Kibana Setup, Prerequisites, Workflow, and Navigation Flow of Log Aggregation and Dashboards

Kibana is an open-source visualization tool that works with Elasticsearch to provide real-time analysis, visualization, and monitoring of logs and structured data. It is primarily used in observability, security, and business intelligence.

---

1. Prerequisites for Kibana Setup

Before setting up Kibana, ensure the following prerequisites:

1.1 Infrastructure Requirements

- **OS**: Kibana can run on Linux, Windows, or macOS.
- **RAM**: At least **4GB of RAM** (recommended: 8GB+ for production).
- **CPU**: Minimum **2 vCPUs** (recommended: 4+ vCPUs).
- **Storage**: SSD storage with high IOPS is recommended.
- **Networking**: Ensure Kibana can communicate with Elasticsearch on port **9200**.

1.2 Software Requirements

- **Elasticsearch**: Kibana is tightly integrated with Elasticsearch and requires a running Elasticsearch cluster.
- **Java** (for Elasticsearch, if applicable).
- **Node.js** (optional for development).
- **Nginx or Apache** (optional for reverse proxy and authentication).
- **Logstash / Beats** (optional for log aggregation).

1.3 Installation Steps

1.3.1 Install Kibana

1. **Download Kibana** from the Elastic website.
2. Extract the package:

tar -xzf kibana-<version>-linux-x86_64.tar.gz

cd kibana-<version>-linux-x86_64

3. Edit the configuration file (kibana.yml):

server.port: 5601

server.host: "0.0.0.0"

elasticsearch.hosts: ["http://localhost:9200"]

4. Start Kibana:

./bin/kibana

5. Open **http://localhost:5601/** in a web browser.

1.3.2 Configure Elasticsearch for Kibana

- Ensure that the Elasticsearch cluster is up and running.
- Update the Elasticsearch configuration (elasticsearch.yml):

cluster.name: my-cluster

node.name: node-1

network.host: 0.0.0.0

http.port: 9200

---

2. Workflow and Navigation Flow of Log Aggregation and Dashboards

2.1 Log Aggregation Process

1. **Log Collection**:
    - Use **Filebeat**, **Logstash**, or **Fluentd** to collect logs.
    - Example: Filebeat configuration (filebeat.yml):

filebeat.inputs:

- type: log

paths:

- /var/log/nginx/access.log

output.elasticsearch:

hosts: ["http://localhost:9200"]

- Start Filebeat:

sudo service filebeat start

2. **Log Ingestion and Parsing**:
    - Logstash pipelines can filter, transform, and enrich logs before sending them to Elasticsearch.

- Example Logstash configuration (logstash.conf):

```
input {
 beats {
  port => 5044
 }
}
filter {
 grok {
  match => { "message" => "%{COMMONAPACHELOG}" }
 }
}
output {
 elasticsearch {
  hosts => ["http://localhost:9200"]
 }
}
```

3. **Indexing in Elasticsearch**:
   - Logs are stored in indices with a schema.
   - Example index creation:

```
curl -X PUT "localhost:9200/logs-2025.02.06"
```

4. **Visualization in Kibana**:
   - Create **Data Views** (formerly known as Index Patterns) in Kibana.
   - Build visualizations and dashboards.

---

3. Key Components of Kibana

3.1 Data View (Formerly Index Patterns)

- Allows Kibana to query specific indices in Elasticsearch.
- Example:
  - Go to **Management > Stack Management > Data Views**.

- Click **Create Data View**.
- Enter logs-* as the index pattern.

3.2 Discover

- Explore raw logs with filters.
- Useful for debugging performance bottlenecks.

3.3 Visualizations

- Types of visualizations:
  - **Line charts** (for trends)
  - **Bar charts** (categorical data)
  - **Pie charts** (proportions)
  - **Maps** (geospatial data)
  - **Metric visualizations** (KPIs)

3.4 Dashboards

- Combine multiple visualizations into a single view.
- Example: Create a dashboard for **application latency monitoring**.
  1. Go to **Dashboards > Create Dashboard**.
  2. Add visualizations like:
     - **Average response time** (line chart).
     - **CPU/memory usage trends** (area chart).
     - **Error rate** (bar chart).

3.5 Alerts & Reporting

- Create alerts for anomalies (e.g., high latency).
- Example: If response time exceeds 500ms, send an alert to Slack.

---

4. Detecting Performance Issues in Kibana

Kibana can be used to monitor **application performance, infrastructure health, and bottlenecks** using various techniques.

