Midterm project, Backend for high loaded environment
Sazanova Aruzhan

REPORT

1. Executive Summary [5 points]
Overview: The Soundwave is an online hobby market for musical community. It has accessories, musical instruments, equipment, merchandise, vinyl etc.
Goals:
- Users can purchase many items with one order.
- Site can hold minimum 20 requests concurrently and 2000 requests totally.
- Fast loading data
- Personalized endpoints.
Outcomes:
-All satisfy the goals.

2. Introduction [10 points]
Significant features:
Scalability: Handles increased traffic.
Reliability: Reduces downtime.
User Experience: Boosts satisfaction.
Cost Efficiency: Lowers operational costs.
Data Processing: Enables real-time insights.
Competitive Advantage: Responds quickly to demand.
Innovation Support: Allows stable feature development.
Global Reach: Consistent performance worldwide.
Examples:
Amazon: Uses load balancing to distribute traffic and caching to improve data retrieval speed.
eBay: Implements a microservices architecture to scale components independently.
Netflix: Utilizes a cloud-based infrastructure for dynamic scaling and resilience.
Facebook: Employs a combination of sharding and caching to manage large user data.
Alibaba: Handles high traffic with a robust distributed system and real-time monitoring.
Google: Uses Bigtable for scalable data storage and load distribution.
LinkedIn: Adopts a Kafka-based messaging system for real-time data processing.
Spotify: Implements microservices and uses Docker containers for efficient resource management.
Twitter: Utilizes Redis for caching and horizontal scaling to manage user requests.
Salesforce: Employs multi-tenancy architecture to efficiently serve multiple customers.

3. Project Objectives [10 points]
- Users can purchase many items with one order.
    - Create orderItem class which will be the part of Order class.
    - Make concurrent task which will count total price for the whole purchase.
- Site can hold minimum 20 requests concurrently and 2000 requests totally:
    - Implement nginx load balancing, add ip_hashing, and health check
    - Ensure everything works with Apache Benchmark.
    - Use gunicorn to run session concurrently.
- Fast loading data:

- Add indexing and use ORM queries that optimize database data retrieve.
- Implement caching backend and low level caching.
- Normalize database entities.

- Personalized endpoints.
  - Implement JWT DRF.
  - Add permission classes.
  - Set admin-only permissions.

4. System Architecture [25 points]

Category:
- Categories are needed to manage products.

Product:
- Products have category (FK), name, stock, price and description

User:
- Is a custom user referencing AbstractUser which has additional field phone_number.

OrderItem:
- Is an order of one Product (FK) and shows the user (FK) that created it, and number of this product(quantity) client wants to purchase.

Order:
- Contains multiple OrderItems, shows the total price, and linked to the user (FK).

Indexing was added to foreign keys to run faster while inner joins.

One-to-Many:
Categories contain many products, users can have many order items and orders.

Many-to-One:
Each product belongs to one category, each order item references one product and one user.

Many-to-Many:
Orders contain multiple order items, and users can belong to multiple groups and have multiple permissions.

The additional UML diagram is provided in my repository.

5. Backend Development [40 points]

5.1 Django Setup

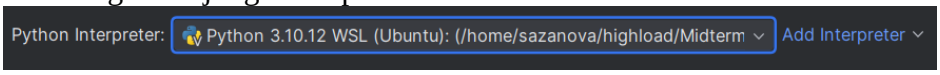
1. Created a virtual environment:

```
sazanova@DESKTOP-8GO62UR:~/highload/Midterm$ python -m venv myvenv
```

2. Activated it:

```
sazanova@DESKTOP-8GO62UR:~/highload/Midterm$ source myvenv/bin/activate
(myvenv) sazanova@DESKTOP-8GO62UR:~/highload/Midterm$
```

3. Configured django interpreter:



4. Installed Django and DRF:

$ pip install django

$ pip install djangorestframework

5. Created django project:

```
(myvenv) sazanova@DESKTOP-8GO62UR:~/highload/Midterm$ django-admin startproject soundwave
```

6. Created app ‘api’:

```
(myvenv) sazanova@DESKTOP-8G062UR:~/highload/Midterm/soundwave$ python manage.py startapp api
```

