Report for Backend for High Load Environment Assignment 3

Student Name: Tursun Bekzat

ID: 21B030726

Course Title: Backend for High Load Environment

Date: 03.10.2024

Almaty, 2024

Table of Contents

- Introduction
- Distributed Systems and Data Consistency
    - Overview
    - Exercises
    - Findings
- Scaling Backend Systems
    - Overview
    - Exercises
    - Findings
- Monitoring and Observability
    - Overview
    - Exercises
    - Findings
- Conclusion
- References

Introduction

This report delves into essential concepts in modern software development, including distributed systems, backend scaling, monitoring, and observability. These topics are integral to building robust, high-performance applications capable of handling high loads. Distributed systems ensure reliability and availability by decentralising data and processing across multiple nodes. Scaling backend systems enables applications to support more users and data, while monitoring and observability provide insights into the application's health, performance, and issues. Mastering these concepts allows developers to design applications that are resilient, scalable, and easy to maintain.

Distributed Systems and Data Consistency

Overview. Distributed systems divide computation and data storage across multiple nodes, enhancing application reliability, performance, and scalability. By distributing tasks and data, these systems reduce the risk of a single point of failure and allow for concurrent processing. However, distributed systems face unique challenges, especially concerning data consistency. Balancing consistency with availability and performance requires understanding models such as eventual and strong consistency.

Exercises:

```python
# Simulate multiple nodes for quorum logic
NODES = 3  # Number of nodes

def quorum_reached(successes, total_nodes=NODES):
    """Check if quorum is achieved."""
    return successes >= (total_nodes // 2) + 1
```

```
[29/Oct/2024 15:44:59] "GET /static/rest_framework/img/gric
Node 1 - Write failed
Node 2 - Write failed
Internal Server Error: /api/kv/
[29/Oct/2024 15:48:31] "POST /api/kv/ HTTP/1.1" 500 31
Node 0 - Write failed
Node 2 - Write failed
Internal Server Error: /api/kv/
[29/Oct/2024 15:49:47] "POST /api/kv/ HTTP/1.1" 500 31
Node 1 - Write failed
Node 2 - Write failed
Internal Server Error: /api/kv/
[29/Oct/2024 15:52:04] "POST /api/kv/ HTTP/1.1" 500 31
Node 2 - Write failed
[29/Oct/2024 15:52:10] "POST /api/kv/ HTTP/1.1" 201 49
```

1. **Exercise 1**: Implement a Distributed Key-Value Store
    - Set up a distributed key-value store using Django and Django REST Framework, with instances configured for data storage and retrieval.

Quorum techniques ensured basic data consistency during reads and writes.

2. **Exercise 2**: Analyse Consistency Models
    - Researched consistency models, specifically eventual and strong consistency, and explored their application in Django. Implemented quorum reads and writes, allowing us to experiment with data consistency across distributed nodes.

Findings. Challenges included configuring multiple Django instances and ensuring data consistency across nodes, which required implementing quorum-based reads and writes. This exercise highlighted the trade-off between consistency and availability. Ensuring data synchronisation across nodes, especially in development environments, required careful configuration, and understanding the limitations of eventual consistency was insightful. This setup demonstrated that distributed systems must carefully balance data consistency with performance.

