KAZAKH KBTU BRITISH
TECHNICAL
UNIVERSITY

Assignment 3

Student: Islam Aip

Course: Backend for high-loaded environment

Date:

Almaty, 2024

Introduction

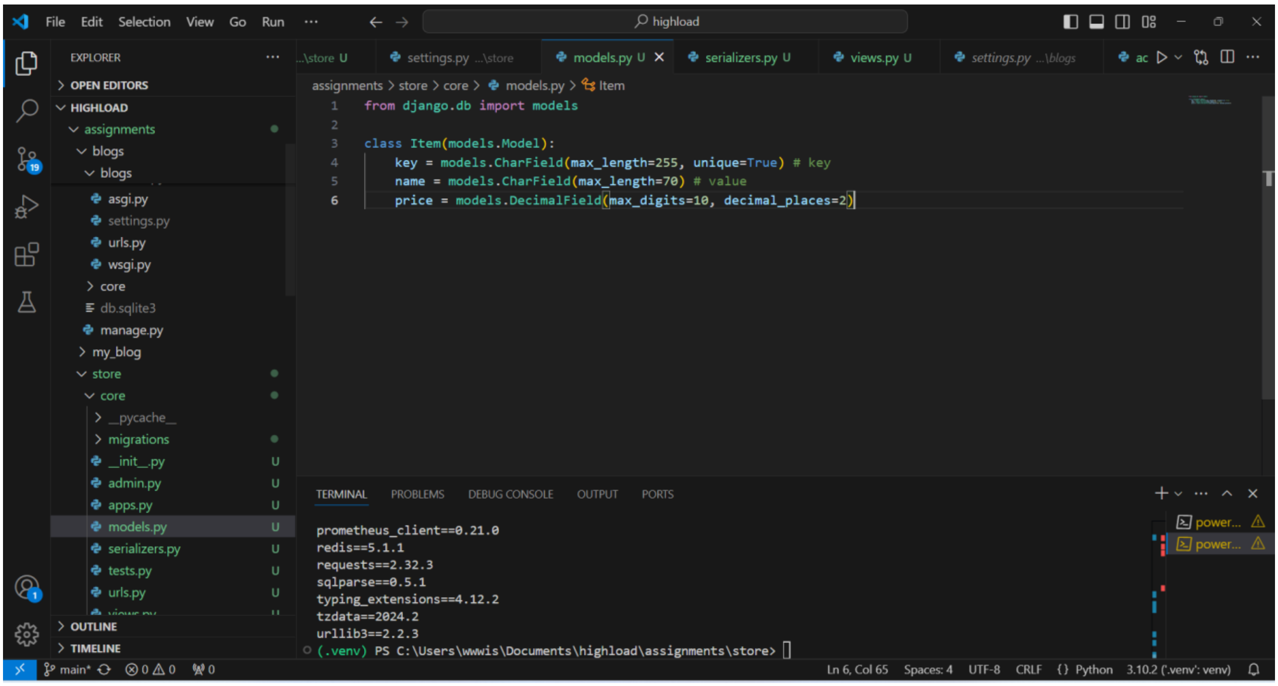
The topics of exercises cover essential concepts in building and managing distributed and scalable backend systems, including key-value stores, data consistency models, load-balancing, and log analysis. These concepts address critical challenges in modern software development, such as handling large-scale data, ensuring reliable and consistent information across systems, and maintaining high availability and performance.

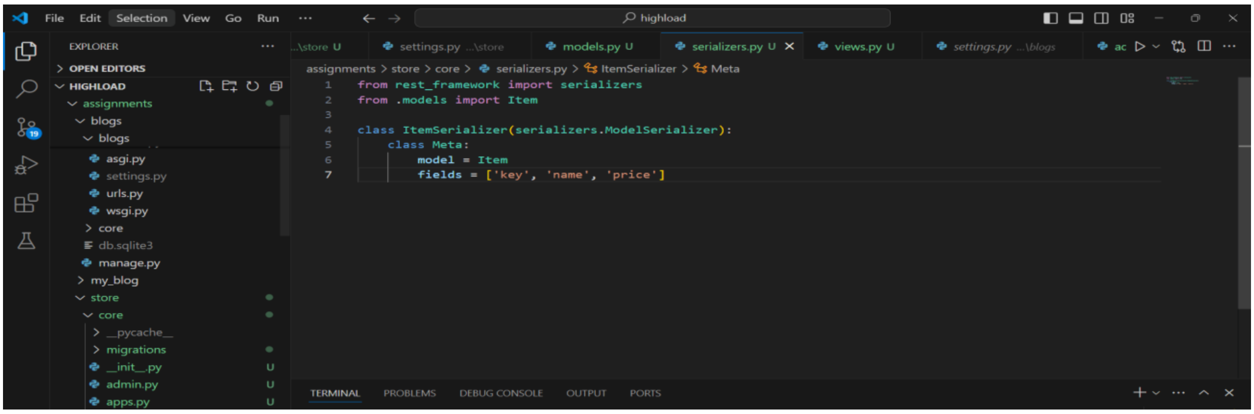
Distributed Systems and Data Consistency

Distributed systems are networks of independent computers that work together to perform complex tasks. They split work across multiple nodes, or servers, which communicate to provide best service. This overall improves reliability, enhances scalability and performance.