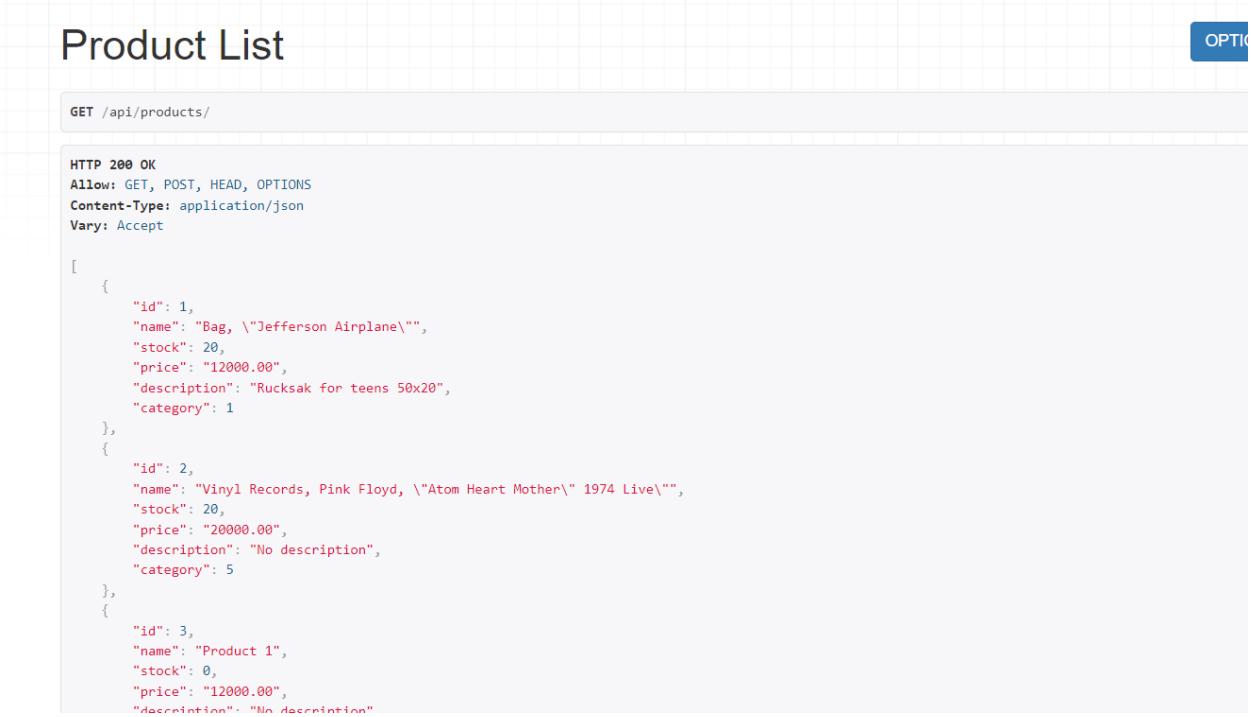
5.2 API Implementation

- Description of implemented API endpoints (e.g., products, orders, users) with examples of request and response formats.

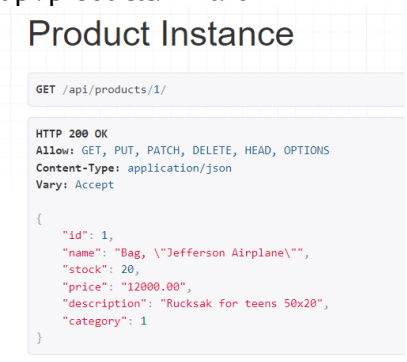
1. http://127.0.0.1:8000/api/products/:

‘GET’:

– api/products/



– api/products/<int:id>



– Api/products/?category=1

```
[
    {
        "id": 1,
        "name": "Bag, \"Jefferson Airplane\"",
        "stock": 20,
        "price": "12000.00",
        "description": "Rucksak for teens 50x20",
        "category": 1
    },
    {
        "id": 3,
        "name": "Product 1",
        "stock": 0,
        "price": "12000.00",
        "description": "No description",
        "category": 1
    },
    {
        "id": 7,
        "name": "Product 4",
        "stock": 0,
        "price": "10000.00",
        "description": "No description",
        "category": 1
    },
    {
        "id": 13,
        "name": "Product 10",
        "stock": 0,
        "price": "20000.00",
        "description": "No description",
        "category": 1
    },
```


