API Performance Testing (k6)

1 Performance Test Objectives

The objective of API performance testing was to evaluate the responsiveness, stability, and scalability of DummyJSON APIs under different load conditions. Performance testing aimed to ensure that selected API endpoints can handle concurrent users while maintaining acceptable response times.

2 Target Endpoints

The following API endpoints were selected for performance testing, as they represent critical backend operations:

- DELETE /products/1
- GET /products
- GET /products/{id}

Additional endpoints were included to provide broader backend coverage during execution.

3 Smoke and Load Profiles

Two performance testing profiles were defined using k6:

- **Smoke Testing:** Executed with a small number of virtual users (3 users) for a short duration to verify basic system availability and responsiveness.
- **Load Testing:** Executed with 25 concurrent virtual users to simulate sustained user traffic and evaluate system behavior under load conditions.

These profiles helped identify performance characteristics at different usage levels.

Ramp-up Strategy

A gradual ramp-up strategy was applied during load testing to simulate realistic user behavior. Instead of introducing all virtual users at once, users were increased progressively to allow the system to stabilize under increasing load.

This approach helped identify performance degradation points and reduced the risk of sudden spikes that do not reflect real-world usage patterns.

Think Time Strategy

Think time was incorporated into performance testing scripts to simulate realistic user behavior. A short delay was added between consecutive API requests to represent the time a user typically spends reviewing data or navigating between actions.

Introducing think time helps avoid unrealistically aggressive request patterns and provides more accurate performance insights under normal usage conditions.

4 KPIs and Thresholds

Key performance indicators (KPIs) were defined to assess API performance:

- Average response time
- 95th percentile response time (p95)
- Request throughput (requests per second)
- Error rate

Thresholds were configured to ensure that the 95th percentile response time remained below 2 seconds and that the error rate stayed within acceptable limits.

KPI Interpretation and Bottleneck Identification

Performance metrics collected during k6 execution were analyzed to identify system behavior under load. The average and 95th percentile response times remained within acceptable thresholds, indicating that the APIs can handle concurrent requests efficiently under normal conditions.

Throughput measurements demonstrated stable request processing rates during both smoke and load tests. An increased error rate observed during full collection execution was primarily related to the inclusion of negative test scenarios and authentication-related dependencies.

Potential bottlenecks identified include repeated authentication requests under load and backend processing limitations of the mock DummyJSON service. These bottlenecks are not representative of production systems but provide valuable insights into interpreting performance results when using simulated backends.

5 Execution Results

The k6 performance tests were executed successfully for both smoke and load profiles. Results indicated that API response times remained within the defined thresholds, with the 95th percentile response time staying well below the 2-second limit.

Performance metrics such as response time distribution, throughput, and error rates were captured from the k6 execution output and documented as execution evidence.

6 Performance Analysis and Recommendations

Performance analysis revealed that DummyJSON APIs demonstrated strong response time performance under concurrent load. Observed error rates were mainly attributed to negative test scenarios, authentication dependencies, and limitations inherent to the mock backend environment.

Potential performance bottlenecks include repeated authentication requests and backend processing overhead during concurrent access. Recommended improvements include caching authentication tokens, isolating negative scenarios from performance test runs, optimizing backend request handling, and introducing rate limiting and connection pooling to improve scalability.