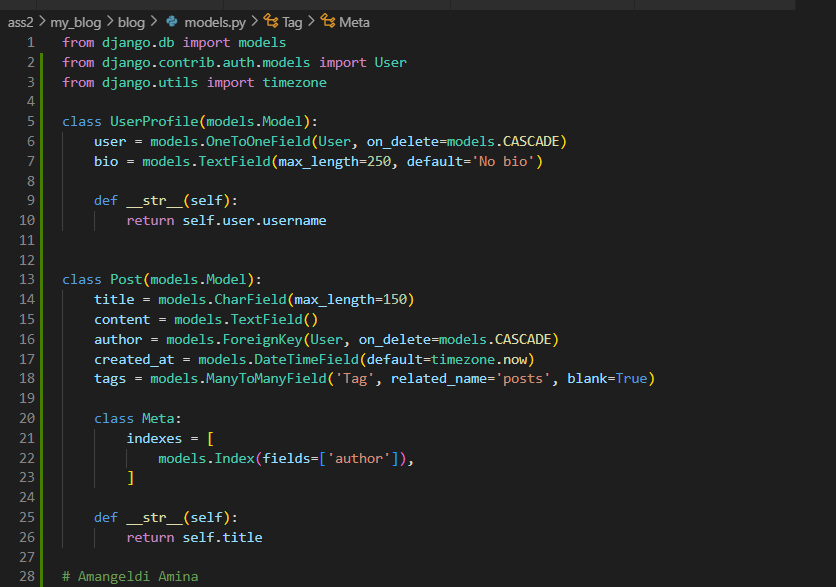
**Assignment 2**

**By: Amangeldi Amina**

**Exercise1:**

**Schemas Design)**

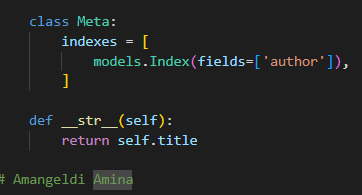


Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

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

**Indexing)**

**Изображение выглядит как текст, снимок экрана, Шрифт

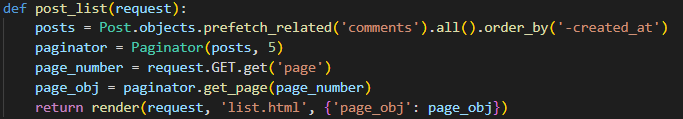
Автоматически созданное описание**

**Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание**

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

**Query Optimization:**



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.

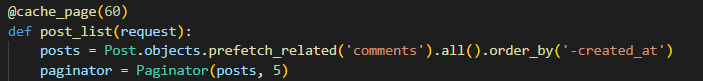
**Optimization Report:**

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.