4.1 Key Metrics to Track

- **Response time** (avg(response_time))

- **Error rate** (count(status_code: 500))
- **CPU and memory utilization** (avg(cpu_usage))
- **Database query performance** (avg(query_execution_time))
- **Elasticsearch query latency** (search.latency)

4.2 Techniques for Performance Issue Detection

4.2.1 Slow Queries Analysis

1. **Enable Elasticsearch Slow Query Logs**
    - Update elasticsearch.yml:

      index.search.slowlog.threshold.query.warn: 2s

      index.search.slowlog.threshold.fetch.warn: 1s
    - Query logs in Kibana:

```
{
 "query": {
  "match": {
   "message": "took more than 2s"
  }
 }
}
```

2. **Create Kibana Visualization**
    - Use **"Query Time vs Request Count"** visualization.
    - Identify peak query times.

4.2.2 Detecting JVM GC Issues

1. **Ship GC logs to Elasticsearch**:
    - Use **Filebeat or Logstash**.
    - Example Logstash filter:

```
filter {
 grok {
  match => { "message" => "%{NUMBER:gc_time}" }
 }
```

}

2. **Analyze in Kibana**:
    - Create a dashboard for gc_time trends.
    - Set alerts for excessive GC pauses.

4.2.3 Memory & CPU Bottlenecks

- Use **Metricbeat** to collect CPU and memory metrics.
- Example metricbeat.yml:

```
metricbeat.modules:
 - module: system
  metricsets:
   - cpu
   - memory
  period: 10s
  output.elasticsearch:
   hosts: ["http://localhost:9200"]
```

- Kibana dashboard can show:
    - **CPU usage spikes**.
    - **High memory consumption**.
    - **Correlate with application logs**.

4.2.4 Analyzing Load Balancer and API Gateway Logs

- If using **AWS ALB logs**, ship them via Logstash.
- Example filter:

```
filter {
 grok {
  match => { "message" => "%{TIMESTAMP_ISO8601:timestamp}
%{NUMBER:response_time}" }
 }
}
```

- Create a Kibana dashboard to analyze:

- **Response time trends.**
- **High error rate periods.**
- **Spikes in load balancer latency.**

Detecting Performance Issues in Kibana – A Technical Deep Dive

Its absolutely possible to detect and troubleshoot performance issues using **Kibana**, provided it is integrated with **Elasticsearch, Logstash, Metricbeat, and APM (Application Performance Monitoring)**. Kibana enables **real-time monitoring**, **log aggregation**, **data visualization**, and **alerting** to identify bottlenecks in application and infrastructure performance.

---

1. Key Performance Issues That Kibana Can Detect

Kibana, in combination with Elasticsearch and Beats, can help identify the following **performance bottlenecks**:

1. **Application Performance Issues**
    - High response times for API requests.
    - Increased error rates (HTTP 500, 503, etc.).
    - Slow database queries affecting user experience.
    - Garbage Collection (GC) pauses affecting Java applications.
2. **Infrastructure Issues**
    - High CPU and memory usage on application servers.
    - Network latency and packet loss in Kubernetes clusters.
    - Load balancer latency affecting request routing.
3. **Database Performance Issues**
    - Slow SQL queries with high execution times.
    - Increased database connection wait times.
    - High database CPU usage.
4. **Elasticsearch Performance Issues**
    - High query response time in Elasticsearch.
    - Slow indexing speed due to large ingestion rates.
    - High heap usage leading to frequent GC pauses.

---

2. Key Techniques for Detecting Performance Issues in Kibana

To detect performance issues using Kibana, we follow **multiple techniques**, including **real-time monitoring, anomaly detection, time-series analysis, and alerting**.

2.1 Log Analysis for Application Performance Issues

By ingesting application logs into **Elasticsearch**, we can detect slow response times, errors, and exceptions.

Step 1: Enable Application Logging

If using Java Spring Boot, configure logging with ELK:

```
logging:
 level:
  root: INFO
  org.springframework.web: DEBUG
  org.hibernate: ERROR
```

Ensure logs are shipped via **Filebeat**:

```
filebeat.inputs:
 - type: log
  paths:
    - "/var/log/app.log"
output.elasticsearch:
    hosts: ["http://localhost:9200"]
```

Start Filebeat:

```
sudo service filebeat start
```

Step 2: Create a Data View in Kibana

- Navigate to **Stack Management > Data Views > Create Data View**.
- Select app-logs-* as the index pattern.

Step 3: Build a Dashboard

- Create a **line chart visualization** for:
    - **Average Response Time** (avg(response_time))
    - **Error Rate** (count(status_code: 500))

- Use the **"Discover" tab** to filter logs:

```
{
 "query": {
  "match": {
   "status_code": "500"
  }
 }
}
```

- Set alerts if response time exceeds a threshold (e.g., **>500ms**).