Exercise 1: Implement a Distributed Key-Value Store

First, we need to create a model to store keys and values in our app and serializer to handle data validation.





Now let’s create views to handle PUT and GET requests for storing and retrieving values.

```python
class KeyValueViewSet(viewsets.ViewSet):
    peers = ["http://127.0.0.1:8001", "http://127.0.0.1:8002"]

    def list(self, request):
        items = Item.objects.all()
        serializer = ItemSerializer(items, many=True)
        return Response(serializer.data)

    def retrieve(self, request, pk=None):
        key = pk
        responses = []

        for peer in self.peers:
            try:
                response = requests.get(f"{peer}/store/{key}/")
                if response.status_code == 200:
                    responses.append(response.json()['name'])
                else:
                    print(f"No 200 OK response from {peer}")
            except requests.exceptions.RequestException as e:
                print(f"Failed to connect to {peer}: {e}")

```

```python
        if len(responses) >= 2:  # Quorum condition
            most_common_value = max(set(responses), key=responses.count)
            return Response({"key": key, "value": most_common_value})

        return Response({"error": "Failed quorum read"}, status=status.HTTP_404_NOT_FOUND)

    def create(self, request):
        responses = []

        for peer in self.peers:
            try:
                response = requests.put(f"{peer}/store/", data=request.data, timeout=2)
                if response.status_code == 201:
                    responses.append(response)
            except requests.exceptions.RequestException as e:
                print(f"Failed to write to {peer}: {e}")

        if len(responses) >= 2: #Quorum condition
            serializer = ItemSerializer(data=request.data)
            if serializer.is_valid():
```


```python
            except requests.exceptions.RequestException as e:
                print(f"Failed to write to {peer}: {e}")

        if len(responses) >= 2: #Quorum condition
            serializer = ItemSerializer(data=request.data)
            if serializer.is_valid():
```


```python
                serializer.save()
                return Response(serializer.data, status=status.HTTP_201_CREATED)

        return Response({"error": "Failed quorum write"}, status=status.HTTP_500_INTERNAL_SERVER_ERROR)

```

Here we run two Django instances ‘http://127.0.0.1:8001’ and ‘http://127.0.0.1:8002’. Then we check responses of both instances. We use Django’s request library to send API requests between running instances. When added we send requests to all nodes, but only wait for 2 nodes to succeed and for reads, we query all nodes and return if at least 2 nodes agree on data.

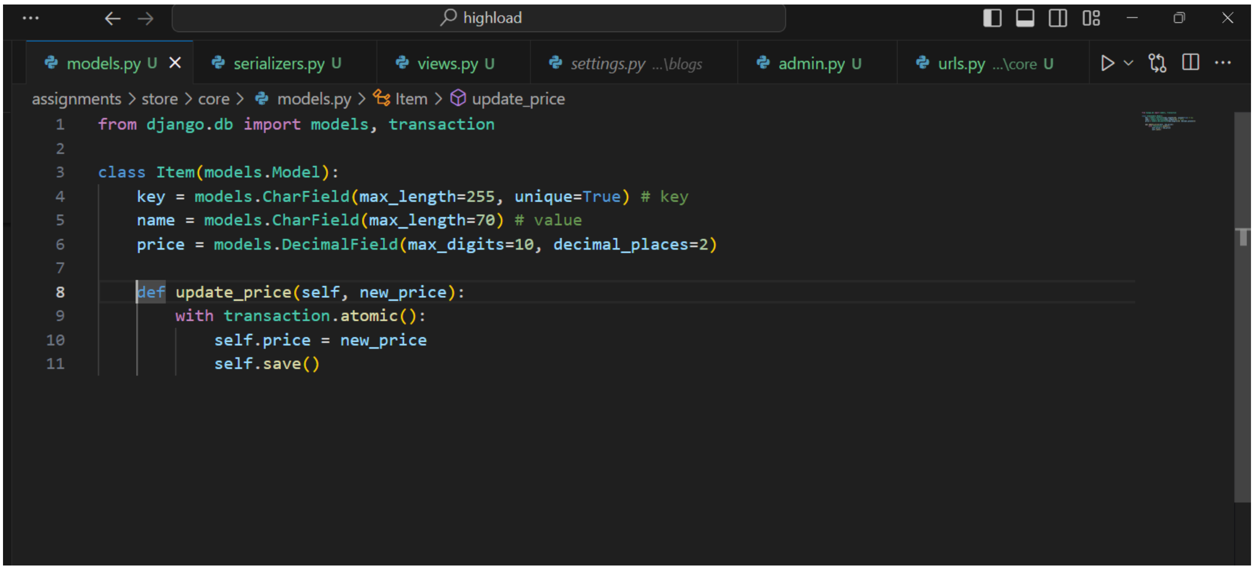
Exercise 2: Analyze Consistency Models

In distributed systems, consistency models describe the guarantees a system provides regarding the visibility of updates across multiple nodes. These models influence how data is synchronized across distributed components, balancing between performance, availability, and accuracy. The choice of consistency model largely depends on the application requirements.

