. Every exercise is done with the intention not only of creating functional components but also of considering their meaning in the greater scheme of web application development.

Django Models

Exercise 1: Creating a Basic Model

The first exercise was to develop the basic Post model for the Django blog application. It should have a field for a title, content, author, and date published, and it should have one method that returns the string representation. I knew that in Django, data needs to have a proper structure, so I started by creating a new Django app called blog inside of the project directory for the encapsulation of model.

This is the model for a Post which should be the core of this blog app (blog/models.py), and it needs to incorporate with the help of Django's ORM for the interactions with the PostgreSQL database in my case. I make such a model with CharField for the title, TextField for the content, ForeignKey relating it to the User (Django built in user) model for author details, and DateTimeField to facilitate an optional publish date and ImageField for storing images in post. Also, I implemented the __str__ method to return the title of each post in human readable format making easier process of debugging, logging, etc.

```python
class Post(models.Model):  13 usages
    title = models.CharField(
        max_length=100, verbose_name="Название", help_text="Максимум 100 символов"
    )
    content = models.TextField(
        blank=True, verbose_name="Контент", help_text="Текст статьи"
    )
    author = models.ForeignKey(
        User, on_delete=models.CASCADE, verbose_name="Автор", help_text="Автор статьи"
    )
    published_date = models.DateTimeField(
        auto_now_add=True,
        verbose_name="Дата публикации",
        help_text="Дата публикации статьи",
    )
    categories = models.ManyToManyField(
        Category, verbose_name="Категории", help_text="Категории статьи"
    )
    image = models.ImageField(upload_to="post_images/", null=True, blank=True)
    objects = PostManager()

    def __str__(self) -> str:
        return self.title

    def __repr__(self) -> str:
        return f"Post(title={self.title}, author={self.author}, published_date={self.published_date})"

    class Meta:
        verbose_name = "Статья"
        verbose_name_plural = "Статьи"
        ordering = ("-published_date",)
```

Figure 1. The Post Model

After defining the model, I created and applied migrations, reflecting the model schema in the database and establishing the foundational structure of the application’s data layer.

```
~/Documents/Git/WebProgrammingMS/Assignment-3/myproject on master ?1
❯ ./manage.py makemigrations
Migrations for 'blog':
  blog/migrations/0001_initial.py
    + Create model Post
```

Figure 2. Making migrations for Blog Model

According to screenshot I created tables with Django migrations commands. This exercise underscored the importance of Django models as a flexible yet powerful way to define data schemas that align with the relational structure of SQL databases. The creation of this model set the groundwork for managing blog data efficiently within the Django ecosystem, allowing seamless interaction with the database and streamlined data manipulation.

Exercise 2: Model Relationships

After the definition of the model of the Post, I extended it by adding other models that would enable the interaction with richer data. I defined the Category model and established a many-to-many relationship between Category and Post meaning one single post might belong to several categories. The Category model will have two fields for naming the category and description of category.

```python
from django.db import models
from django.contrib.auth.models import User

from blog.manager import PostManager


class Category(models.Model):  3 usages
    name = models.CharField(
        max_length=50, verbose_name="Название", help_text="Максимум 50 символов"
    )
    description = models.TextField(
        blank=True, verbose_name="Описание", help_text="Описание категории"
    )

    def __str__(self) -> str:
        return self.name

    def __repr__(self) -> str:
        return f"Category(name={self.name})"

    class Meta:
        verbose_name = "Категория"
        verbose_name_plural = "Категории"
        ordering = ("name",)
```

Figure 3. The Category Model

I then defined a Comment model representing the users' comments about specific posts and

attached it to Post and Author by using a foreign key. So, comment model will have a referenced post field, a referenced author field, a comment content field and a creation time. This is like many web applications where there is usually content divided into many categories and enriched with user comments.

```python
class Comment(models.Model):  2 usages
    post = models.ForeignKey(
        Post,
        on_delete=models.CASCADE,
        verbose_name="Статья",
        help_text="Статья к которой относится комментарий",
    )
    author = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        verbose_name="Автор",
        help_text="Автор комментария",
    )
    content = models.TextField(verbose_name="Контент", help_text="Текст комментария")
    published_date = models.DateTimeField(
        auto_now_add=True,
        verbose_name="Дата публикации",
        help_text="Дата публикации комментария",
    )

    def __str__(self) -> str:
        return f"{self.author} - {self.published_date}"

    def __repr__(self) -> str:
        return f"Comment(author={self.author}, published_date={self.published_date})"

    class Meta:
        verbose_name = "Комментарий"
        verbose_name_plural = "Комментарии"
        ordering = ("-published_date",)
```