2.2 Monitoring System Metrics (CPU, Memory, Disk, Network)

System-level performance issues like **high CPU, memory leaks, and disk bottlenecks** can be monitored using **Metricbeat**.

Step 1: Configure Metricbeat

Install and configure **Metricbeat**:

```
sudo apt-get install metricbeat
```

Edit the metricbeat.yml file:

```
metricbeat.modules:
 - module: system
  metricsets: ["cpu", "memory", "network"]
  period: 10s
  output.elasticsearch:
   hosts: ["http://localhost:9200"]
```

Start Metricbeat:

```
sudo service metricbeat start
```

Step 2: Build a Dashboard

- Use **"Lens" visualization** in Kibana to create:
  - **CPU Usage (%) Over Time** (system.cpu.total.pct)
  - **Memory Consumption Trends** (system.memory.used.pct)

- **Disk I/O Latency**
- **Network Packet Drops**

Step 3: Set Alerts

- **Trigger alerts** when CPU usage > **85%** for 5 minutes.
- **Trigger alerts** if memory utilization crosses **90%**.

2.3 Slow Query Analysis for Database Performance

Database slow queries often impact overall system performance.

Step 1: Enable Slow Query Logs

For **MySQL**, modify my.cnf:

```
slow_query_log = 1
long_query_time = 2
log_output = FILE
slow_query_log_file = /var/log/mysql-slow.log
```

For **PostgreSQL**, modify postgresql.conf:

```
log_min_duration_statement = 2000
log_statement = 'all'
```

Step 2: Ship Logs to Elasticsearch

Modify **Filebeat configuration** (filebeat.yml):

```
filebeat.inputs:
 - type: log
  paths:
   - "/var/log/mysql-slow.log"
output.elasticsearch:
 hosts: ["http://localhost:9200"]
```

Start Filebeat:

```
sudo service filebeat restart
```

Step 3: Create a Kibana Dashboard

- **Visualize slow queries by execution time**.

- Query example:

```
{
 "query": {
  "range": {
   "query_time": {
    "gte": "2s"
   }
  }
 }
}
```

Step 4: Set Alerts

- If **query execution time** exceeds **5s**, trigger an alert.
- If **DB connections exceed a threshold**, send a Slack alert.

2.4 Elasticsearch Performance Monitoring

Since Kibana relies on **Elasticsearch**, monitoring Elasticsearch itself is critical.

Step 1: Enable Slow Query Logs

Modify elasticsearch.yml:

```
index.search.slowlog.threshold.query.warn: 2s
index.search.slowlog.threshold.fetch.warn: 1s
```

Step 2: Query Elasticsearch Metrics in Kibana

Query Elasticsearch for slow queries:

```
{
 "query": {
  "match": {
   "message": "took more than 2s"
  }
 }
}
```

Step 3: Set Up an Elasticsearch Performance Dashboard

- **Query response time trend (search.latency)**
- **Slow indexing rate (indexing.latency)**
- **JVM heap usage (jvm.mem.heap_used_percent)**

Step 4: Detect High Heap Usage

If JVM Heap usage > **75%**, potential **GC issues**. Create alerts to restart Elasticsearch nodes if necessary.

2.5 APM-Based Application Performance Monitoring

Using **Elastic APM**, we can trace application transactions and detect slow endpoints.

Step 1: Install APM Server

```
sudo apt-get install apm-server
```

Modify apm-server.yml:

```
apm-server:
 host: "0.0.0.0:8200"
output.elasticsearch:
 hosts: ["http://localhost:9200"]
```

Start APM:

```
sudo service apm-server start
```

Step 2: Integrate with Java Application

Add the **Elastic APM agent**:

```
<dependency>
 <groupId>co.elastic.apm</groupId>
 <artifactId>apm-agent-attach</artifactId>
 <version>1.27.0</version>
</dependency>
```

Start the application with APM:

```
java -javaagent:/path/to/elastic-apm-agent.jar -
Delastic.apm.server_urls=http://localhost:8200 -jar myapp.jar
```

Step 3: Create APM Dashboards

- **Track slow transactions**.
- **Monitor database query execution time**.
- **Identify high-latency APIs**.

---

3. Conclusion

✓ Kibana can detect application, system, database, and Elasticsearch performance issues.
✓ Using Filebeat, Metricbeat, and APM, you can ingest logs and metrics into Elasticsearch for analysis.
✓ Dashboards and alerts help detect bottlenecks in real time.
✓ Anomaly detection can identify performance degradations before they impact users.

Case Study 1: Slow Application Response Times Due to High Load

Scenario:

A banking application was experiencing **intermittent slow response times**, especially during peak hours. The latency of some API endpoints exceeded **5 seconds**, causing timeouts.

