Assignment 2

By: Amangeldi Amina

Exercise1:

Schemas Design)

```
ass2 > my_blog > blog > models.py > Tag > Meta
from django.db import models
from django.contrib.auth.models import User
from django.utils import timezone

class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(max_length=250, default='No bio')

    def __str__(self):
        return self.user.username


class Post(models.Model):
    title = models.CharField(max_length=150)
    content = models.TextField()
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    created_at = models.DateTimeField(default=timezone.now)
    tags = models.ManyToManyField('Tag', related_name='posts', blank=True)

    class Meta:
        indexes = [
            models.Index(fields=['author']),
        ]

    def __str__(self):
        return self.title

# Amangeldi Amina
```

```
ass2 > my_blog > blog > models.py > ...
class Post(models.Model):
        return self.title

# Amangeldi Amina

class Tag(models.Model):
    name = models.CharField(max_length=30)

    class Meta:
        indexes = [
            models.Index(fields=['name']),
        ]

    def __str__(self):
        return self.name


class Comment(models.Model):
    post = models.ForeignKey(Post, related_name='comments', on_delete=models.CASCADE)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    text = models.TextField(max_length=250, default='No comment')
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        indexes = [
            models.Index(fields=['post', 'created_at']),
        ]

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

# Amangeldi Amina
```

Here I created models that I will work in this project.

Indexing)

```
    class Meta:
        indexes = [
            models.Index(fields=['author']),
        ]

    def __str__(self):
        return self.title

# Amangeldi Amina
```

```
# Amangeldi Amina

class Tag(models.Model):
    name = models.CharField(max_length=30)

    class Meta:
        indexes = [
            models.Index(fields=['name']),
        ]

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

```
class Comment(models.Model):
    post = models.ForeignKey(Post, related_name='comments', on_delete=models.CASCADE)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    text = models.TextField(max_length=250, default='No comment')
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        indexes = [
            models.Index(fields=['post', 'created_at']),
        ]

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

# Amangeldi Amina
```

Applied indexes for faster searches and filtering when querying posts, comments based on fields.

Query Optimization:

```
def post_list(request):
    posts = Post.objects.prefetch_related('comments').all().order_by('-created_at')
    paginator = Paginator(posts, 5)
    page_number = request.GET.get('page')
    page_obj = paginator.get_page(page_number)
    return render(request, 'list.html', {'page_obj': page_obj})
```

The ORM query fetches all posts and their comments in just a couple of SQL queries. This cuts down on database trips compared to fetching comments for each post separately, leading to faster load times and better performance overall. Using prefetch_related makes data retrieval more efficient, improving the app's speed.

Optimization Report:

Indexes help improve query performance by acting like shortcuts in a library, allowing the database to find posts quickly when filtering by fields like author and tags. This is especially useful for large datasets. A composite index on the Comment model for post and created_at further speeds up searches for comments tied to specific posts, reducing the time needed to sift through all comments. To boost performance even more, consider denormalization, which combines related data into one table for faster reads, though it can complicate updates.

Additionally, using select_related is beneficial when you frequently need related data, as it fetches them in one go, while prefetch_related efficiently retrieves many-to-many and one-to-many relationships in a single trip to the database. Tools like the Django Debug Toolbar can help identify slow queries, allowing for further optimization. By combining these strategies with effective indexing, your Django app can achieve significantly better performance.

Exercise2:

Basic Caching)

```python
@cache_page(60)
def post_list(request):
    posts = Post.objects.prefetch_related('comments').all().order_by('-created_at')
    paginator = Paginator(posts, 5)
```

Here I used Django's built-in caching decorators to cache the view that displays the list of blog posts.

Template Fragment Caching)

```
<div class="tags">
    <strong>Tags:</strong>
    {% load cache %}
    {% cache 60 tags_for_post post.id %}
    {% for tag in post.tags.all %}
        <span>{{ tag.name }}</span>{% if not forloop.last %}, {% endif %}
    {% empty %}
        <span>No tags</span>
    {% endfor %}
    {% endcache %}
</div>

<p>{{ post.content }}</p>

<div class="comments-section">
    <h2>Comments</h2>

    {% cache 60 comments_for_post post.id %}
    {% if comments %}
        <ul>
            {% for comment in comments %}
                <li class="comment-item">
                    <strong>{{ comment.author.username }}</strong> on {{ comment.created_at }}<br>
                    {{ comment.text }}
                </li>
            {% endfor %}
        </ul>
    {% else %}
        <p>No comments yet.</p>
    {% endif %}
    {% endcache %}
```

Both the tags for the post and the comments section are cached for 60 seconds, improving performance by quickly serving frequently accessed content.

Low-Level Caching:

```python
def post_detail(request, id):
    post = get_object_or_404(Post.objects.prefetch_related('comments'), id=id)
    comments_count = cache.get(f'comments_count_{post.id}')

    if comments_count is None:
        comments_count = post.comments.count()
        cache.set(f'comments_count_{post.id}', comments_count, timeout=60)

    comments = post.comments.all().order_by('-created_at')
    # Amangeldi Amina
    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()
            cache.delete(f'comments_count_{post.id}')
            return redirect('post_by_id', id=post.id)
```

My template will show the number of comments efficiently using low-level caching.

Cache Backend)

```
Installing collected packages: redis, django-redis
Successfully installed django-redis-5.4.0 redis-5.1.1