Figure 4. The Comment Model

Such way I have modularity: each model can be changed out independently. Of course, migrations were created and applied to update the database schema accordingly. I can define complex data relationships using the Django ORM capabilities that could support interactive and dynamic data structures critical in modern content-focused web applications.

Exercise 3: Custom Manager

I used a custom manager for the Post model to extend its functionality for filtering and retrieving data based on standard use cases. This custom manager would include methods like returning only published posts and retrieving posts by an author. This would be considered

querying selectively because, in a real application situation, different users and permissions might require restricted views of the data.

I created the PostManager in blog/manager.py and integrated it with the Post model.

```python
from django.db import models
from django.db.models import QuerySet


class PostManager(models.Manager):  2 usages  new *
    def published(self) -> QuerySet:  new *          You, Yesterday • Uncommitted changes
        return self.filter(published_date__isnull=False)

    def by_author(self, author_username: str) -> QuerySet:  new *
        return self.filter(author__username=author_username)

```

Figure 5. The Custom PostManager

By adding the PostManager as a default manager in the Post, the querying of the model is improved in efficiency and readability. Because this custom manager allows for modular construction of queries, it saves a great deal of code elsewhere in the application, improving maintainability by reducing repetition.

Django Views

Exercise 4: Function-Based Views

I used the most straightforward way to handle the HTTP request for posts and details of the posts, which is function-based views - FBVs. I allowed these views to handle two major activities: one for listing all the published posts and the other for showing detailed individual posts based on their unique identifier. So, in blog/api/base/views.py, I created post_list to fetch and present all the published posts, and post_detail to show one post.

```python

def post_list_view(request: HttpRequest) -> HttpResponse:  2 usages
    posts = Post.objects.all()
    return render(request, template_name: "post_list.html", context: {"posts": posts})


def post_detail_view(request: HttpRequest, pk: int) -> HttpResponse:  2 usages
    post = get_object_or_404(Post, pk=pk)
    return render(request, template_name: "post_detail.html", context: {"post": post})

```

Figure 6. The Function Based View for list and detail view

These views are then mapped to URLs in blog/api/base/router.py accessible to the users in viewing posts.

```python

urlpatterns = [
    path("posts/", post_list_view, name="post_list"),
    path("posts/new/", PostCreateView.as_view(), name="post_new"),
    path("posts/<int:pk>/", post_detail_view, name="post_detail"),
]

```

Figure 7. The URL's for Blog Pages

This exercise demonstrated how easy it was to work with FBVs for simple data handling, where data could be made available and displayed directly in a human-readable format.

Exercise 5: Class-Based Views

I refactored some views from function-based to class-based to increase modularity and scalability. Therefore, by using Django's generic views, the code would be simpler and more reusable. I created PostListView and PostDetailView within blog/views.py by using ListView and DetailView, respectively, and these can deal with precisely the same functionality as before but with greater flexibility.

The class PostListView returns only published posts while the class PostDetailView returns a single post based on its ID.

```python
class PostListView(ListView):  2 usages
    model = Post
    template_name = "post_list.html"
    context_object_name = "posts"
    ordering = ["-published_date"]
    paginate_by = 10

    def get_queryset(self) -> QuerySet:
        return Post.objects.all()


class PostDetailView(DetailView):  2 usages
    model = Post
    template_name = "post_detail.html"
    context_object_name = "post"

    def get_queryset(self) -> QuerySet:
        return Post.objects.all()
```

Figure 8. The Class Based View for list and detail view

The use of CBVs cleaned the code, made it more readable, and allowed easy expansions in the future because Django has been designed to be modular.