Step 1: Data Collection

- **Logs** from the application were collected using **Filebeat**.
- **Metrics** (CPU, memory, thread pool) were collected using **Metricbeat**.
- **Transaction traces** were collected using **Elastic APM**.

Step 2: Kibana Dashboard Setup

A **custom dashboard** was created in Kibana to monitor:

- **Average API Response Time (ms)**
- **Error Rate (%)**
- **JVM Heap Usage (%)**
- **CPU Utilization (%)**

Query to Identify Slow API Calls:

```
{
  "query": {
    "range": {
      "response_time": {
        "gte": "5000"
      }
    }
  }
}
```

Visualization: API Response Time Over Time

- A **line chart** plotted API response times.
- A **bar chart** visualized slow API calls grouped by endpoint.

Step 3: Root Cause Analysis

1. **High CPU Usage:**
    - Kibana showed **CPU spikes above 90%** during peak hours.
    - JVM **Garbage Collection (GC) logs** showed **long GC pauses (avg. 2s)**.
2. **Thread Pool Exhaustion:**
    - **Thread dump analysis** indicated that the application **ran out of worker threads**, causing request queueing.
    - Too many active threads in java.util.concurrent.ThreadPoolExecutor.
3. **Database Bottleneck:**
    - Database logs (shipped via **Filebeat**) showed **slow SQL queries** running over **3 seconds**.
    - Query example:

    SELECT COUNT(*) FROM transactions WHERE user_id = ?;

Step 4: Resolution & Optimization

✓ **JVM tuning:** Increased heap memory allocation and enabled **G1GC** for better memory management.

✓ **Thread pool tuning:** Increased worker thread count and optimized connection pool settings.
✓ **Database indexing:** Added indexes to frequently queried columns in the transactions table.

Alerting Configuration:

- Trigger alert **if API response time exceeds 2s for 5 minutes**.
- Send Slack notifications using:

```
{
  "actions": {
    "slack_notification": {
      "message": "High API latency detected (>2s). Check JVM and DB logs."
    }
  }
}
```

Case Study 2: Elasticsearch Slow Query Performance

Scenario:

An **e-commerce platform** using **Elasticsearch** for product searches reported **slow query performance**, affecting customer experience.

Step 1: Enabling Slow Query Logs

Elasticsearch **slow query logging** was enabled:

```
index.search.slowlog.threshold.query.warn: "2s"
index.search.slowlog.threshold.fetch.warn: "1s"
```

Step 2: Kibana Dashboard Setup

A **custom dashboard** in Kibana monitored:

- **Query Execution Time**
- **Search Latency Over Time**
- **Indexing Performance**

Query to Find Slow Queries

```
{
  "query": {
```

```
    "range": {
     "took": {
      "gte": "2000"
     }
    }
   }
}
```

Visualization: Query Latency Histogram

- A **histogram** grouped slow queries by execution time.
- A **heatmap** mapped query execution times against product categories.

Step 3: Root Cause Analysis

1. **Large Result Sets**:
    - Some **search queries** returned **100,000+ results**, causing excessive data transfer.
    - Example slow query:

```
{
 "query": {
  "match": {
   "product_description": "wireless headphones"
  }
 },
 "size": 100000
}
```

    - Solution: **Limit size to 50** and implement **pagination**.
2. **Expensive Aggregations**:
    - Kibana logs showed **slow queries with aggregations**:

```
{
 "aggs": {
  "avg_price": {
   "avg": {
```

```
        "field": "price"
      }
    }
  }
}
```

- Solution: Precompute aggregation results in a **separate index**.

3. **High JVM Heap Usage**:
   - Elasticsearch **JVM heap exceeded 80%**, leading to **frequent GC pauses**.
   - Solution: **Increase heap size** (-Xms8g -Xmx8g) and **use circuit breakers**.

Step 4: Resolution & Optimization

✓ **Query optimization:** Used match_phrase instead of match for better relevancy.
✓ **Index sharding:** Increased the number of **shards from 3 to 6** to improve search distribution.
✓ **Elasticsearch caching:** Enabled request_cache for repeated queries.

Alerting Configuration:

- If **search latency exceeds 2s**, trigger an alert in Kibana.
- Log alert messages:

```
{
  "actions": {
    "email_notification": {
      "message": "Elasticsearch query latency exceeded 2s. Investigate!"
    }
  }
}
```

Case Study 3: Kubernetes Network Latency Issues

Scenario:

A **Kubernetes-based microservices system** was experiencing **intermittent high network latency**, causing API timeouts.

Step 1: Collecting Logs and Metrics

- **Filebeat** collected application logs.

- **Metricbeat** collected CPU, memory, and **network statistics**.