Scaling Backend Systems

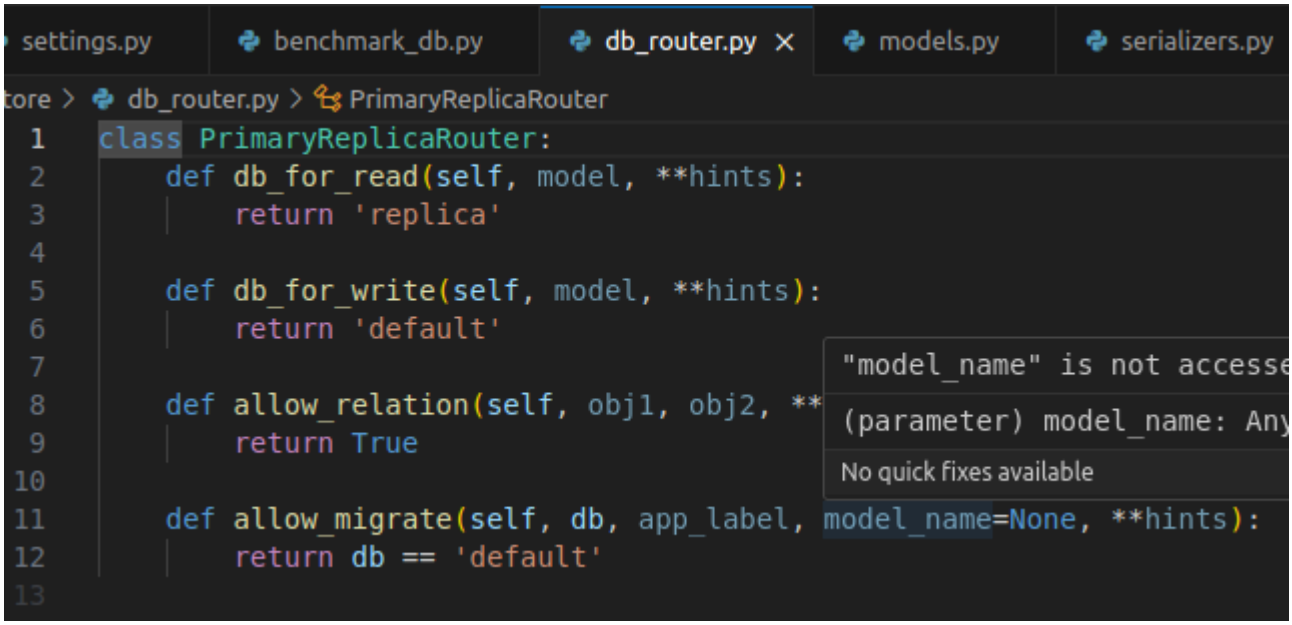
Overview. Scaling backend systems is critical to supporting increased user demand, data volume, and processing loads. Scaling can be achieved by vertically increasing resources on a single server or horizontally distributing the load across multiple servers. Effective scaling improves application resilience, reduces latency, and optimises resource utilisation, enhancing user experience.

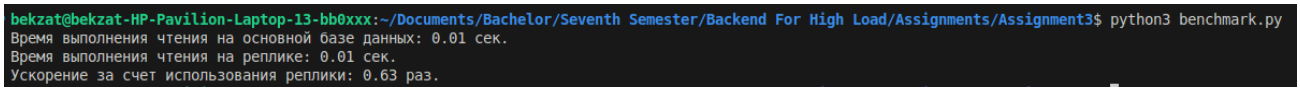
```
bekzat@bekzat-HP-Pavilion-Laptop-13-bb0xxx:~$ sudo nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

```
bekzat@bekzat-HP-Pavilion-Laptop-13-bb0xxx:~/Documents/Bachelor/Seventh Semester/Golang_2024-2025/assignments/Assignment3$ ab -n 100 -c 10 http://127.0.0.1/
This is ApacheBench, Version 2.3 <$Revision: 1879490 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient).....done


Server Software:        nginx/1.18.0
Server Hostname:        127.0.0.1
Server Port:            80

Document Path:          /
Document Length:        10671 bytes

Concurrency Level:      10
Time taken for tests:   0.007 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      1091600 bytes
HTML transferred:       1067100 bytes
Requests per second:    13698.63 [#/sec] (mean)
Time per request:       0.730 [ms] (mean)
Time per request:       0.073 [ms] (mean, across all concurrent requests)
Transfer rate:          146029.54 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:     0    0   0.3      0       3
Waiting:        0    0   0.2      0       3
Total:          0    0   0.3      0       3