Exercise 6: Handling Forms

So, user input handling needed to be done to enable creating posts. I therefore created a form using Django's ModelForm for validation and processing of new post data. In blog/forms.py, I defined PostForm to encapsulate all the required fields to make sure only valid data are processed and stored in the database.

```
1  from django import forms
2  from blog.models import Post
3
4
5  class PostForm(forms.ModelForm):  2 usages
6      class Meta:
7          model = Post
8          fields = ["title", "content", "author", "categories", "image"]
9
```

Figure 9. The Post Form for Creating New Posts

I used a class-based view called PostCreateView to handle form submissions. This uses the generic view CreateView that handles rendering and validation and saving of the form data. This view simply redirects to post list page upon form submission by user.

```
40
41  class PostCreateView(CreateView):  2 usages
42      model = Post
43      form_class = PostForm
44      template_name = "post_new.html"
45      success_url = reverse_lazy("post_list")
46
```

Figure 10. The Class Based View for Creating New Posts

This exercise has shown how form processing with Django is a lot easier to validate data and strengthen user experience through error handling while making the data processing secure.

Django Templates

Exercise 7: Basic Template Rendering

Template designs allow the display of dynamically generated content. I have created a post_list.html that shows the rendering list of posts showing titles of posts, authors, and format publication date. Django's template language makes it possible to embed dynamic data right inside the HTML, allowing an efficient interface for rendering. I'm extending base template, but I will talk about it in the next exercise.

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

{% block content %}
    <h2>Blog Posts</h2>
    <ul>
        {% for post in posts %}
            <li>
                <a href="{% url 'post_detail' post.pk %}">{{ post.title }}</a>
                by {{ post.author.username }}
                on {{ post.published_date|date:"F d, Y" }}
            </li>
        {% empty %}
            <li>No posts available.</li>
        {% endfor %}
    </ul>
{% endblock %}
```

Figure 11. The Base Template for Blog Pages

For individual post details, post_detail.html was created to display the title, author, content, category, and an optional image with comments if they provided. If not, then just show template text.

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

{% block content %}
    <h2>{{ post.title }}</h2>
    <p>By {{ post.author.username }}
        on {{ post.published_date|date:"F d, Y" }}
        in {{ post.categories.all|join:", " }}</p>
    <p>{{ post.content }}</p>
    {% if post.image %}
        <img src="{{ post.image.url }}" alt="Post Image">
    {% endif %}
    <hr>
    <h3>Comments</h3>
    {% for comment in post.comment_set.all %}
        <div class="comment">
            <p><strong>{{ comment.author.username }}</strong> on {{ comment.published_date|date:"F d, Y" }}</p>
            <p>{{ comment.content }}</p>
        </div>
    {% empty %}
        <p>No comments yet.</p>
    {% endfor %}
{% endblock %}
```

Figure 12. The Template for Blog Detailed Page

This dynamic rendering allows users to navigate and view posts in a structured format, improving user experience.

Exercise 8: Template Inheritance

To make the templates less cumbersome, I created an abstract base template for reusable HTML components like headers, navigation, and footer areas. It's named base.html and will be the standard base template where all the other templates can apply so that the styling and layout repeat with minimal redundancy.

```
{% extends 'base.html' %}   Sanzhar, Yesterday • added: Assignment 3

{% block content %}
    <h2>Diable Blog Posts</h2>
    <ul>
        {% for post in posts %}
            <li>
                <a href="{% url 'post_detail' post.pk %}">{{ post.title }}</a>
                by {{ post.author.username }}
                on {{ post.published_date|date:"F d, Y" }}
            </li>
        {% empty %}
            <li>No posts available.</li>
        {% endfor %}
    </ul>
{% endblock %}

```

Figure 13. The Template for Blog List or Home Page

Child templates will extend base.html; thus, changes in layout will be the same for all pages. This form of inheritance model is crucial for implementing a consistent design across the whole application.