Step 2: Kibana Dashboard Setup

The following visualizations were created:

- **Latency Heatmap:** API response times across regions.
- **Pod-Level Network Traffic:** Network requests per Kubernetes pod.

Query to Detect High Latency

```
{
 "query": {
  "range": {
   "latency": {
    "gte": "2000"
   }
  }
 }
}
```

Step 3: Root Cause Analysis

1. **Packet Drops in Kubernetes Pods**
    - Kibana logs showed **high packet drops** in specific nodes:

```
{
 "event.dataset": "system.network",
 "network.dropped.packets": ">1000"
}
```

2. **AWS Load Balancer Latency**
    - Analyzed **AWS ALB logs** and found **p99 latency spikes > 3s**.
3. **Intermittent DNS Resolution Delays**
    - CoreDNS logs showed **delays resolving service names**.

Step 4: Resolution & Optimization

✓ **Increased Kubernetes pod replicas** to distribute load.
✓ **Enabled CoreDNS caching** for faster DNS resolution.
✓ **Optimized AWS ALB configurations** by enabling **HTTP/2 keep-alive**.

Alerting Configuration:

- If **API latency exceeds 1s**, trigger an alert:

```
{
  "actions": {
    "slack_notification": {
      "message": "High network latency detected! Check Kubernetes pods."
    }
  }
}
```

Hands-on Examples for Setting Up Alerts, Queries, and Visualizations in Kibana

Below are **step-by-step** examples to set up **queries, visualizations, and alerts** in **Kibana** for performance monitoring. These examples cover **application monitoring, database slow queries, Elasticsearch performance, and Kubernetes network latency**.

---

1. Setting Up Queries in Kibana (Discover Tab)

Example 1: Querying Slow API Requests

If an API response time exceeds **500ms**, we need to track these slow responses.

Steps:

1. Open **Kibana > Discover**.
2. Select the appropriate **Data View** (formerly Index Pattern), e.g., app-logs-*.
3. Use the following **KQL (Kibana Query Language) filter**:

```
response_time > 500
```

4. Click **"Update"** to see results.
5. Save the query for future analysis.

Equivalent Elasticsearch JSON Query

If querying Elasticsearch directly:

```
{
 "query": {
  "range": {
   "response_time": {
    "gte": 500
   }
  }
 }
}
```

Example 2: Detecting Frequent HTTP 500 Errors

To find all logs where the application returns **HTTP 500** errors:

KQL Query in Kibana Discover

status_code: 500

Equivalent Elasticsearch Query

```
{
 "query": {
  "match": {
   "status_code": "500"
  }
 }
}
```

Example 3: Identifying Slow Database Queries

If SQL queries take more than **2 seconds**, use this query.

KQL Query

query_time > 2000

Elasticsearch JSON Query

```
{
 "query": {
  "range": {
   "query_time": {
    "gte": 2000
   }
  }
 }
}
```

---

2. Creating Visualizations in Kibana

Example 1: Visualizing API Response Time Trends

1. **Go to Kibana → Visualize**.
2. Click **Create Visualization**.
3. Select **Line Chart**.
4. Choose the data source **app-logs-***.
5. Set:
    - **Y-axis:** Average response_time.
    - **X-axis:** Time (@timestamp).
6. Click **Save** and add it to a **Dashboard**.

Example 2: Slow Query Distribution in Database

1. **Go to Kibana → Visualize**.
2. Click **Create Visualization**.
3. Choose **Bar Chart**.
4. Select **db-logs-*** as the data source.
5. Configure:
    - **Y-axis:** Count of logs.
    - **X-axis:** Terms aggregation on query_time.
    - Add a filter: query_time > 2000.
6. Click **Save** and add it to a **Dashboard**.

---

3. Creating Dashboards in Kibana

Example: Application Performance Dashboard

1. **Go to Kibana → Dashboard**.
2. Click **Create New Dashboard**.
3. Click **Add Existing Visualizations**:
    - API Response Time Trend (Line Chart).
    - Slow Database Queries (Bar Chart).

- HTTP 500 Errors (Metric Count).

4. Save the dashboard as **"App Performance Dashboard"**.

---

4. Setting Up Alerts in Kibana

Example 1: Alert for High API Response Times

1. **Go to Kibana → Stack Management → Rules & Connectors**.
2. Click **Create Rule**.
3. Choose **Elasticsearch Query**.
4. Set:
   - Name: **High API Latency Alert**
   - Index: **app-logs-***.
   - Query:

```
{
 "query": {
  "range": {
   "response_time": {
    "gte": 1000
   }
  }
 }
}
```

5. Set a **Trigger Condition**:
   - If response_time > 1000ms for 5 minutes.
6. **Choose Action**:
   - Send an **email alert** or **Slack notification**.
   - Message:

```
{
 "message": "Warning! API response time exceeded 1s. Investigate possible performance issues."
}
```