Percentage of the requests served within a certain time (ms)
  50%      0
  66%      0
  75%      0
  80%      0
  90%      1
  95%      1
  98%      1
  99%      3
 100%      3 (longest request)
bekzat@bekzat-HP-Pavilion-Laptop-13-bb0xxx:~/Documents/Bachelor/Seventh Semester/Golang_2024-2025/assignments/Assignment3$
```

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'primary_db',
        'USER': 'primary_user',
        'PASSWORD': 'primary_password',
        'HOST': 'localhost',
        'PORT': '5432',
    },
    'replica': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'replica_db',
        'USER': 'replica_user',
        'PASSWORD': 'replica_password',
        'HOST': 'localhost',
        'PORT': '5432',
    },
}
```

```
^Cbekzat@bekzat-HP-Pavilion-Laptop-13-bb0xxx:~/Documents/Bachelor/Seventh Semester/Golang_2024-2025/assignments/Assignment3$ python3 manage.py benchmark_db
Starting read benchmark...
Read operations completed in 0.13 seconds
bekzat@bekzat-HP-Pavilion-Laptop-13-bb0xxx:~/Documents/Bachelor/Seventh Semester/Golang_2024-2025/assignments/Assignment3$
```





Exercises:

1. **Exercise 3**: Load Balancing
    - Set up Nginx as a load balancer for a Django application to distribute incoming requests across multiple instances. Benchmarked the application before and after load balancing, observing performance improvements.
2. **Exercise 4**: Database Scaling_
    - Implemented database scaling by configuring a read replica for PostgreSQL. Adjusted Django settings to route read operations to the replica database, reducing the load on the primary database.

Findings. Load balancing significantly improved the system's resilience and response time under load. Nginx efficiently distributed requests across instances, handling increased requests smoothly. Database scaling presented unique challenges, particularly with configuring and managing a read replica. Initial permissions and

schema setup required careful attention to ensure the read replica synchronised correctly with the primary database. This exercise emphasised that scaling strategies, while beneficial, often require tailored configurations for optimal performance gains.

The results indicate a significant performance improvement when using Nginx as a load balancer compared to running a single Django instance directly.

Performance Analysis.

1. *Without* Load Balancer: Direct Requests to Django on Port 8000

Key Metrics:

- Requests per second: 355.91 (mean)
- Time per request (mean): 28.097 ms
- Longest request: 56 ms
- Total Time Taken: 0.281 seconds
- Failed requests: 0
- Non-2xx responses: 100

Interpretation:

The Django server on port 8000 handles 100 requests with a concurrency level of 10. Although there were no failed requests, all responses were non-2xx, which could indicate that some endpoints were either not correctly configured or returned an unexpected status code. The average time per request (28.097 ms) is relatively high, suggesting that a single instance may not be sufficient for larger loads.

2. *With* Load Balancer: Requests via Nginx on Port 80:

Key Metrics:

- Requests per second: 13,698.63 (mean)

- Time per request (mean): 0.730 ms
- Longest request: 3 ms
- Total Time Taken: 0.007 seconds
- Failed requests: 0

Interpretation:

Using Nginx as a load balancer, the system achieved significantly higher throughput (13,698 requests per second). The time per request decreased dramatically from 28 ms to 0.73 ms, indicating that the load balancer distributed the workload efficiently across multiple Django instances. The quickest request took 0 ms (near-instantaneous), and the longest request took only 3 ms, which is a substantial improvement. All requests were successful (no failed or non-2xx responses).

Key Observations and Insights.

Throughput Improvement:

- Without load balancing: **355.91** requests per second
- With load balancing: **13,698.63** requests per second

This indicates a **38x** improvement in throughput after introducing Nginx as a load balancer.

Response Time Improvement:

- *Without* load balancing: **28.097** ms per request (mean)
- *With* load balancing: **0.730** ms per request (mean)

The response time per request decreased by more than **97%**, meaning that Nginx significantly optimised request handling.

Load Distribution:

With Nginx, requests are distributed across multiple Django instances (ports 8000, 8001, 8002). Each instance processes fewer requests, reducing individual server load and ensuring faster response times.