Exercise 9: Static Files and Media

To give it more of a nice look and to be user interactive, I added custom styling via CSS and set up the application to allow user media uploads for images created. I've defined STATIC_URL and MEDIA_URL in settings.py for serving static and media files, and defined STATICFILES_DIRS and MEDIA_ROOT as directories containing static & media resources. The capability of Django to manage static and media files enables easy integration of these resources, important at least for any modern, user-friendly application.

```
92  STATIC_URL = "/static/"
93  STATICFILES_DIRS = [BASE_DIR / "static"]   Sanzhar, Yesterday • added: Assignment 3
94
95  MEDIA_URL = "/media/"
96  MEDIA_ROOT = BASE_DIR / "media"
97
```

Figure 14. Static and Media Directories configured in settings.py

These settings configure the setup so that CSS, JavaScript, and images used in the UI, would go into the /static directory, while user-uploaded images associated with posts, would go into the /media directory. This structure keeps resources organized and correctly served by Django.

```
1   * {
2       margin: 0;
3       padding: 0;
4       box-sizing: border-box;
5   }
6
7   html, body {
8       font-family: 'Helvetica Neue', Arial, sans-serif;
9       background-color: #1a1a1a;
10      color: #e0e0e0;
11      line-height: 1.6;
12      height: 100%;
13      margin: 0;
14      padding: 0;
15  }
16
17  .container {
18      min-height: calc(67vh);   Sanzhar, Yesterday • added: Assignment 3
19      padding: 20px;
20  }
21
22  header, footer {
23      background-color: #222;
24      color: #fff;
25      padding: 20px;
26      text-align: center;
27  }
```

Figure 15. CSS styles for Blog Pages

I developed a CSS in static/css/style.css file for the styling of the application. Within this file, I have styles for layout and typography, among other things that would assist in making the application fresh and cohesive. This enables me to style headers, navigation links, and a footer which will match from page to page while classes specific to post titles and content increase readability.

Results

The development of the blog application in Django integrated data modeling with view handling and template rendering into one coherent, interactive, user-friendly system. A structured approach has leveraged all aspects of the MVT framework in Django to solve different perspectives of application functionality that led to a modular, efficient, and scalable solution.

Each of these exercises was leveraging a previous one, established a sound data structure, efficiently handled queries, and dynamically provided content to the user via visually appealing templates. It also simplified query management with a custom manager and further aligned the application with best practices by migrating from function-based views to class-based views. Forms allowed the user to create new content with built-in validation, and template inheritance provided a consistent, maintainable UI across pages.

Overcame some challenges, mainly with handling complex model relationships and setting up media file compatibility. Integration of the many-to-many relationship between Post and Category, ensuring uploads of media files are served properly, has been done carefully by paying attention to Django's ORM and file-handling configurations. This was achieved by referencing the Django documentation and testing configurations in a systematic manner to produce a full workable application that could easily be expanded and maintained.

Conclusion

In conclusion, this project really drove home how necessary and how functional the MVT (Model-View-Template) architecture in Django can be in developing robust and scalable web applications. Indeed, Models did provide a robust framework for defining data schema and their relations, while Views facilitated controlled interactions between data and presentation. Templates completed the framework: delivering dynamic, user-friendly content that really drove the application both in functionality and appeal.

Through hands-on experience with features inclusive of custom managers, class-based views, form handling, and template inheritance, I better learned how each contributed something to the modularity, efficiency, and maintainability of the application. Adding to that, the treatment of static and media files further underlined the versatility of Django in the creation of an application which is also appealing to the eye. This structured approach cleaned up not only development but also brought scalability and usability of the application in every aspect. It provided me with broadened essential skills for any future web development projects.

References

1. Django Documentation - https://docs.djangoproject.com
2. CSS styles - https://developer.mozilla.org/en-US/docs/Web/CSS
3. HTML basics - https://developer.mozilla.org/en-US/docs/Web/HTML
4. Django templates - https://docs.djangoproject.com/en/5.1/topics/templates/