7. Save the rule.

Example 2: Alert for Elasticsearch Slow Queries

1. **Go to Kibana → Stack Management → Rules & Connectors**.
2. Click **Create Rule**.
3. Choose **Elasticsearch Query**.
4. Set:
    - Name: **Slow Query Alert**
    - Index: **es-logs-***.
    - Query:

```
{
 "query": {
  "range": {
   "took": {
    "gte": 2000
   }
  }
 }
}
```

5. Set Trigger Condition:
    - If took > 2000ms for 3 consecutive queries.
6. **Choose Action**:
    - Send an alert via **Slack**:

```
{
 "message": "Warning! Elasticsearch queries are taking more than 2s. Investigate query performance!"
}
```

7. Save the rule.

Example 3: Alert for Kubernetes Pod Network Latency

1. **Go to Kibana → Stack Management → Rules & Connectors**.
2. Click **Create Rule**.
3. Choose **Elasticsearch Query**.
4. Set:
    - Name: **Kubernetes Network Latency Alert**
    - Index: **k8s-metrics-***.
    - Query:

```
{
 "query": {
  "range": {
   "latency": {
    "gte": 500
   }
  }
 }
}
```

5. Set Trigger Condition:
    - If latency > 500ms for more than **5 minutes**.
6. **Choose Action**:
    - Send an **email alert** to DevOps team.
    - Restart affected pods using **Kubernetes API calls**.

---

5. Real-time Monitoring Using Kibana

Example: Monitoring Real-time Logs

1. **Go to Kibana → Observability → Logs**.
2. Select **Live Stream**.
3. Apply a filter:

response_time > 1000

4. Observe real-time log entries.

Step-by-Step Kibana Hands-on Lab

This hands-on lab will guide you through setting up **Kibana**, ingesting logs and metrics, visualizing performance data, and creating alerts.

---

1. Setup the ELK Stack (Elasticsearch, Logstash, Kibana)

1.1 Install Elasticsearch

Step 1: Download and Install Elasticsearch

Run the following commands to download and extract Elasticsearch:

```
wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-8.4.0-linux-x86_64.tar.gz
tar -xzf elasticsearch-8.4.0-linux-x86_64.tar.gz
cd elasticsearch-8.4.0
```

Step 2: Configure Elasticsearch

Edit the configuration file config/elasticsearch.yml:

```
network.host: 0.0.0.0
http.port: 9200
cluster.name: "elk-cluster"
node.name: "node-1"
```

Step 3: Start Elasticsearch

./bin/elasticsearch

Verify it is running:

curl -X GET "http://localhost:9200"

1.2 Install Kibana

Step 1: Download and Install Kibana

```
wget https://artifacts.elastic.co/downloads/kibana/kibana-8.4.0-linux-x86_64.tar.gz
tar -xzf kibana-8.4.0-linux-x86_64.tar.gz
cd kibana-8.4.0
```

Step 2: Configure Kibana

Edit the configuration file config/kibana.yml:

```
server.port: 5601

server.host: "0.0.0.0"

elasticsearch.hosts: ["http://localhost:9200"]
```

Step 3: Start Kibana

```
./bin/kibana
```

Access Kibana in your browser at **http://localhost:5601/**.

1.3 Install Logstash (Optional, for Log Parsing)

If you need to process logs before storing them in Elasticsearch:

```
wget https://artifacts.elastic.co/downloads/logstash/logstash-8.4.0-linux-x86_64.tar.gz

tar -xzf logstash-8.4.0-linux-x86_64.tar.gz

cd logstash-8.4.0
```

Configure **Logstash Pipeline** (logstash.conf):

```
input {
 file {
  path => "/var/log/app.log"
  start_position => "beginning"
 }
}
filter {
 grok {
  match => { "message" => "%{TIMESTAMP_ISO8601:timestamp} %{LOGLEVEL:loglevel}
%{GREEDYDATA:message}" }
 }
}
output {
 elasticsearch {
  hosts => ["http://localhost:9200"]
```

```
    index => "app-logs"
  }
 }
```

Run Logstash:

```
./bin/logstash -f logstash.conf
```

---

2. Ingest Logs and Metrics

2.1 Ship Application Logs Using Filebeat

Step 1: Install Filebeat

wget https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-8.4.0-linux-x86_64.tar.gz

tar -xzf filebeat-8.4.0-linux-x86_64.tar.gz

cd filebeat-8.4.0

Step 2: Configure Filebeat

Edit filebeat.yml:

```
filebeat.inputs:
 - type: log
   paths:
    - "/var/log/app.log"


output.elasticsearch:
hosts: ["http://localhost:9200"]
```