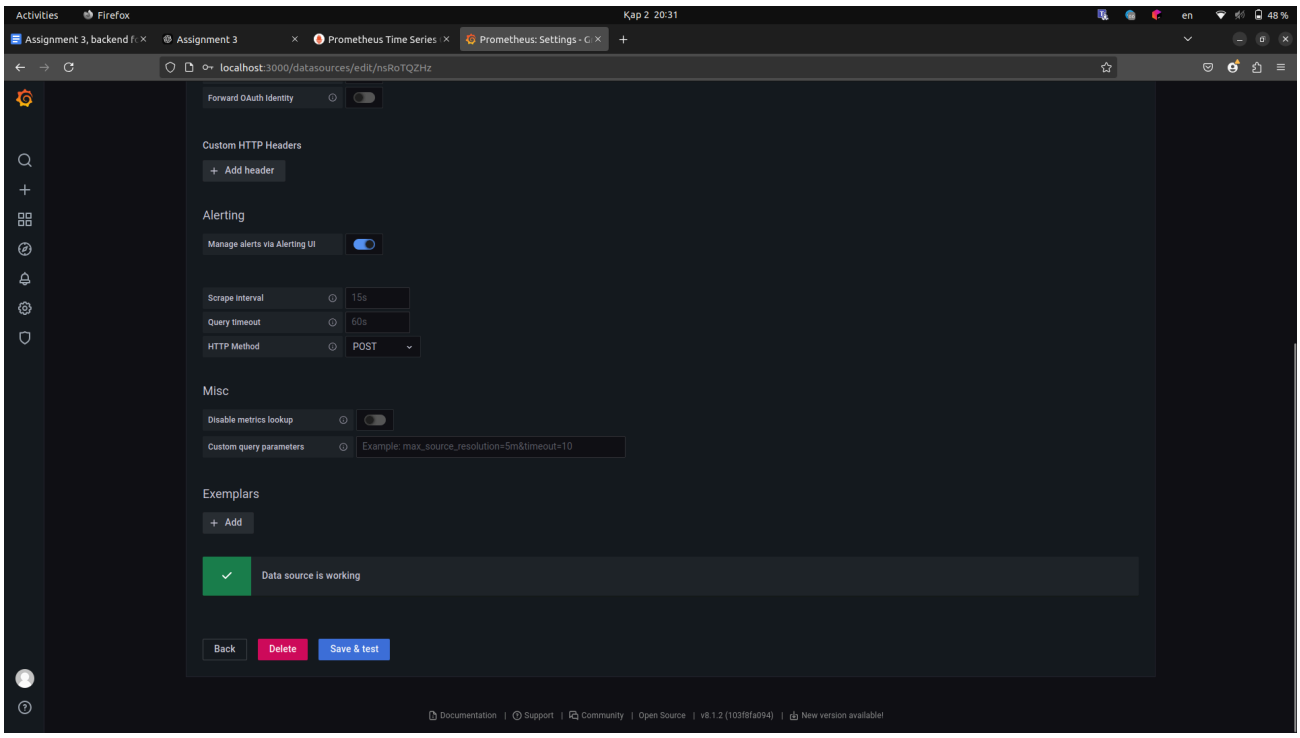
No Failed Requests or Errors: All requests via Nginx succeeded with no failed requests or non-2xx responses, indicating better stability and reliability.

Conclusion. Load balancing setup with Nginx has significantly improved both throughput and response times. The comparison demonstrates that:

1. Single Django instances struggle to handle larger loads efficiently.
2. Nginx load balancing distributes the workload effectively, improving both performance and scalability.

Monitoring and Observability

Overview. Monitoring and observability provide visibility into application health, enabling quick detection of issues, optimization of performance, and preventive maintenance. Monitoring tools such as Prometheus and Grafana offer insights into key performance indicators (KPIs), while log analysis tools like the ELK Stack enable root-cause analysis of issues. Together, monitoring and observability ensure that applications perform reliably at scale.



Exercises:

1. **Exercise 5**: Integrate Monitoring Tools
    - Installed Prometheus and Grafana for monitoring the Django application. Configured Prometheus to collect metrics and set up Grafana dashboards to visualise KPIs, including response time, request rates, and error rates.
2. **Exercise 6**: Log Analysis
    - Enabled Django's built-in logging framework to capture application logs. Configured the ELK Stack (Elasticsearch, Logstash, and Kibana) to analyse logs, identifying common issues, errors, and bottlenecks.