– api/categories/

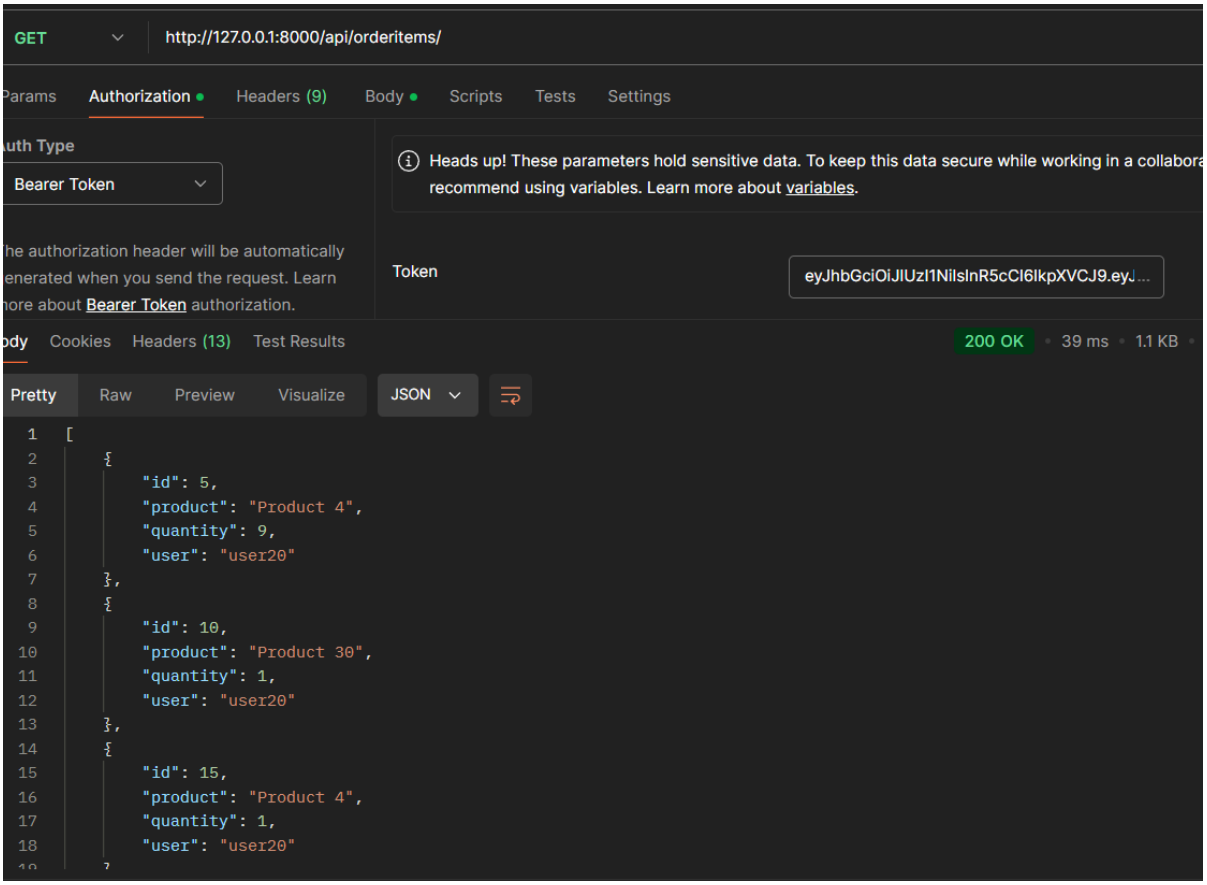

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "Merch"
    },
    {
        "id": 2,
        "name": "Musical Instruments"
    },
    {
        "id": 4,
        "name": "Musical Equipment"
    },
    {
        "id": 5,
        "name": "Vinyl Records"
    },
    {
        "id": 6,
        "name": "Disks"
    },
    {
        "id": 7,
        "name": "Accessories"
```


- api/token/

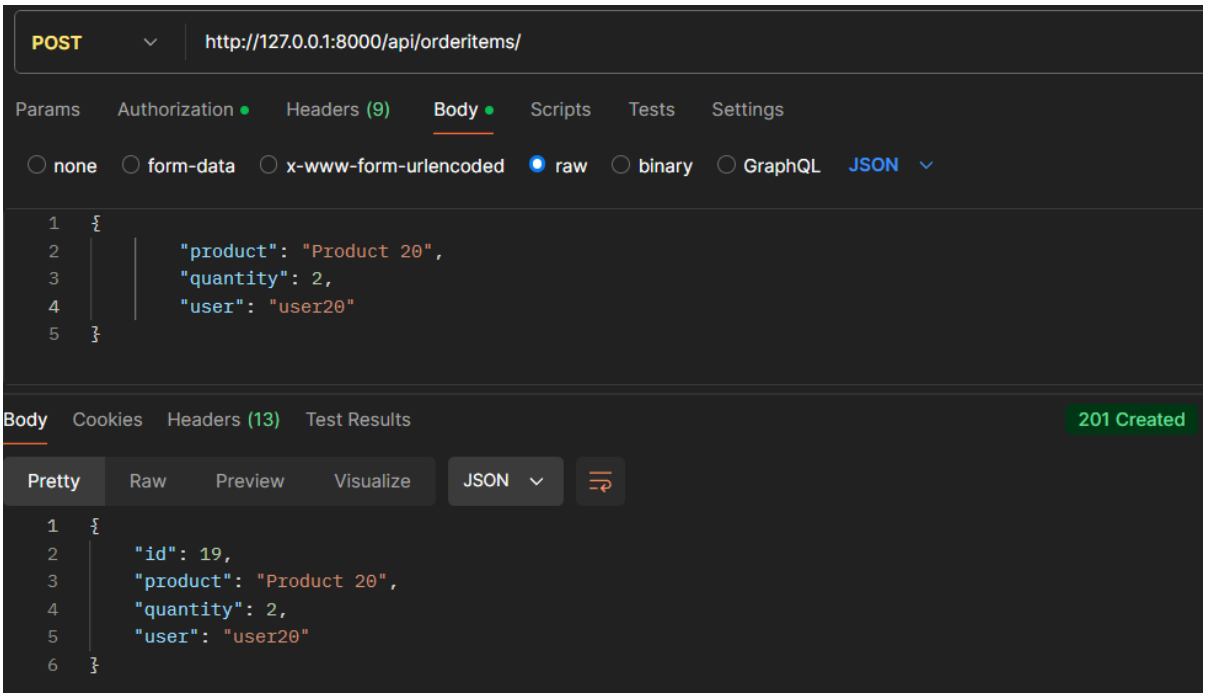
Post username, password; Get access, refresh tokens

```
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTcyOTUzNTYyNywiaWF0IjoxNzI5NDQ5MjI3LCJqdGkiOiJkZjE1MDNiZDJlNTY0ZmQ5YWRhYTdiM2
    "access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzI5NDQ5NTI3LCJpYXQiOjE3Mjk0NDkyMjcsImp0aSI6IjBiNmQ2ZDMxODFjZDRlYTA5ZThhMmM1YmI