Step 3: Start Filebeat

```
./filebeat -e
```

2.2 Collect System Metrics Using Metricbeat

Step 1: Install Metricbeat

wget https://artifacts.elastic.co/downloads/beats/metricbeat/metricbeat-8.4.0-linux-x86_64.tar.gz

tar -xzf metricbeat-8.4.0-linux-x86_64.tar.gz

cd metricbeat-8.4.0

Step 2: Configure Metricbeat

Edit metricbeat.yml:

```
metricbeat.modules:
 - module: system
   metricsets: ["cpu", "memory", "load"]
   period: 10s
   output.elasticsearch:
    hosts: ["http://localhost:9200"]
```

Step 3: Start Metricbeat

```
./metricbeat -e
```

---

3. Create a Data View in Kibana

1. Open **Kibana → Stack Management → Data Views**.
2. Click **"Create Data View"**.
3. Enter app-logs-* as the pattern.
4. Select the timestamp field (@timestamp).
5. Click **Save**.

---

4. Create Visualizations

4.1 Visualizing API Response Time

1. Go to **Kibana → Visualize**.
2. Click **Create Visualization**.
3. Choose **Line Chart**.
4. Select **app-logs-*** as the data source.
5. Configure:
    - **Y-axis:** avg(response_time)
    - **X-axis:** @timestamp
6. Save and add to the **Dashboard**.

4.2 Detecting Frequent HTTP 500 Errors

1. Go to **Kibana → Visualize**.
2. Click **Create Visualization**.
3. Choose **Bar Chart**.
4. Select **app-logs-*** as the data source.
5. Configure:
    - **Y-axis:** count(status_code: 500)
    - **X-axis:** @timestamp
6. Save the visualization.

---

5. Create a Dashboard

1. Go to **Kibana → Dashboard**.
2. Click **Create Dashboard**.
3. Click **Add Existing Visualizations**:
    - API Response Time (Line Chart).
    - HTTP 500 Errors (Bar Chart).
4. Save as **"Performance Monitoring Dashboard"**.

---

6. Set Up Alerts

6.1 Alert for High API Response Times

1. Go to **Kibana → Stack Management → Rules & Connectors**.
2. Click **Create Rule**.
3. Choose **Elasticsearch Query**.
4. Set:
    - Name: **High API Latency Alert**
    - Index: **app-logs-***.
    - Query:

```
{
 "query": {
```

```
      "range": {
        "response_time": {
          "gte": 1000
        }
      }
    }
  }
}
```

5. Set Trigger Condition:
    - If response_time > 1000ms for 5 minutes.
6. **Choose Action**:
    - Send an **email or Slack notification**.

---

7. Real-time Log Monitoring

1. **Go to Kibana → Observability → Logs**.
2. Select **Live Stream**.
3. Apply filter:

   response_time > 1000

---

8. Advanced Monitoring

8.1 Elasticsearch Query Performance Monitoring

Enable **slow query logs** in Elasticsearch:

index.search.slowlog.threshold.query.warn: "2s"

Query slow logs in Kibana:

```
{
  "query": {
    "match": {
      "message": "took more than 2s"
    }
  }
}
```

Correlating GC Logs, Heap Dumps, and JVM Metrics in Kibana

This step-by-step guide will help you correlate **GC logs, heap dumps, and JVM performance metrics** in Kibana to detect and analyze Java application performance issues such as **memory leaks, high GC pauses, and excessive heap consumption**.

---

1. Why Correlate GC Logs, Heap Dumps, and JVM Metrics?

Java applications running in production often suffer from **performance bottlenecks** related to:

- **Frequent Full GC pauses** causing application slowdowns.
- **High heap usage** leading to OutOfMemoryError (OOM).
- **Memory leaks** where objects are not being garbage collected.
- **Thread contention** affecting response times.

By **correlating GC logs, heap dumps, and JVM metrics in Kibana**, we can: ✓ Identify **slow GC cycles** and their impact on response times.
✓ Detect **heap memory spikes** and investigate retained objects.
✓ Monitor **CPU and thread activity** to find inefficiencies.