1. Strong Consistency

Strong consistency ensures that once an update is made to a data item, all nodes see this update immediately. In other words, when a read operation occurs after a write, it will always return the most recent value, regardless of the node from which it's accessed. This model provides a system where all clients see updates instantly, as if they're interacting with a single data storage.

In Django, we can achieve this by using centralized database with atomic transactions.



```python
from django.db import models, transaction

class Item(models.Model):
    key = models.CharField(max_length=255, unique=True) # key
    name = models.CharField(max_length=70) # value
    price = models.DecimalField(max_digits=10, decimal_places=2)

    def update_price(self, new_price):
        with transaction.atomic():
            self.price = new_price
            self.save()
```

- This atomic transaction ensures that once the price updated, it is immediately available to all clients.

Strong consistency may result in high latency, as all nodes need to confirm the update.

2. Eventual Consistency

Eventual consistency is a model where updates are made asynchronously. As a result, when data is updated, it may not be immediately visible across all nodes, but given time without further updates, all nodes will eventually come to the same state and clients will see the same data. This model prioritizes availability and low latency.

```python
class Item(models.Model):
    def update_price(self, new_price):
            self.price = new_price
            self.save()

    def update_price_async(self, new_price):
        self.price = new_price
        self.save()

        propagate_price_update.delay(self.id, new_price)

@shared_task
def propagate_price_update(product_id, new_price):
    product = Item.objects.get(id=product_id)
    product.price = new_price
    product.save()
    cache.set(f'product_price_{product_id}', new_price)  # Cache the new price
```

There may be a slight delay in data visibility across nodes, leading to temporary inconsistencies.

3. Causal Consistency

Causal consistency provides guarantees based on the causal relationships between events. If an operation (for example A) causes or precedes another operation (for example B), then every node will observe A before B. Causal consistency allows nodes to see causally related updates in the correct order but doesn't require strict ordering for unrelated events.

```python

    def update_price_async(self, new_price):
        self.price = new_price
        self.save()

        propagate_price_update.delay(self.id, new_price)

    def update_price_causally(self, new_price, timestamp):
        if timestamp > self.last_updated:
            self.price = new_price
            self.last_updated = timestamp
            self.save()

@shared_task
def propagate_price_update(product_id, new_price):
    product = Item.objects.get(id=product_id)
    product.price = new_price
    product.save()
    cache.set(f'product_price_{product_id}', new_price)  # Cache the new price
```

Implementing causal consistency can add complexity, especially in distributed systems where operations may arrive out of order.

Scaling Backend Systems

Overview: Discuss the need for scaling backend systems.

- **Exercises:**
  - Summarize the key points from each exercise.

- **Findings:** Analyze the effectiveness of the scaling methods implemented.

Exercise 3: Load Balancing

To distribute incoming traffic and increase fault tolerance, we can use a load balancer (e.g., Nginx) to manage requests across multiple Django instances.

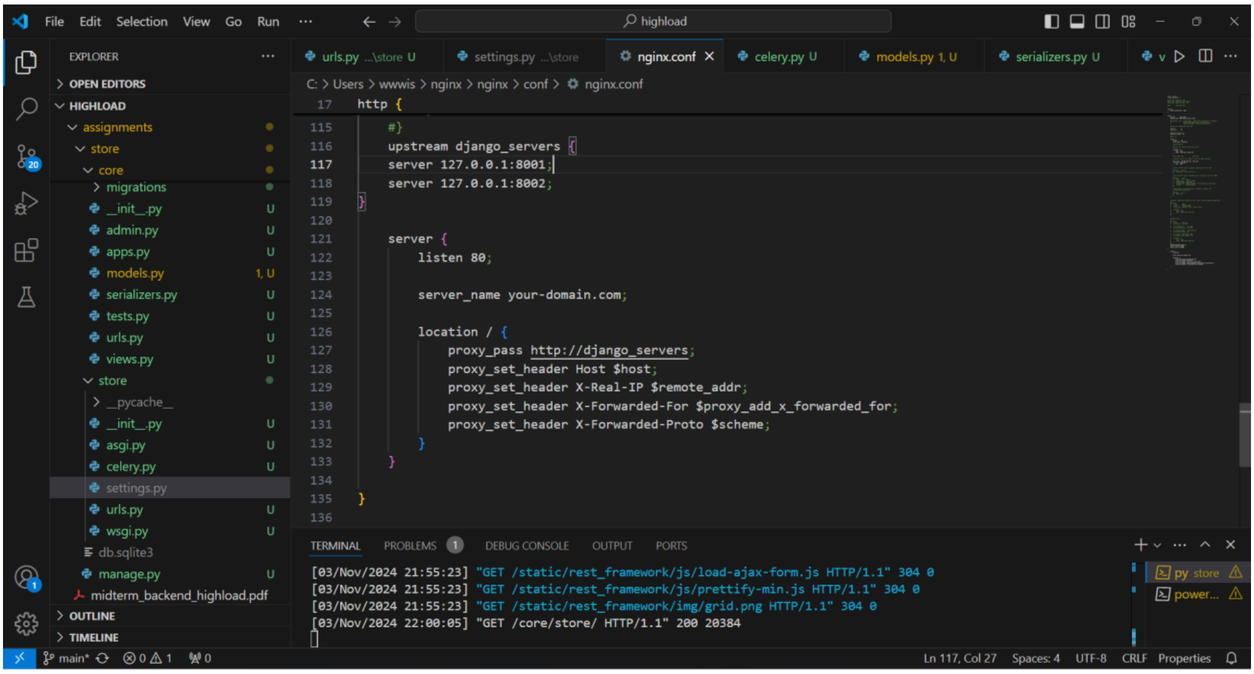
Let’s run our app in different ports to simulate different instances.