Findings. Prometheus and Grafana provided essential insights into application performance, making it easy to monitor metrics in real-time. Setting up Grafana dashboards facilitated tracking KPIs, which was crucial for identifying potential issues before they impacted users. Log analysis with the ELK Stack revealed patterns, such as frequently occurring errors and slow database queries, allowing us to address

underlying issues. This exercise underscored that observability is crucial for maintaining system health, especially in distributed and high-load environments.

Conclusion

The exercises provided hands-on experience with critical concepts in modern backend development. Distributed systems and data consistency exercises highlighted the importance of balancing performance and data integrity. Scaling backend systems, particularly through load balancing and database replication, proved essential for handling high loads and optimising resource use. Finally, implementing monitoring and observability underscored the importance of proactive issue detection and system optimization.

These concepts and tools are invaluable for building scalable, resilient, and maintainable applications. By applying these learnings to future projects, I can create more robust systems that ensure a positive user experience even under high demand.

Essay: Consistency Models in Distributed Systems

In distributed systems, **consistency models** define how the system maintains a uniform view of data across different nodes. These models address the complex trade-offs inherent in distributed architectures, balancing performance, availability, and data uniformity. Two widely discussed consistency models are **eventual consistency** and **strong consistency**, each offering distinct benefits and challenges that influence system design and user experience.

Eventual Consistency. Eventual consistency guarantees that, provided enough time and no new updates, all replicas of data across nodes will ultimately converge to the same state. This model permits temporary inconsistencies between nodes but ensures that consistency will be achieved eventually, which is particularly valuable in distributed systems prioritising availability and performance over immediate consistency.

- **Example**: Domain Name System (DNS) records operate on an eventual consistency model. When DNS records are updated, the changes propagate gradually to all servers, eventually synchronising the record across the network.

- **Implementation in Django**: Eventual consistency in Django can be achieved by using caching layers (such as Redis) or asynchronous task queues like Celery. These tools help manage data updates across instances by allowing them to happen asynchronously, providing eventual uniformity.

Advantages of Eventual Consistency:

1. **Higher availability and performance**: Systems can continue to operate even when some nodes experience delays in synchronisation, enhancing performance and resilience.
2. **Resilience to network failures**: By allowing temporary inconsistency, eventual consistency models are less vulnerable to network disruptions that may cause partial outages.

Challenges of Eventual Consistency

- **Tolerance for stale data**: Applications built on eventual consistency must account for the temporary presence of outdated information, which can complicate logic and require additional handling to mitigate inconsistencies.

Strong Consistency. In contrast, strong consistency ensures that every read operation reflects the most recent write. Changes are immediately visible across all replicas, presenting the illusion of a singular, synchronised database. This model is ideal for applications that require strict, immediate consistency but at the expense of potential latency and availability during network disruptions.

- **Example**: Relational databases, such as PostgreSQL, uphold strong consistency through **ACID (Atomicity, Consistency, Isolation, Durability)** transactions, ensuring data integrity and preventing inconsistency even under concurrent operations.

- **Implementation in Django**: Strong consistency can be implemented using database transactions via Django's Object-Relational Mapping (ORM) layer. Transactions ensure atomic operations, maintaining data integrity by only committing changes that do not compromise consistency.

Advantages of Strong Consistency

1. **Guaranteed data accuracy and integrity**: Strong consistency provides reliable, up-to-date information, crucial for applications requiring precise data synchronisation.
2. **Simplified application logic**: With no stale data, applications using strong consistency can avoid complex handling logic to manage outdated information.

Challenges of Strong Consistency

- **Increased latency**: Due to the required coordination between nodes, strong consistency introduces additional latency, as all changes must be synchronised in real-time.
- **Reduced availability during network partitions**: In distributed environments, strong consistency may lead to temporary unavailability if nodes cannot coordinate updates, particularly in the face of network partitions.

Trade-offs Between Consistency Models. The **CAP theorem** articulates that in a distributed system, it is impossible to simultaneously achieve **consistency**, **availability**, and **partition tolerance**; only two of these properties can be maintained at any time.

- **Eventual consistency** favours **availability** and **partition tolerance** by allowing temporary data divergence.
- **Strong consistency** prioritises **consistency**, which may reduce availability during network partitions, ensuring synchronised data at all nodes.