Appendices

Figure 16. Blog Add Page

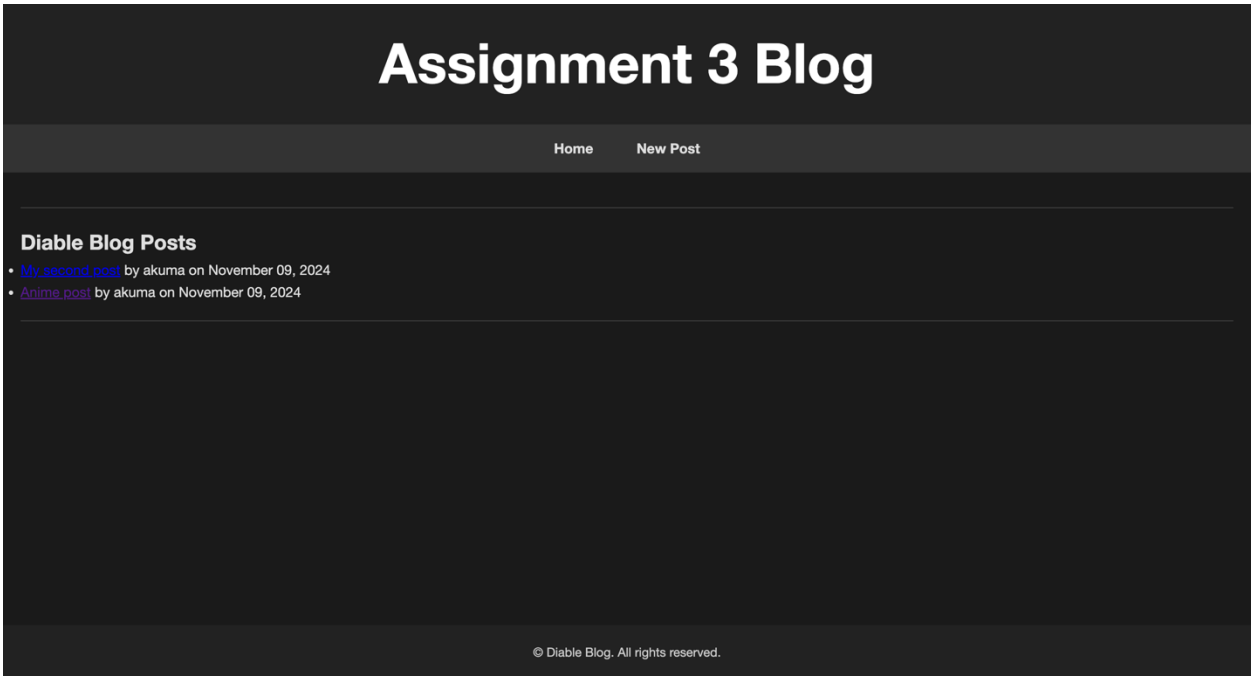

Figure 17. Blog Home Page

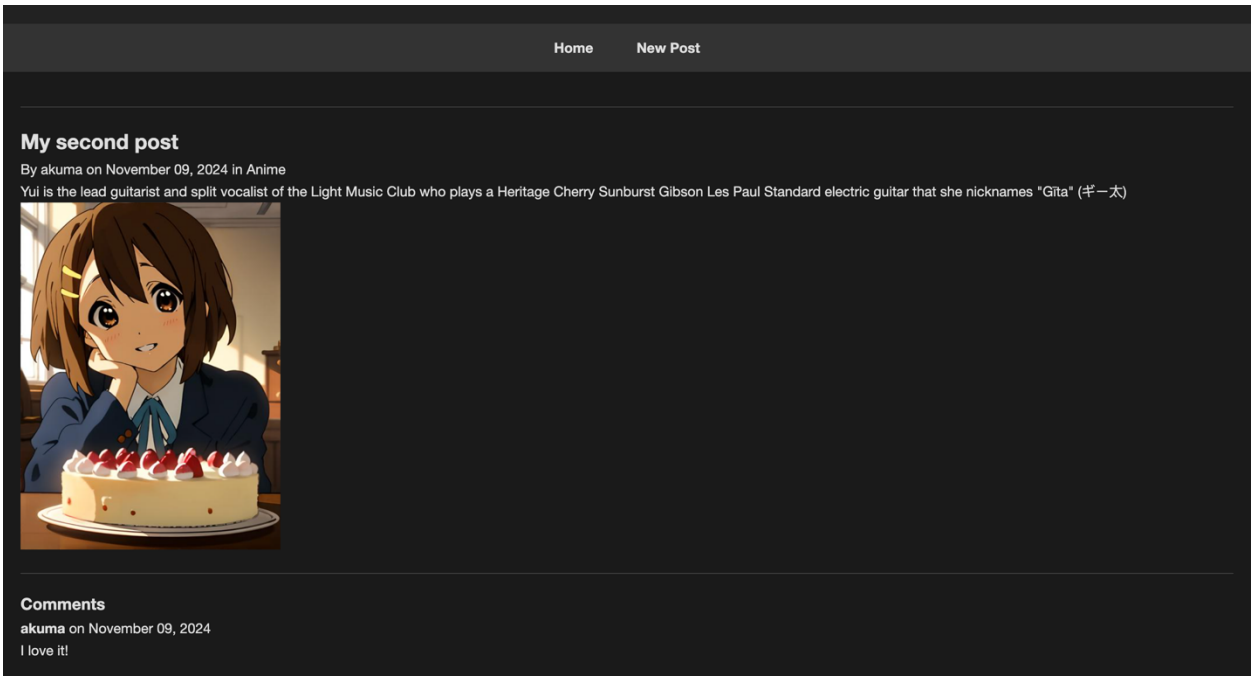

Figure 18. Blog Detailed Page with Comments

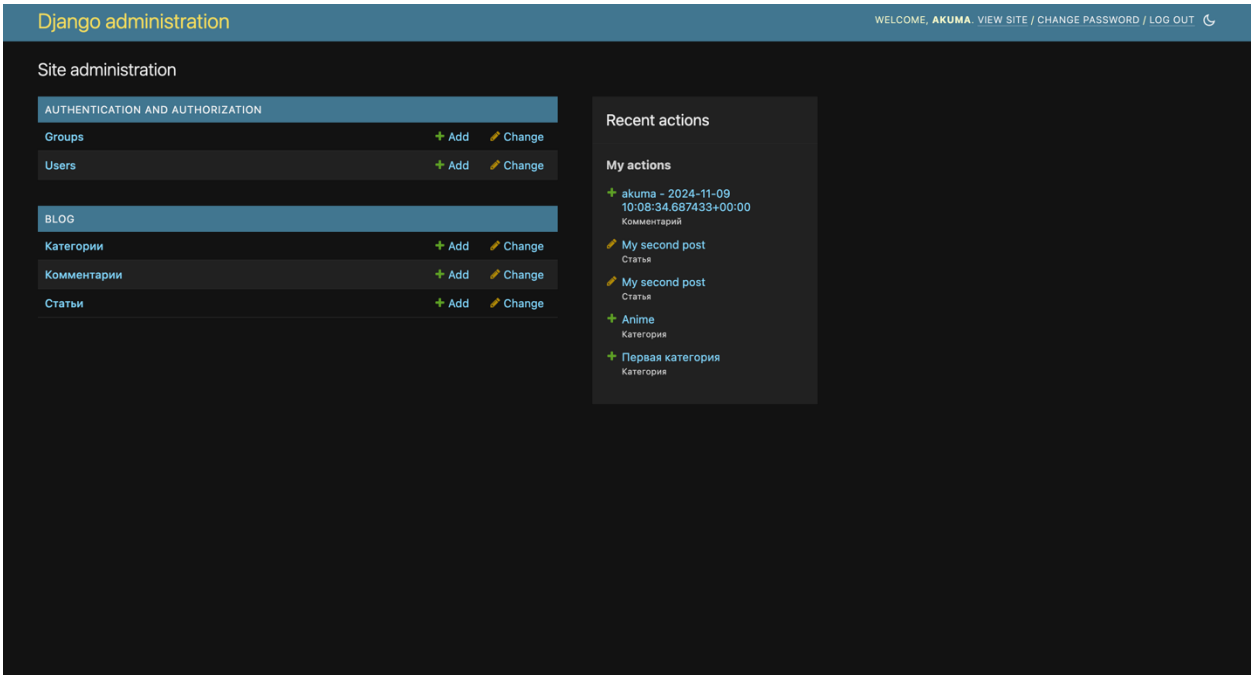

Figure 19. Django Admin Page

```python
@admin.register(Post)  Sanzhar
class PostAdmin(admin.ModelAdmin):
    """Admin panel configuration for the Post model."""
    list_display = ("title", "author", "published_date")
    list_filter = ("author", "published_date")
    search_fields = ("title", "content")
    ordering = ("-published_date",)
    fields = ("title", "content", "author", "published_date")
    readonly_fields = ("published_date",)
    list_per_page = 10


@admin.register(Category)  Sanzhar
class CategoryAdmin(admin.ModelAdmin):
    """Admin panel configuration for the Category model."""
    list_display = ("name", "description")
    search_fields = ("name",)
    ordering = ("name",)
    list_per_page = 10


@admin.register(Comment)  Sanzhar
class CommentAdmin(admin.ModelAdmin):
    """Admin panel configuration for the Comment model."""  Sanzhar, Yesterday • added: Assignment 3
    list_display = ("post", "author")
    list_filter = ("author",)
    search_fields = ("post__title", "content")
    ordering = ("-published_date",)
    fields = ("post", "author", "content", "published_date")
    readonly_fields = ("published_date",)
    list_per_page = 10

```

Figure 20. Django Admin Configuration

```python
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": os.getenv("POSTGRES_DB"),
        "USER": os.getenv("POSTGRES_USER"),
        "PASSWORD": os.getenv("POSTGRES_PASSWORD"),
        "HOST": os.getenv("POSTGRES_HOST"),
        "PORT": os.getenv("POSTGRES_PORT"),
    }
}