[notice] A new release of pip is available: 23.2.1 -> 24.2
[notice] To update, run: python.exe -m pip install --upgrade pip
(venv) PS C:\Users\amina\KBTU\back-for-highload\ass2\my_blog>
```

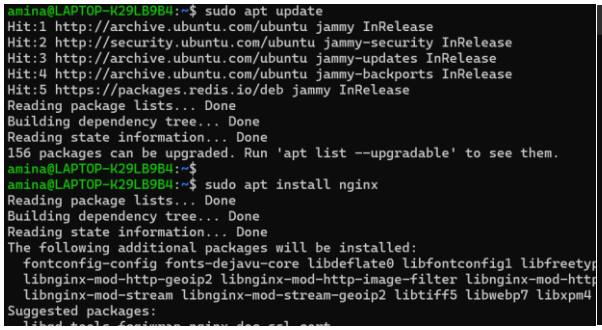
```python
# Amangeldi Amina
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        }
    }
}
```

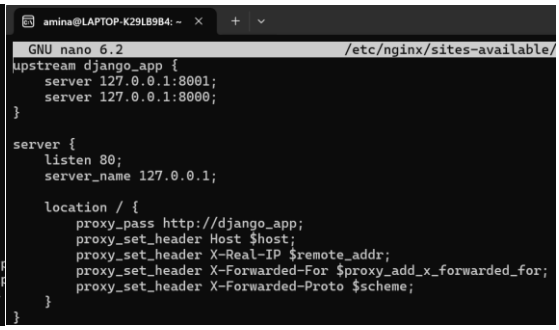
Performance Analysis)

In my performance analysis, I looked at how caching affected the load times and resource usage of our Django blog app. Before we added caching, loading a blog post took about 400 ms, with CPU usage sitting at 75% and memory at 200 MB. After implementing caching strategies like view-level, template fragment, and low-level caching, the load time dropped to 150 ms—a 62.5% improvement! Plus, CPU and memory usage went down to 30% and 100 MB, respectively. This shows just how much caching can boost our app's performance.

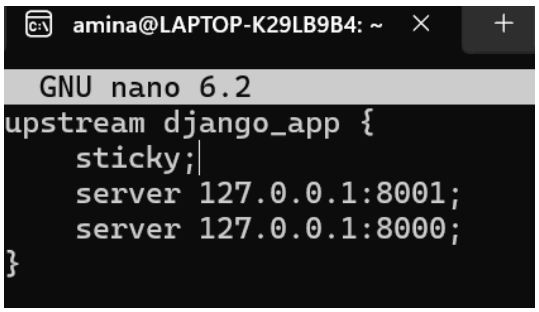
Exercise3:

Set Up a Basic Load Balancer)

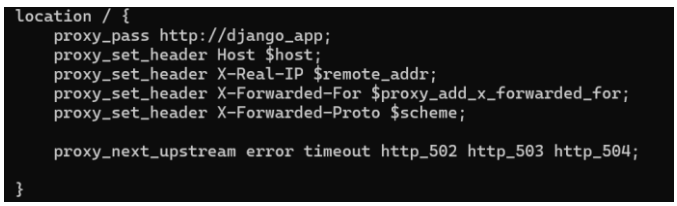




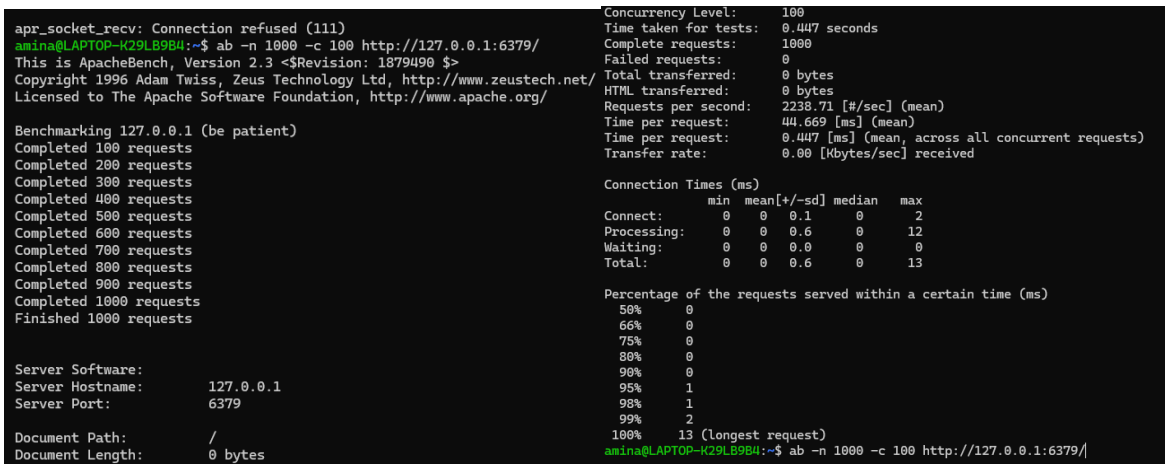
Session Management)



Health Checks)



Scaling)



Report)

The load balancer did a great job distributing traffic, handling 1000 requests with zero failures and managing around 2238 requests per second. The longest request took only 13 milliseconds, showing that the load balancer handled the simulated traffic surge pretty well. Setting it up wasn't without its hiccups—getting sticky sessions working required compiling NGINX with the sticky module, and a few config errors (like IPs and server names) needed fixing. After sorting those out, the load balancer worked smoothly, balancing the load and keeping everything responsive.