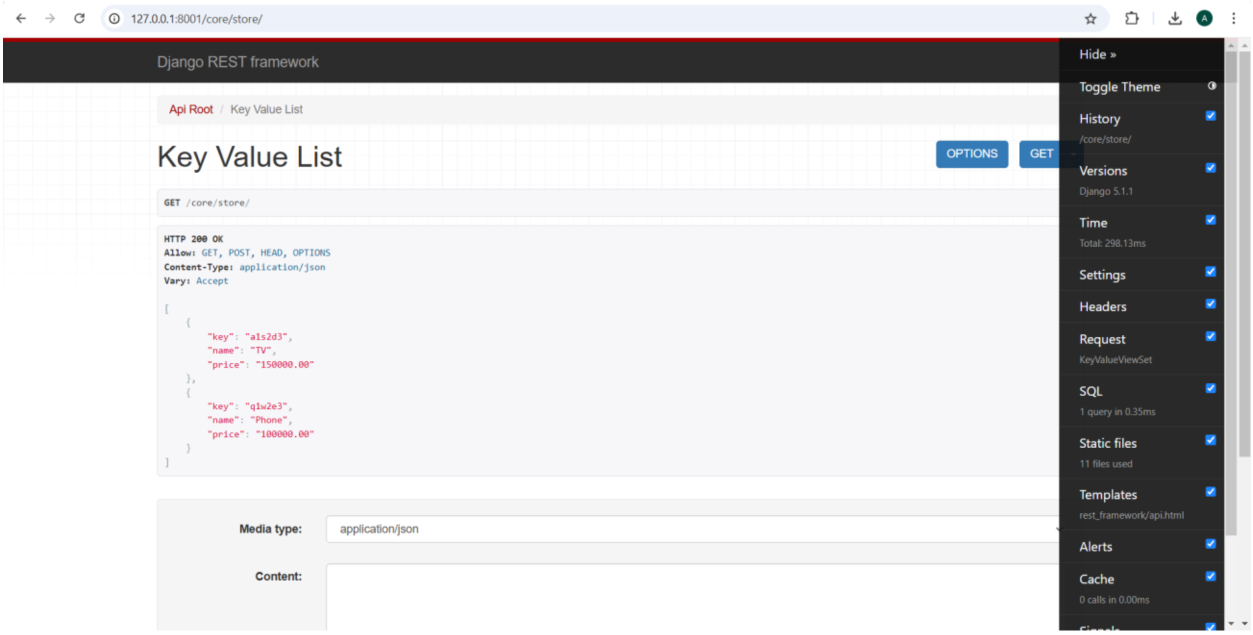
```
Django version 5.1.1, using settings 'store.settings'
Starting development server at http://127.0.0.1:8001/
Quit the server with CTRL-BREAK.
```

```
Django version 5.1.1, using settings 'store.settings'
Starting development server at http://127.0.0.1:8002/
Quit the server with CTRL-BREAK.
```

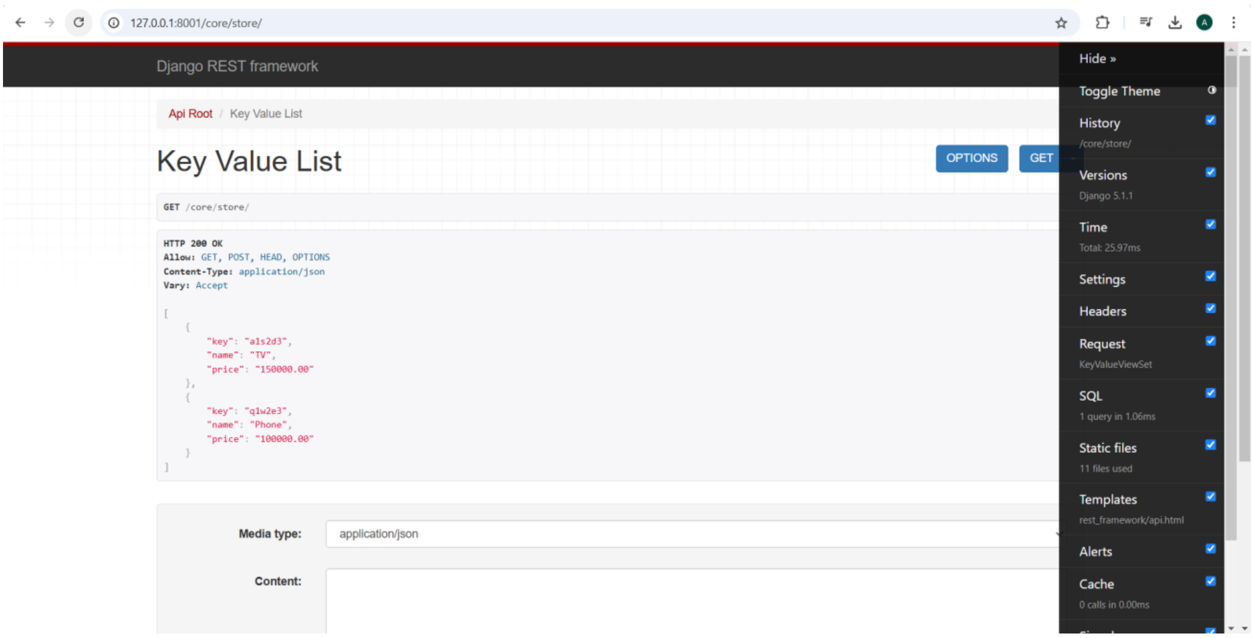
Now let’s configure Nginx.