}
```


- Api/orderitems/
- GET



-
- POST



- Api/token/refresh

- Api/orders/
- GET

- Api/products/?name= Product 1

-


```
[
    {
        "id": 3,
        "name": "Product 1",
        "stock": 0,
        "price": "12000.00",
        "description": "No description",
        "category": 1
    },
    {
        "id": 13,
        "name": "Product 10",
        "stock": 0,
        "price": "20000.00",
        "description": "No description",
        "category": 1
    },
    {
        "id": 14,
        "name": "Product 11",
        "stock": 0,
        "price": "20000.00",
        "description": "No description",
```

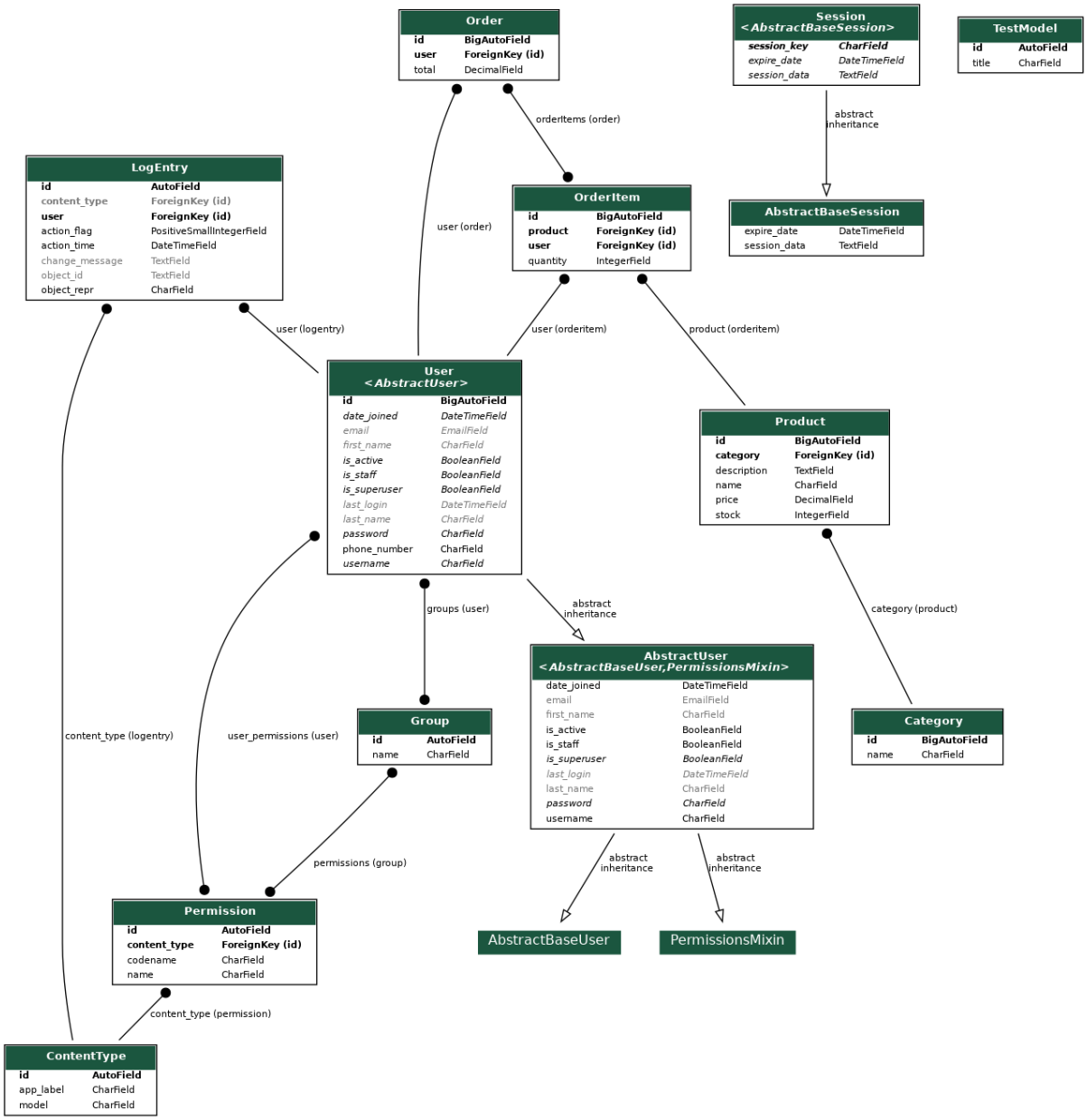

6. Database Design and Optimization

6.1 Schema Design

- Overview of the database schema, including entity-relationship diagrams.