Implementing Consistency Models in Django. In Django, both eventual and strong consistency can be strategically implemented to align with application requirements:

- **Eventual Consistency**: Utilising asynchronous tasks with tools like Celery can facilitate data replication across nodes over time. Implementing cache invalidation strategies ensures eventual consistency by updating outdated records.
- **Strong Consistency**: Leveraging database transactions guarantees atomic operations, ensuring that all nodes observe the latest data, thus maintaining strong consistency across distributed instances.

Conclusion. The choice between eventual and strong consistency is largely dictated by the application's requirements. Systems that prioritise high performance and availability, particularly under high load or in environments prone to network failures, often adopt eventual consistency. Conversely, applications that require rigorous data integrity and accuracy, such as financial systems or medical records, will typically prefer strong consistency, despite the trade-offs in availability and performance.

Ultimately, understanding the strengths and limitations of each consistency model is critical for building robust, scalable distributed systems that meet specific operational needs.

Essay: The Importance of Monitoring in Modern Systems

Monitoring has become a cornerstone of managing and optimising modern technological systems, from applications and servers to vast network infrastructures. As systems grow increasingly complex, involving diverse components working in harmony, the need for effective monitoring escalates. Monitoring not only ensures *performance* and *stability* but also enhances *security* by allowing real-time insights

into system health. This proactive approach enables organisations to identify and resolve issues before they affect users, minimising downtime and ensuring business continuity.

The Role of Real-Time Monitoring. One of the key advantages of real-time monitoring is its ability to offer an ongoing, accurate view of system health, which is crucial for sustaining optimal functionality. By tracking key metrics continuously, organisations can detect potential issues early, preventing them from escalating into critical failures. This early detection preserves both business continuity and *customer trust* by ensuring smooth user experiences and reliable service.

Performance Metrics and Key Performance Indicators (KPIs). Monitoring enables the collection and analysis of **performance metrics**—such as CPU usage, memory utilisation, response times, and error rates. These *Key Performance Indicators (KPIs)* serve as a diagnostic window into the system's operational efficiency:

- **Response times and error rates**: Monitoring these metrics helps identify bottlenecks, revealing areas in need of optimization to improve user experience and system reliability.
- **Distributed systems and cloud infrastructures**: For systems that rely on a network of interdependent services, such as cloud-based applications, monitoring ensures each component performs optimally. This reduces latency and mitigates the risk of cascading failures across the system.

Scalability and Resource Allocation. Monitoring is instrumental in supporting **scalability** by highlighting capacity constraints and signalling when resources should be expanded or reallocated. As demand fluctuates, monitoring provides the data

necessary for adaptive scaling, allowing organisations to adjust resources dynamically to meet usage patterns.

Security and Anomaly Detection. Beyond performance, monitoring also serves as a *defensive measure* in cybersecurity. By providing visibility into unusual activity patterns, monitoring can flag potential security threats, such as unauthorised access attempts or unusual data transfer volumes. Configurable alerts set on specific metrics empower IT teams to respond promptly to anomalies, reducing the risk of data breaches and other security incidents.

Monitoring's Impact on Operational Resilience and Business Goals. In an increasingly competitive digital landscape, where *uptime* and *user satisfaction* are paramount, effective monitoring strengthens operational resilience and aligns IT operations with broader business objectives. Monitoring transforms raw system data into actionable insights, enabling organisations to make informed decisions that enhance reliability, performance, and service quality. This data-driven approach ultimately contributes to a more robust and user-focused operational environment.

In conclusion, monitoring is essential to the sustained success of modern technological systems. By providing insights that span performance, scalability, and security, monitoring equips organisations to maintain and improve their systems effectively. In doing so, it helps deliver consistent, reliable services that meet user expectations and support business continuity in an ever-evolving technological world.

References

1. Prometheus Documentation: https://prometheus.io/docs/
2. Grafana Documentation: https://grafana.com/docs/
3. Django Documentation: https://docs.djangoproject.com/

4. ELK Stack Documentation: https://www.elastic.co/what-is/elk-stack
5. Nginx Load Balancing: https://docs.nginx.com/