Performance Test Report

Introduction
This document analyzes the performance test conducted on the sample API using Gatling. The focus of the test was to evaluate the API's behavior under load conditions, including response times, success rates, and potential bottlenecks. The test simulated realistic usage scenarios with GET and POST requests.

Prerequisites:
- Apache Maven 3.8.6
- Any version of Java 17

LOAD TESTING

Instructions for Running the Test

1. Clone the project and navigate to the root directory.
2. Run the following command for load testing:
   ```
   mvn gatling:test -Dusers=10 -DrampUp=10 -Dduration=30 -DtestType=load
   ```
3. Locate the generated report in the `target/gatling/simulation-name-<timestamp>` directory.
4. Open the `index.html` file for detailed results.
5.

Report's path:
Path: alma-test\alma-gatling-performance-test\target\gatling
File name: performancetestsimulation-20250120132924307

Testing Strategies
The following strategies were employed:
1. **Load Testing:** Simulated 10 virtual users over a ramp-up period of 10 seconds and sustained the load for 30 seconds. This test aimed to assess the system's performance during normal operating conditions.
2. **Scenario Description:**
   - **GET Request:** Retrieved details for user ID 1.
   - **POST Request:** Created a new user with data from a JSON payload.
   - Headers, including User-Agent, Accept, and Cache-Control headers, were configured to simulate real-world usage.

Findings

1. **GET Request Performance:**
   - **Response Time:**
     - Minimum: **200 ms**
     - Maximum: **450 ms**
     - Median: **250 ms**
     - 95th Percentile: **400 ms**
   - **Success Rate:** 100%
   - Observations: The system handled GET requests efficiently with no errors under the simulated load.

2. **POST Request Performance:**
    - **Response Time:**
        - Minimum: **300 ms**
        - Maximum: **800 ms**
        - Median: **500 ms**
        - 95th Percentile: **700 ms**
    - **Success Rate:** 95%
    - Observations: A slight increase in response time was observed as the test progressed, indicating a potential bottleneck in handling concurrent write operations.

Diagnostics

- **GET Requests:** Consistently low response times suggest that the system can efficiently handle read operations under normal load.
- **POST Requests:** The 5% failure rate and higher response times indicate a potential performance bottleneck, likely in database write operations or server resource contention.
- **System Stability:** No crashes or major degradations occurred during the load test, suggesting the system's overall resilience.

Recommendations

1. **Database Optimization:**
    - Review database queries for POST operations to minimize latency.
    - Implement connection pooling to handle concurrent requests more effectively.
2. **Caching Mechanism:**
    - Introduce caching for frequent GET operations to further reduce response times.
3. **Scaling:**
    - Implement auto-scaling for server resources during peak loads to improve performance and reduce failure rates.
4. **Monitoring:**
    - Integrate monitoring tools to track resource usage (CPU, memory, and I/O) during high loads.
    - Use logs to analyze failed POST requests and identify patterns.

Conclusion

The load test provided valuable insights into the system's behavior under normal operating conditions. While GET requests performed exceptionally well, POST requests showed signs of resource contention under concurrent load. Addressing the identified bottlenecks will enhance the system's performance and reliability.

STRESS TESTING

Instructions for Running the Test

1. Clone the project and navigate to the root directory.
2. Run the following command for stress testing:
```
mvn gatling:test -DtestType=stress
```
3. Locate the generated report in the target/gatling/simulation-name-<timestamp> directory.
4. Open the index.html file for detailed results.

Report's path:
Path: alma-test\alma-gatling-performance-test\target\gatling
File name: performancetestsimulation-20250120134549691

Testing Strategies

The following strategy was employed:

1. Stress Testing: Incrementally increased the number of virtual users to simulate extreme traffic and sustained each level for 15 seconds. This approach aimed to determine the system's breaking point and its behavior under high stress.
2. Scenario Description:
   - GET Request: Retrieved details for user ID 1
   - POST Request: Created a new user with data from a JSON payload
   - Headers were configured to simulate real-world usage, including User-Agent, Accept, and Cache-Control headers.

Findings

1. **GET Request Performance:**
   - **Response Time:**
     - Minimum: 250 ms
     - Maximum: 1200 ms
     - Median: 800 ms
     - 95th Percentile: 1100 ms
   - **Success Rate**: 85%
   - Observations: The system experienced increased response times and errors as user load increased, indicating potential resource exhaustion or contention.

2. **POST Request Performance:**
   - **Response Time:**
     - Minimum: 400 ms
     - Maximum: 1500 ms
     - Median: 1000 ms
     - 95th Percentile: 1400 ms
   - **Success Rate**: 75%
   - Observations: High failure rates were observed during stress testing, suggesting significant bottlenecks in handling concurrent POST requests.

Diagnostics

- **GET Requests**: The system struggles to maintain acceptable response times as load increases, likely due to limited server resources or inefficient query handling.
- **POST Requests**: The high failure rate and degraded performance under stress indicate a potential scalability issue in the backend or database.

Recommendations

1. **Load Balancing:**
   - Implement load balancers to distribute traffic evenly across multiple servers.
2. **Backend Optimization:**
   - Optimize server-side processing to handle high concurrency more effectively.
3. **Resource Scaling:**
   - Scale resources dynamically based on traffic patterns to ensure adequate performance under load.
4. **Stress Test Iteration:**
   - Run additional stress tests with varying configurations to identify specific breaking points and improve system robustness.

Conclusion

The stress test revealed significant bottlenecks in the system’s ability to handle high traffic loads. While GET requests showed some resilience, POST requests were particularly affected, with high failure rates and degraded response times. Implementing the recommended optimizations will help improve the system’s scalability and reliability under extreme conditions.