```python
class Category(models.Model):
    name = models.CharField(max_length=200, db_index=True, unique=True)
    aruzhan
    def __str__(self):
        return self.name


aruzhan
class Product(models.Model):
    category = models.ForeignKey(Category, on_delete=models.CASCADE, db_index=True)
    name = models.CharField(max_length=200, db_index=True)
    stock = models.IntegerField(default=0)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    description = models.TextField(default="No description")
    aruzhan
    def __str__(self):
        return self.name

aruzhan
class User(AbstractUser):
    phone_number=models.CharField(max_length=15)
    aruzhan
    def __str__(self):
        return self.username


6 usages  aruzhan
class OrderItem(models.Model):
    product = models.ForeignKey(Product, on_delete=models.CASCADE, db_index=True)
    quantity = models.IntegerField(default=1)
    user = models.ForeignKey(User, on_delete=models.CASCADE, db_index=True)

    aruzhan
    def __str__(self):
        return f"{self.product.name} (x{self.quantity})"
aruzhan *
class Order(models.Model):
    orderItems = models.ManyToManyField(OrderItem, default= None)
    total = models.DecimalField(max_digits=10, decimal_places=2, default=0)
    user = models.ForeignKey(User, on_delete=models.CASCADE, db_index=True)
    aruzhan *
    def __str__(self):
        return f"{self.total} in total"
```

Here is the ERD(entity-relationship diagram) made by django-extensions + graphviz



6.2 Query Optimization

- Techniques used to optimize database queries, with performance benchmarks.

I used prefetch_related and select_related to ease data retrieving process:

```
queryset = Order.objects.filter(user=self.request.user) \
    .prefetch_related('orderItems__product') \
    .select_related('user')
```

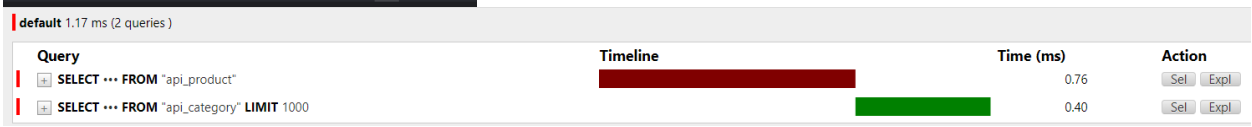
```
queryset = OrderItem.objects.filter(user=self.request.user).select_related('product', 'user')
```

- Filtering, sorting product list is available:

```python
def get_queryset(self):
    queryset = Product.objects.all()

    name = self.request.query_params.get('name', None)
    if name:
        queryset = queryset.filter(name__icontains=name)

    category_id = self.request.query_params.get('category', None)
    if category_id:
        queryset = queryset.filter(category_id=category_id)

    order_by_name = self.request.query_params.get('order_by_name', None)
    if order_by_name == 'asc':
        queryset = queryset.order_by('name')
    elif order_by_name == 'desc':
        queryset = queryset.order_by('-name')

    queryset = queryset.select_related('category')

    return queryset
```

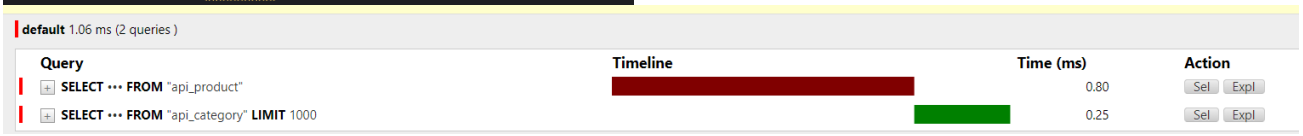
- Without select_related in products it retrieves from api_category table for 400ms.

```python
queryset = Product.objects.all()
```



- With select_related in products it retrieves from api_category for 250ms.

```python
queryset = Product.objects.all().select_related('category')
```



7. Caching Strategies

7.1 Caching has been applied to:

- Product listings (ProductViewSet) > cache-aside strategy
- User-specific Orders (OrderViewSet) > cache-aside strategy
- User-specific Order Items (OrderItemViewSet) > cache-aside strategy
- User sessions (via Django's session caching)
- Example of cache-aside:

```python
def list(self, request, *args, **kwargs):
    cached_products = cache.get('products_list')
    if cached_products:
        return Response(cached_products)

    response = super().list(request, *args, **kwargs)
    cache.set('products_list', response.data, timeout=60 * 15)
    return response
```

7.2 Caching Policies:

- I used Django Redis for backend caching:

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

- Most data is cached for 15 minutes according to the data flow in my API.

```
cache.set('products_list', response.data, timeout=60 * 15)
```

7.3 Cache Invalidation: delete caches after updating, creating or destroying data (e.g Order).

```
new *
def create(self, request, *args, **kwargs):
    response = super().create(request, *args: *args, **kwargs)
    cache.delete(f"user_orders_{self.request.user.id}")
    return response

new *
def update(self, request, *args, **kwargs):
    response = super().update(request, *args: *args, **kwargs)
    cache.delete(f"user_orders_{self.request.user.id}")
    return response

new *
def destroy(self, request, *args, **kwargs):
    response = super().destroy(request, *args: *args, **kwargs)
    cache.delete(f"user_orders_{self.request.user.id}")
    return response
```

- I have used user-specific Order Items caching to delete caches that differ according to the user session:

```
cache.delete(f"user_orders_{self.request.user.id}")
```

7.3 User sessions are stored in the cache backend with the following settings:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
SESSION_CACHE_ALIAS = 'default'
```

The session cache is managed automatically by Django.

7.4 Example:

- I requested a list of products: in ‘cache’ section of Django Debug Toolbar we can see it sets a cache for 900 sec and returns it to us.

Time (ms)	Type	Arguments	Keyword arguments	Backend
13.7512	get	('products_list',)	{}	<django_redis.cache.RedisCache object at 0x7f8cde65d390>
1.7851	set	('products_list', [{'id': 1, 'name': 'Bag, "Jefferson Airplane"', 'stock': 20, 'price': '12000.00', 'description': 'Rucksak for teens 50x20', 'category': 1}, {'id': 2, 'name': 'Vinyl Records, Pink Floyd, "Atom Heart Mother" 1974 Live"', 'stock': 20, 'price': '20000.00', 'description': 'No description', 'category': 5}])	{'timeout': 900}	<django_redis.cache.RedisCache object at 0x7f8cde65d390>

- But when I create new item, it invalidates(deletes) the cache to refresh it:

Time (ms)	Type	Arguments	Keyword arguments	Backend
3.3678	delete	('products_list',)	{}	<django_redis.cache.RedisCache object at 0x7fcb073393c0>

- The same happens when I ‘PUT’ changes on name of the same product:

Time (ms)	Type	Arguments	Keyword arguments	Backend
0.4229	delete	('products_list',)	{}	<django_redis.cache.RedisCache object at 0x7f6c93d15390>

8. Load Balancing Techniques

For managing load balancing I’ve used nginx and gunicorn tools. Also implemented health check, ip_hashing.