---

2. Setting Up Log Collection for JVM Monitoring

2.1 Enabling GC Logs in Java

Java provides options to log **Garbage Collection (GC) activities**.
Modify the Java startup command:

```
java -Xlog:gc*:file=/var/log/gc.log:time,level,tags -jar myapp.jar
```

For Java 8:

```
java -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:/var/log/gc.log -jar myapp.jar
```

2.2 Collecting GC Logs Using Filebeat

1. **Install Filebeat**:

   ```
   wget https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-8.4.0-linux-x86_64.tar.gz
   tar -xzf filebeat-8.4.0-linux-x86_64.tar.gz
   cd filebeat-8.4.0
   ```

2. **Configure Filebeat to Collect GC Logs**
   Edit filebeat.yml:

```
filebeat.inputs:
 - type: log
   paths:
    - "/var/log/gc.log"
   multiline.pattern: "^[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}:[0-9]{2}"
   multiline.negate: true
   multiline.match: after

output.elasticsearch:
 hosts: ["http://localhost:9200"]
 index: "gc-logs"
```

3. **Start Filebeat**:

   ./filebeat -e

2.3 Collecting JVM Metrics Using Metricbeat

Metricbeat can collect **JVM heap usage, thread counts, and CPU load**.

1. **Install Metricbeat**:

   wget https://artifacts.elastic.co/downloads/beats/metricbeat/metricbeat-8.4.0-linux-x86_64.tar.gz

   tar -xzf metricbeat-8.4.0-linux-x86_64.tar.gz

   cd metricbeat-8.4.0

2. **Enable the Jolokia Module for JVM Monitoring**:

   metricbeat modules enable jolokia

3. **Configure Metricbeat (metricbeat.yml)**:

```
metricbeat.modules:
 - module: jolokia
   metricsets: ["jvm"]
   hosts: ["http://localhost:8778/jolokia"]
```

```
namespace: "jvm"
period: 10s
output.elasticsearch:
 hosts: ["http://localhost:9200"]
```

4. **Start Metricbeat**:

```
./metricbeat -e
```

2.4 Capturing Heap Dumps for Memory Leak Analysis

1. **Capture a Heap Dump When an OOM Occurs** Modify Java options:

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/var/dumps/heapdump.hprof
```

2. **Manually Capture a Heap Dump**

```
jmap -dump:format=b,file=/var/dumps/heapdump.hprof <PID>
```

3. **Ship Heap Dump Metadata to Elasticsearch** Use Filebeat to collect heap dump metadata (heap-analyzer.log):

```
filebeat.inputs:
 - type: log
  paths:
   - "/var/log/heap-analyzer.log"
```

---

3. Creating a Data View in Kibana

1. **Go to Kibana → Stack Management → Data Views**.
2. Click **"Create Data View"**.
3. Enter gc-logs-* as the pattern.
4. Select the timestamp field (@timestamp).
5. Click **Save**.

Repeat this for:

- jvm-metrics-*
- heap-metadata-*

---

4. Creating Visualizations in Kibana

4.1 GC Pause Duration Over Time

1. **Go to Kibana → Visualize**.
2. **Create a Line Chart**.
3. Set:
    - **Y-axis:** avg(gc_pause_time).
    - **X-axis:** @timestamp.
4. **Filter:** gc_type: "Full GC".
5. **Save the visualization**.

4.2 Heap Usage Trend Analysis

1. **Go to Kibana → Visualize**.
2. **Create an Area Chart**.
3. Set:
    - **Y-axis:** avg(heap_used).
    - **X-axis:** @timestamp.
4. **Save and add to a Dashboard**.

4.3 Correlating GC Logs and Heap Usage

1. **Create a New Dashboard**.
2. Add:
    - **GC Pause Duration** (Line Chart).
    - **Heap Usage Trend** (Area Chart).
    - **Thread Count** (Metric Visualization).
3. Apply filters:

    gc_pause_time > 2s AND heap_used > 80%
4. Save the **"JVM Performance Dashboard"**.

---

5. Setting Up Alerts

5.1 Alert for Long GC Pauses

1. **Go to Kibana → Stack Management → Rules & Connectors**.
2. **Create a New Rule**.
3. Select **Elasticsearch Query**.
4. Set:
    - Name: **"High GC Pause Detected"**.
    - Index: **gc-logs-***.
    - Query:

```
{
 "query": {
  "range": {
   "gc_pause_time": {
    "gte": 2000
   }
  }
 }
}
```

5. **Trigger Conditions**:
    - If **GC Pause > 2s** for **3 occurrences**.
6. **Set Action**:
    - Send **Slack alert**:

```
{
"message": "High GC Pause detected! Investigate heap usage and GC logs."
}
```

5.2 Alert for High Heap Usage

1. **Go to Kibana → Stack Management → Rules & Connectors**.
2. **Create a New Rule**.
3. Select **Elasticsearch Query**.

4. Set:
    - Name: **"Heap Usage Alert"**.
    - Index: **jvm-metrics-***.
    - Query:

```
{
 "query": {
  "range": {
   "heap_used": {
    "gte": 85
   }
  }
 }
}
```

5. **Trigger Conditions**:
    - If **heap usage > 85%** for **5 minutes**.
6. **Set Action**:
    - Send an **email alert to DevOps team**.