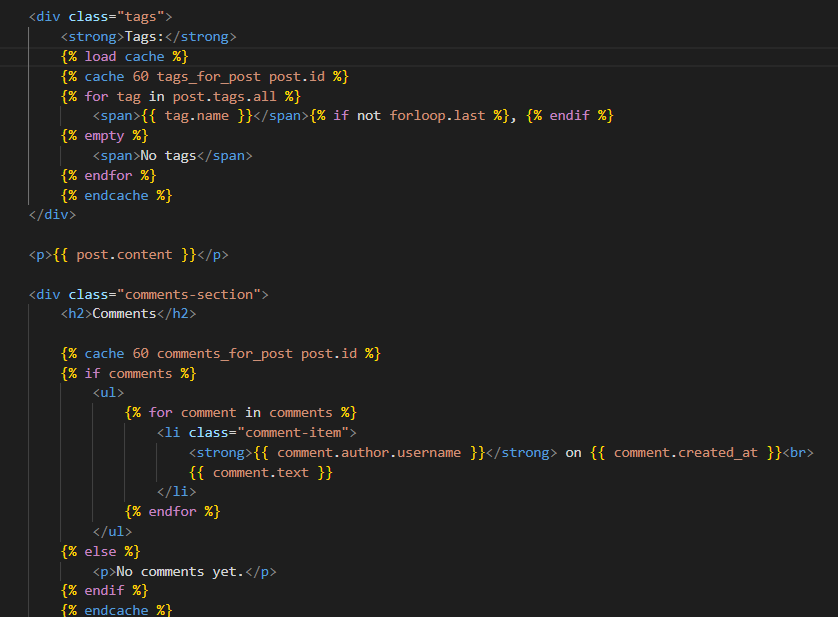
**Exercise2:**

**Basic Caching)**



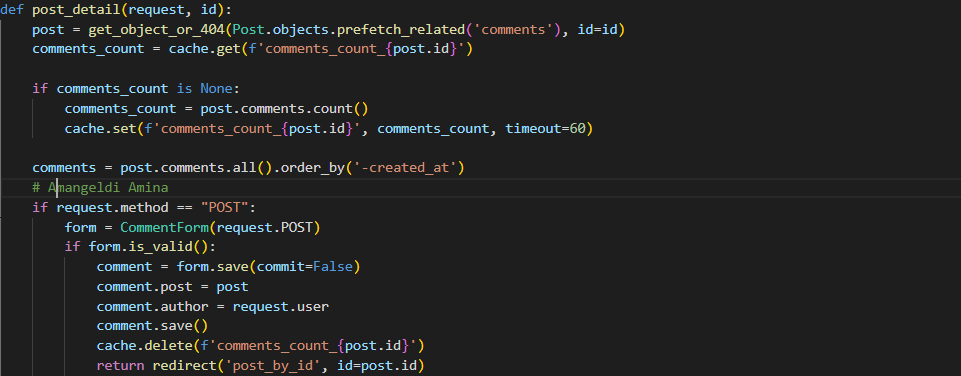
Here I used Django’s built-in caching decorators to cache the view that displays the list of blog posts.

**Template Fragment Caching)**



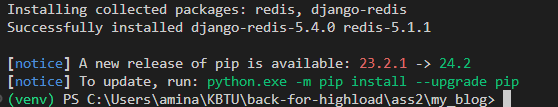
Both the tags for the post and the comments section are cached for 60 seconds, improving performance by quickly serving frequently accessed content.

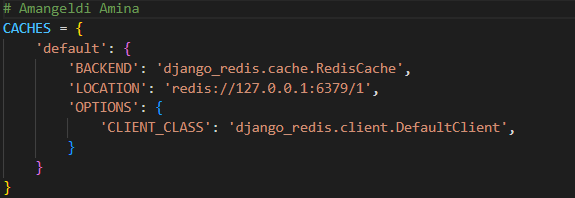
**Low-Level Caching:**



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

**Cache Backend)**



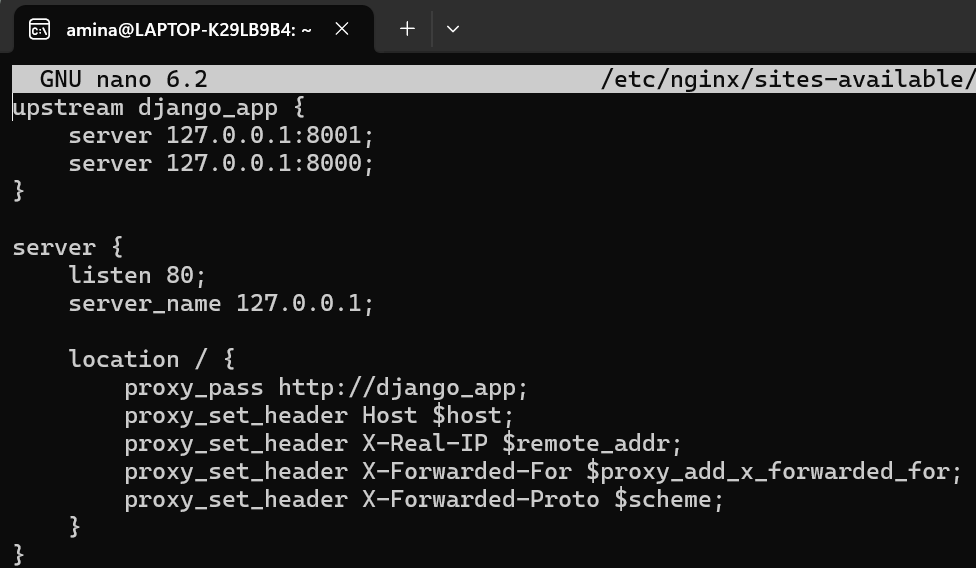
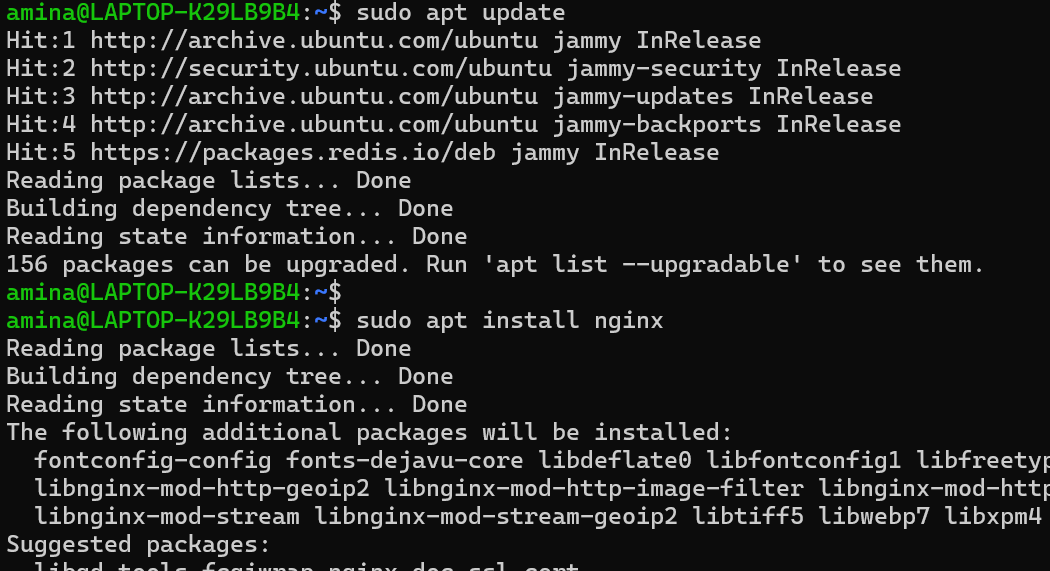