8.1 Setup

- Installed gunicorn:

```
(myvenv) sazanova@DESKTOP-8GO62UR:~/highload/Midterm/soundwave$ pip install gunicorn
```

- Created nginx.conf file:

```
(myvenv) sazanova@DESKTOP-8GO62UR:~/highload/Midterm/soundwave$ sudo nano nginx.conf
```

- Created ‘staticfiles’ file, added directory of static files into settings.py, urls.py, and collected all static files:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')

STATIC_URL = 'static/'
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('api.urls')),
    path('__debug__/', include('debug_toolbar.urls')),
    path('health/', include('health_check.urls')),
]+ static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

```
(myvenv) sazanova@DESKTOP-8GO62UR:~/highload/Midterm/soundwave$ python manage.py collectstatic
```

8.2 Configuration of nginx

- Used 2 basic and 1 backup servers for this project, set the limit of fails and fail timeouts.
- Used IP hashing to maintain session consistency:

```
http{
    upstream django_servers {
        ip_hash;
        server 127.0.0.1:8000 max_fails=3 fail_timeout=30s;
        server 127.0.0.1:8001 max_fails=3 fail_timeout=30s;
        server 127.0.0.1:8002 backup;
    }
```

- Each server block contains:
  - Location of static files:

```
location /static/ {
    alias /home/sazanova/highload/Midterm/soundwave/staticfiles;
    autoindex on;
}
```

– Requests distributor:

```
location / {
    proxy_pass http://127.0.0.1:8000;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    }
```

– Health check location:

```
location /health {
    proxy_pass http://django_servers/health;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    }
```

9. Distributed Systems and Data Consistency

- Discussion on the implementation of distributed systems concepts and how data consistency was maintained.

Installed RabbitMQ and django-celery-results;

Implemented celery.py:

```
import os

from celery import Celery

os.environ.setdefault( key: 'DJANGO_SETTINGS_MODULE', value: 'soundwave.settings')

app = Celery('soundwave')
app.config_from_object( obj: 'django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()
```

```
CELERY_BROKER_URL = 'amqp://localhost'
CELERY_RESULT_BACKEND = 'django-db'
CELERY_ACCEPT_CONTENT = ['json']
CELERY_TASK_SERIALIZER = 'json'
```

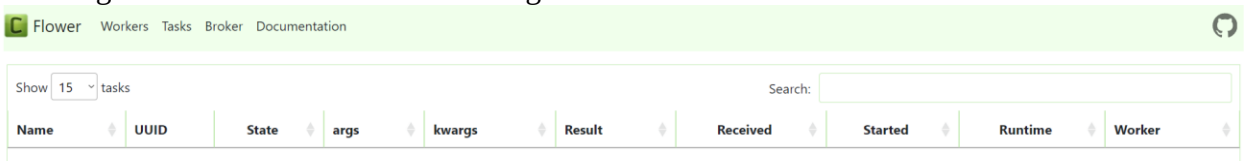
and celery setup on settings.py

Added concurrent tasks to tasks.py:

```python
from celery import shared_task

from api.models import Order


@shared_task
def process_order(order_id):
    try:
        order = Order.objects.get(id=order_id)

        charge_payment(order)
        update_stock(order)

        print(f"Order {order.id} processed successfully.")
    except Order.DoesNotExist:
        print(f"Order with id {order_id} does not exist.")
    except Exception as e:
        print(f"Error processing order {order_id}: {e}")
```

```python
1 usage
def charge_payment(order):
    total_amount = 0

    for order_item in order.orderItems.all():
        total_amount += order_item.product.price * order_item.quantity

    order.total = total_amount
    order.save()

    print(f"Total amount for order {order.id} calculated as {order.total}.")


1 usage
def update_stock(order):
    for order_item in order.orderItems.all():
        product = order_item.product
        product.stock -= order_item.quantity
        product.save()
    print(f"Updating stock for order {order.id}.")
```

Challenges faced: Vizualization of messages on flower:



Mitigation: used django celery results instead.

Dataconsistency also was provided with caching strategies that were mentioned above.