```

Figure 21. Database Configuration

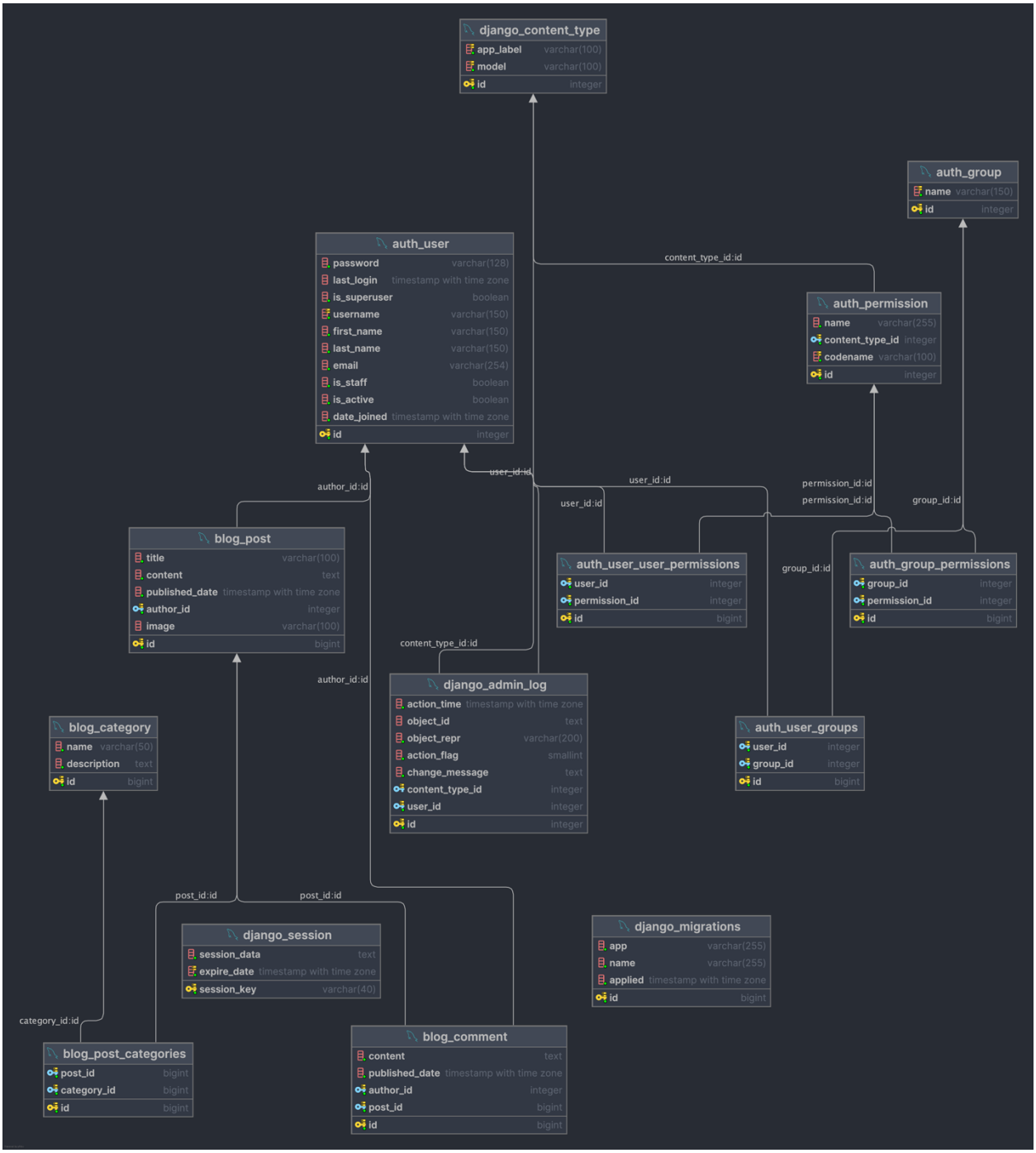

Figure 22. Database Schema