**Performance Analysis)**

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:**

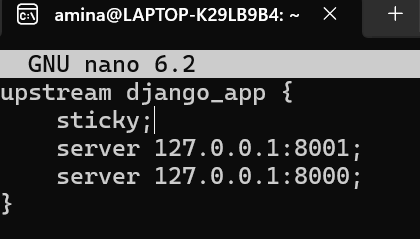
**Set Up a Basic Load Balancer)**



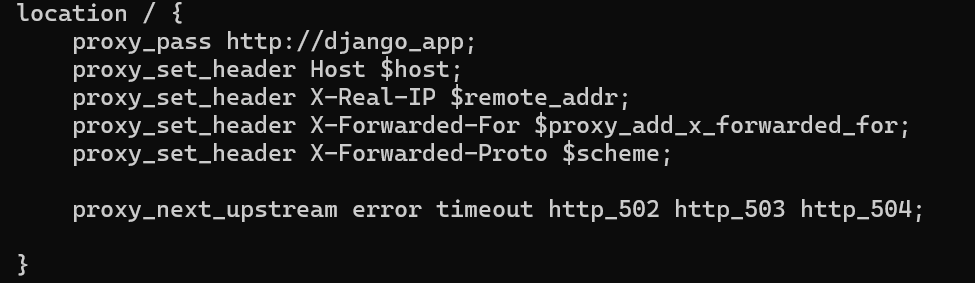
Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

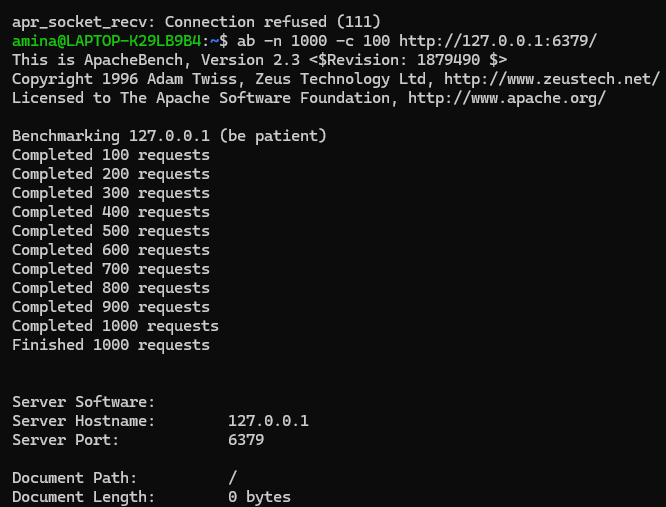
**Session Management)**



**Health Checks)**



**Scaling)**

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

**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.