- **10. Testing and Quality Assurance**
- Overview of testing strategies employed, including unit tests, integration tests, and results..

```python
from selenium import webdriver
from django.contrib.staticfiles.testing import StaticLiveServerTestCase

class OrderEndToEndTests(StaticLiveServerTestCase):
    def setUp(self):
        self.browser = webdriver.Chrome()

    def tearDown(self):
        self.browser.quit()

    def test_user_can_place_order(self):
        self.browser.get(self.live_server_url + '/')
        self.browser.find_element_by_name('product').send_keys('Product Name')
        self.browser.find_element_by_name('quantity').send_keys('2')
        self.browser.find_element_by_name('submit').click()
        self.assertIn( member: 'Order placed successfully', self.browser.page_source)
```

```python
from django.test import TestCase
from django.urls import reverse
from api.models import Product

class OrderFunctionalTests(TestCase):
    def setUp(self):
        self.product = Product.objects.create(price=30.0)

    def test_order_page_loads(self):
        response = self.client.get(reverse('order-create'))
        self.assertEqual(response.status_code,  second: 200)
        self.assertTemplateUsed(response,  template_name: 'order_form.html')

    def test_order_submission(self):
        response = self.client.post(reverse('order-create'), {'items': [{'product_id': self.product.id, 'quantity': 2}]})
        self.assertEqual(response.status_code,  second: 302)
        self.assertRedirects(response, reverse('order-success'))
```

```python
import pytest
from django.urls import reverse
from soundwave.api.models import Order, Product

@pytest.mark.django_db
def test_create_order():
    product = Product.objects.create(price=20.0)
    response = client.post(reverse('order-create'), {'items': [{'product_id': product.id, 'quantity': 3}]})
    assert response.status_code == 201
    assert Order.objects.count() == 1
    assert Order.objects.first().total() == 60.0
```

```python
import pytest
from soundwave.api.models import *

@pytest.mark.django_db
def test_order_total_calculation():
    product1 = Product.objects.create(price=10.0)
    product2 = Product.objects.create(price=5.0)
    order = Order()
    order.items.add(OrderItem(product=product1, quantity=2))
    order.items.add(OrderItem(product=product2, quantity=1))
    assert order.total() == 25.0
```

11. Monitoring and Maintenance

- Description of monitoring tools and practices established to ensure system health and performance. Tools to check if everything is working properly:
- To perform health-check I installed *django-health-check*. When I go to "domain/health" I get:

System status

Service	Status	Time Taken
Cache backend: default	working	0.0044 seconds
DatabaseBackend	working	0.0207 seconds

- To simulate high traffic, I used Apache Benchmark tool and set it as:
ab -n 10000 -c 100 http://localhost:8000/api (total 10 000 requests and 100 concurrent)

```
Concurrency Level:      100
Time taken for tests:   159.670 seconds
Complete requests:      10000
Failed requests:        0
Non-2xx responses:      10000
Total transferred:      2830000 bytes
HTML transferred:       0 bytes
Requests per second:    62.63 [#/sec] (mean)
Time per request:       1596.699 [ms] (mean)
Time per request:       15.967 [ms] (mean, across all concurrent requests)
Transfer rate:          17.31 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.6      0      10
Processing:    14 1589 1231.1   1470   13856
Waiting:       14 1589 1231.1   1470   13856
Total:         24 1589 1231.0   1470   13856

Percentage of the requests served within a certain time (ms)
  50%   1470
  66%   1488
  75%   1500
  80%   1510
  90%   1530
  95%   1550
  98%   1576
  99%  13732
 100%  13856 (longest request)
```

By the received data with 100 concurrent requests and 10000 requests the web site processes 1 request in 16ms and longest request took almost 14 sec (when –n 100 –c 10 it tooks about 0.2 sec for the longest request). But finally, there are no failed requests, and approximate speed is 62requests/sec.

12. Challenges and Solutions

- RabbitMQ: Installation took a lot of time cause there were some damaged packages when installing.
-

13. Conclusion [10 points]

This project successfully developed a **high-load e-commerce API** using Django, addressing key challenges associated with performance, scalability, and data consistency. The architecture incorporates advanced techniques such as **Redis caching**, **NGINX load balancing**, and **asynchronous task processing** using RabbitMQ, which collectively enhance the system's ability to handle significant traffic while maintaining low response times.

Key achievements include optimizing database queries with `select_related` and `prefetch_related`, reducing database load through caching, and ensuring user-specific data consistency even under high traffic. The system was tested to handle 10,000 requests with 100 concurrent users, demonstrating a robust performance of **62 requests per second** without failed requests.

Future improvements could focus on **further optimizing the caching strategy**, implementing **horizontal scaling** for the database, and improving **monitoring tools** for real-time traffic analysis. Overall, the project achieved its objectives of building a scalable, high-performance backend for a high-load e-commerce environment.

15. Appendices [0 points, write if necessary]

- UML is on https://github.com/xydownik/highload_Ubuntu/tree/master/Midterm