We can open the Nginx configuration file and modify it to create upstream block and then configure the server to use this upstream for requests.





Before adding Load Balancer



After adding Load Balancer

Exercise 4: Database Scaling

For database scaling, we can create a read replica in PostgreSQL, which allows read operations to be distributed across the primary and replica, improving read performance.

First, we need to set up replica database for our app. Let's create new user and database.

```
SQL Shell (psql)
--------------------------------------------------
 C:/Program Files/PostgreSQL/14/data/postgresql.conf
(1 row)


store_db=# CREATE ROLE replica_user WITH REPLICATION PASSWORD 'replica_password' LOGIN;
CREATE ROLE
store_db=# select * users;
ОШИБКА:  ошибка синтаксиса (примерное положение: "users")
LINE 1: select * users;
                 ^
store_db=# \du;
                                   List of roles
  Role name   |                         Attributes                         | Member of
--------------+------------------------------------------------------------+-----------
 postgres     | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 replica_user | Replication                                                | {}


store_db=# SET role replica_user
store_db-# ;
SET
store_db=> SELECT current_user, session_user;
 current_user | session_user
--------------+--------------
 replica_user | postgres
(1 row)


store_db=> |
```

```
postgres=# SELECT current_user, session_user;
 current_user | session_user
--------------+--------------
 postgres     | postgres
(1 row)


postgres=# set role replication_user;
ОШИБКА:  роль "replication_user" не существует
postgres=# set role replica_user;
SET
postgres=> create database replica_store_db;
CREATE DATABASE
postgres=> |
```

Now let's modify our Django app and specify our primary and replica databases.

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'or_store_db',
        'USER': 'postgres',
        'PASSWORD': '',
        'HOST': 'localhost',
        'PORT': '5432'
    },
    'replica': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'replica_store_db',
        'USER': 'replica_user',
        'PASSWORD': '',
        'HOST': 'localhost',
        'PORT': '5432',
    },
}
```

We can create db_router.py to direct read operations to the replica database:

```python
class PrimaryReplicaRouter:
    def db_for_read(self, model, **hints):
        return 'replica'

    def db_for_write(self, model, **hints):
        return 'default'

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        return db == 'default'
```