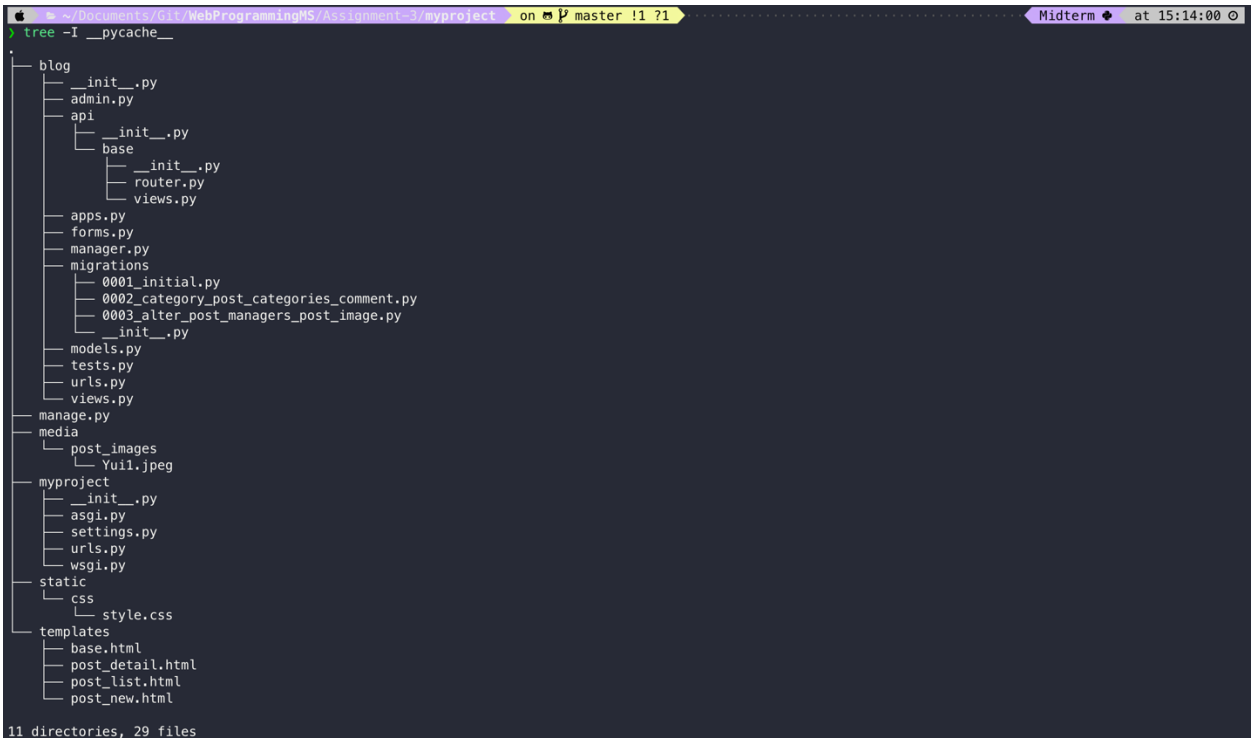

Figure 23. Project Structure

Figure 24. Aplying Migrations