```
(.venv) PS C:\Users\wwwis\Documents\highload\assignments\store> py manage.py migrate --database=replica
```

Monitoring and Observability

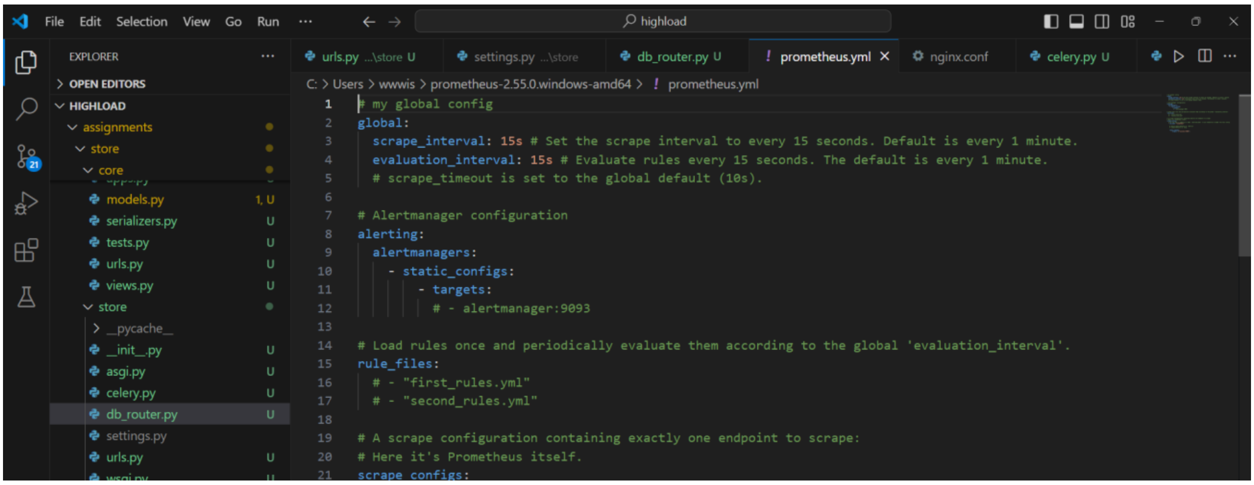
Monitoring is critical in modern software systems as it provides real-time insights into the health, performance, and reliability of applications. For Django applications, monitoring helps ensure the application's availability and responsiveness, providing early warnings for issues such as high latency, errors, or unexpected traffic spikes.

By using tools like Prometheus and Grafana, you can track key performance indicators and identify trends over time. For instance, monitoring response times and error rates allows developers to quickly detect performance bottlenecks and failures, enabling proactive response before they impact users. Furthermore, tracking resource usage (e.g., CPU, memory) is crucial for optimizing infrastructure costs and preventing downtime due to resource exhaustion.

With Grafana's dashboards, developers and DevOps teams can visualize these metrics clearly and set up alerts for critical thresholds. This not only facilitates troubleshooting but also guides data-driven improvements. For Django applications, monitoring thus plays an essential role in maintaining a high-quality, user-centered service, minimizing downtime, and optimizing application performance and resource usage.

Exercise 5: Integrate Monitoring Tools

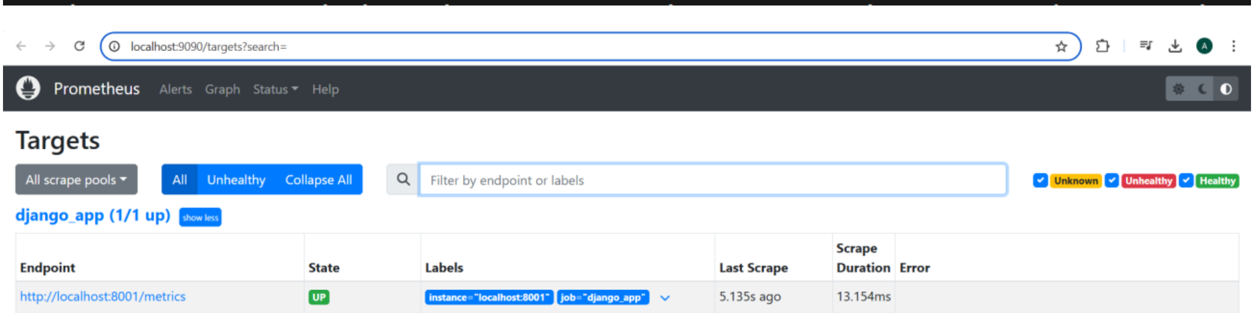
First we need to download and extract and navigate to the folder. After we can edit configuration file.



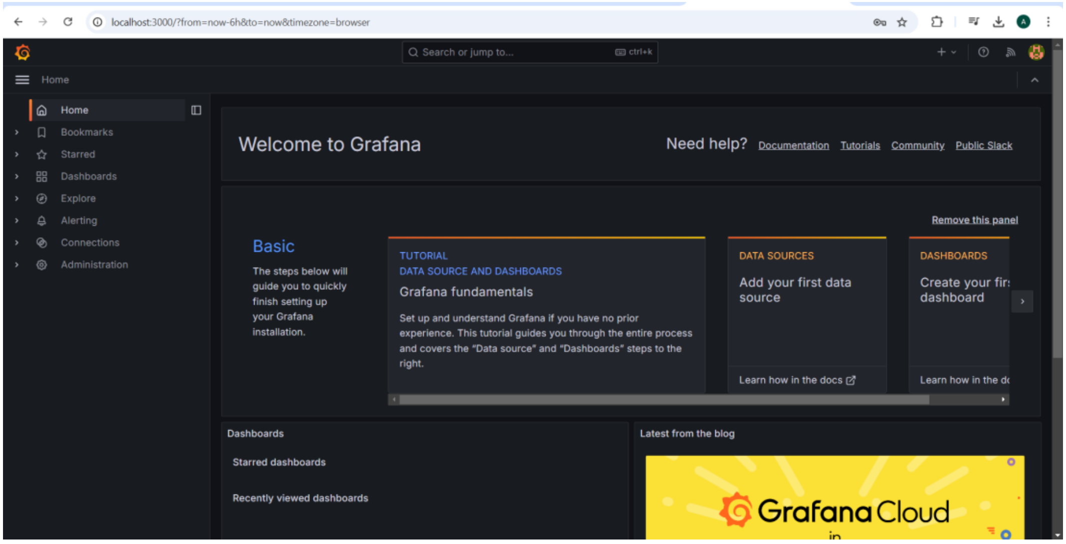
Let’s add our Django application as a target and install django-prometheus to expose metrics in a format Prometheus can scrape:

```
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ["localhost:9090"]
  - job_name: "django_app"

    static_configs:
      - targets: ['localhost:8000']
```

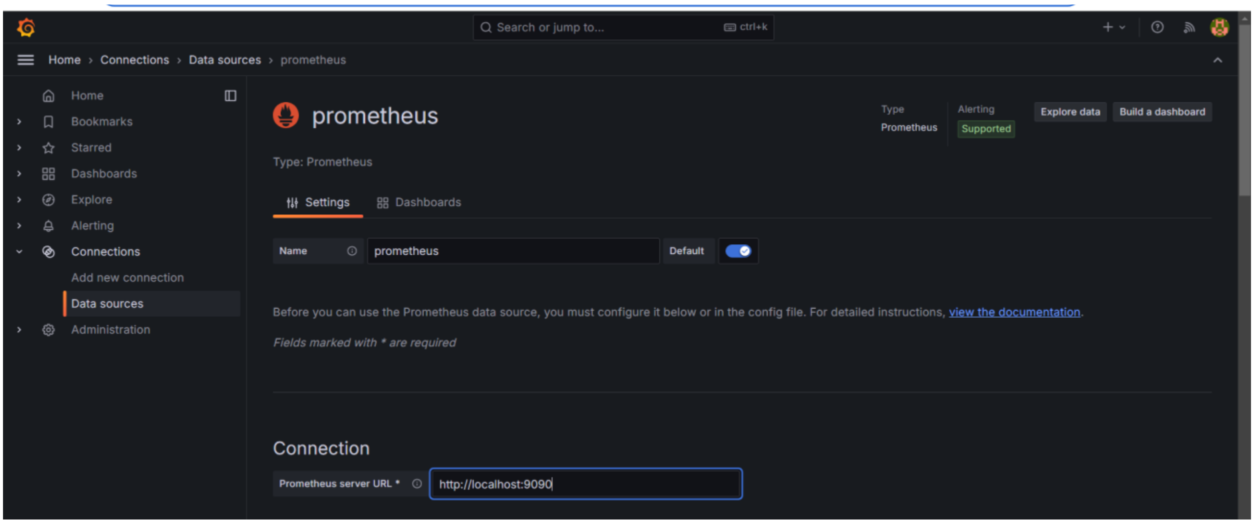
```
(.venv) PS C:\Users\wwwis\Documents\highload\assignments\store> pip install django-prometheus
```

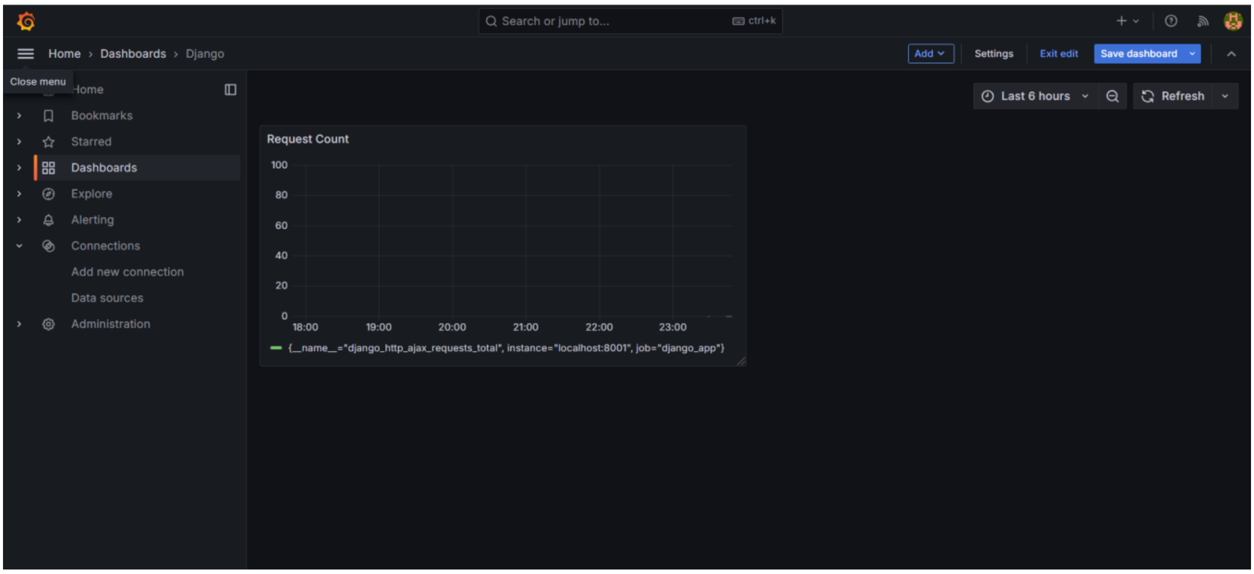


Now let’s install Grafana and create dashboards.



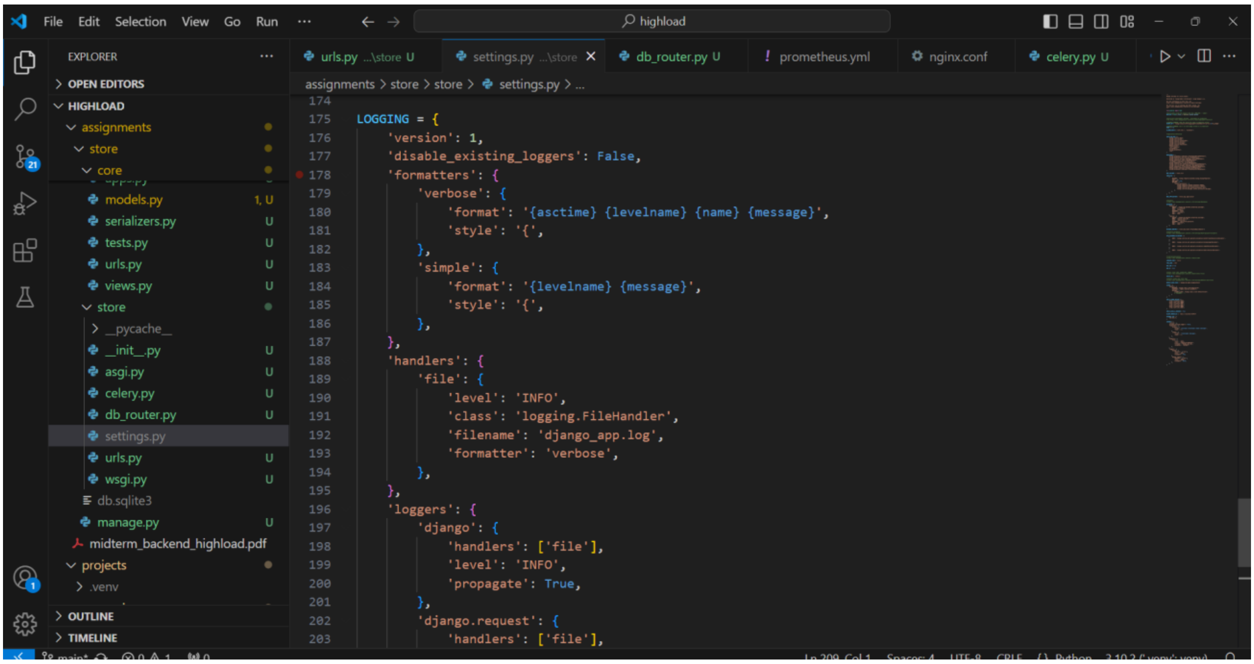
Now let’s add Prometheus URL to Data Sources

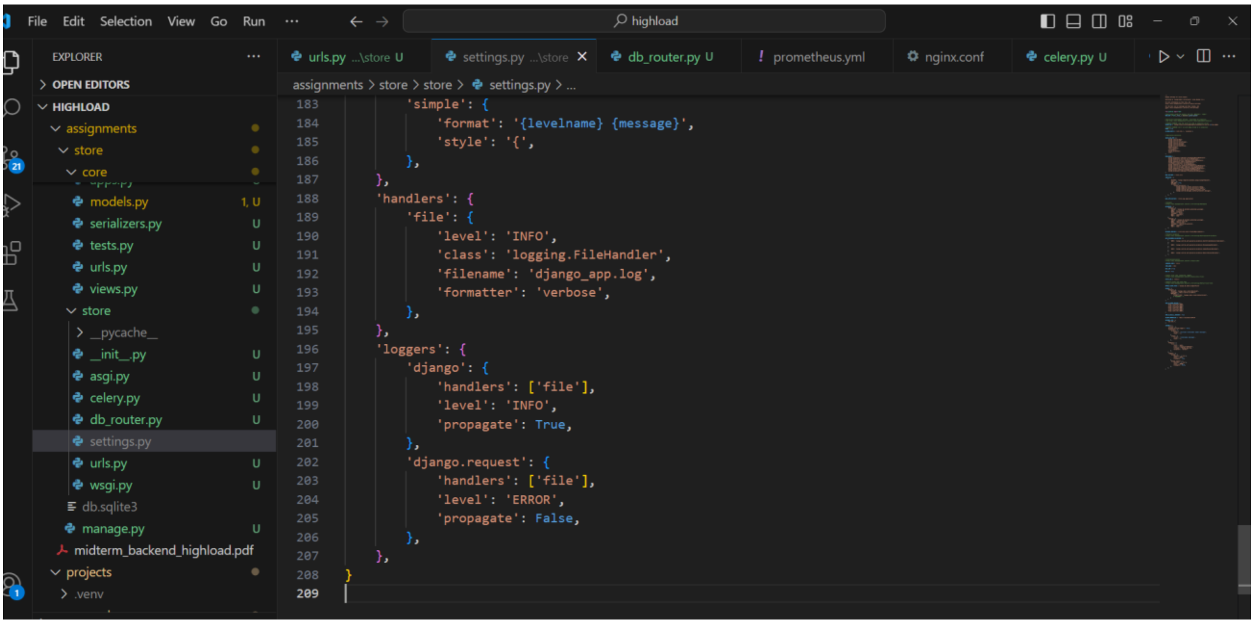




Exercise 6: Log Analysis

Django provides a flexible logging system that can be configured in settings.py.





Conclusion

The exercises cover important topics in designing and managing scalable, reliable, and consistent backend systems for distributed applications. Implementing a distributed key-value store and exploring consistency models highlighted the balance between availability, latency, and data integrity essential in distributed systems. Load balancing and database scaling shows the importance of scalability in ensuring application responsiveness under high traffic. The integration of monitoring tools like Prometheus and Grafana and logging help to identify bottlenecks and improve performance.