Testing Manual

29.0.2024

Introduction

This manual outlines the comprehensive testing strategy and processes for the Smart Inventory Web App. It serves as a definitive guide for quality assurance team members, developers, and stakeholders involved in the testing process. Our goal is to ensure the highest quality, reliability, and user satisfaction for our application.

Test Strategy

Our test strategy adopts a shift-left approach, integrating testing throughout the development lifecycle. We focus on early defect detection, continuous integration, and maintaining a balance between automated and manual testing efforts.

Key Principles

- Early and continuous testing
- Automation-first approach
- Risk-based testing prioritization
- Continuous feedback and improvement

Types of Tests

- Unit Tests: Verify individual functions, methods, and small code units in isolation.
- Component Tests: Validate the functionality of isolated UI components.
- Integration Tests: Ensure different parts of the application work together correctly.
- End-to-End (E2E) Tests: Simulate real user scenarios, validating the entire application workflow.
- API Tests: Verify the functionality and reliability of AWS backend services.
- Performance Tests: Evaluate the system's responsiveness and stability under various load conditions.
- Security Tests: Identify vulnerabilities and ensure data protection.

Test Tools

Cypress

- Primary tool for unit, component and E2E testing
- Provides a browser-based testing environment
- Supports automated test script creation and execution

AWS CloudWatch

- Monitors and logs AWS service performance
- Helps in identifying and debugging issues in AWS resources

Quality Assurance Requirements

- Reliability:
    - Ensured comprehensive test coverage across all test types.
    - load testing
- Usability:
    - Primarily addressed by E2E tests and usability testing sessions.
- Maintainability:
    - Enhanced by well-documented, modular test suites.
- Scalability:
    - Verified through load testing and AWS service configuration tests.
- Security:
    - Ensured through regular vulnerability scan audits and role-based tests.

Testing Processes

Component Tests

- Use Cypress Component Testing
- Mount individual Angular components in isolation
- Interact with components using Cypress commands
- Assert expected behavior and DOM changes

Integration Tests

- Test interactions between different modules or services
- Use Cypress for frontend integration testing
- Implement API mocking when necessary

End-to-End (E2E) Tests

- Write tests simulating complete user journeys Use Cypress for E2E testing Interact with UI elements using
- Cypress commands Assert expected application state and API responses Run tests with npx cypress run

API Tests

- API testing was conducted using the built-in API tester in the Amazon API Gateway console.

Test Data Management

Maintain a separate test database with anonymized data Use data factories to generate test data programmatically Implement data cleanup procedures after test execution Version control test data alongside the codebase

Continuous Integration and Deployment

- Integrate all tests into the CI/CD pipeline run unit and integration tests before every merge to main or develop.
- Execute E2E tests before deployment to staging.
- Perform smoke tests after each production deployment.

Issue Tracking and Resolution

Use a dedicated issue tracking system (e.g., Jira, GitHub Issues) Categorize and prioritize issues based on severity and impact Implement a clear workflow for issue resolution and retesting Conduct regular triage meetings to review and update issue status

Non-Functional Requirements Testing

Usability

Reference usability doc

Security

Security Vulnerability Scanning

Initial Scan Results using Hosted Scan (only provides one free scan a month)

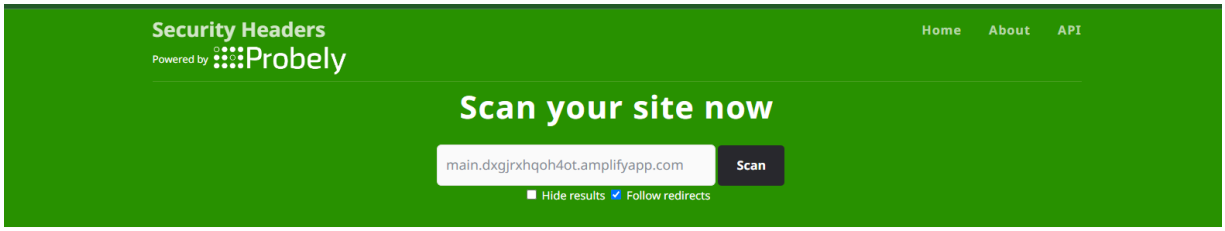
ZAP Scan Summary: Key Findings vs. Our Configuration

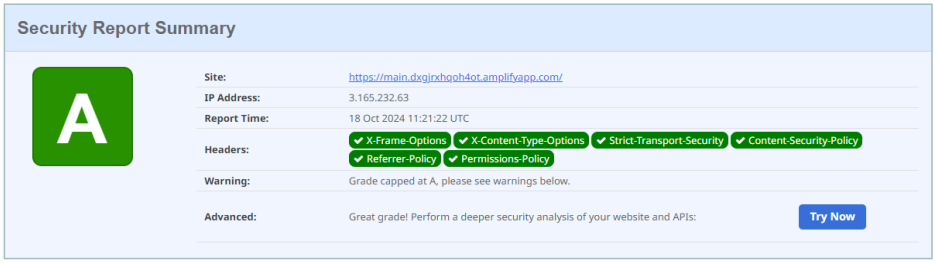
ZAP Scan Results

1. Content Security Policy (CSP)
   Finding: Medium risk - "unsafe-eval", "unsafe-inline"
   Our Config: Intentionally allows these for external resources
   Rationale: Necessary for current styling functionality with CDNs
2. Cross-Domain Configuration
   Finding: Medium risk - misconfiguration
   Our Config: Broad access allowed
   Rationale: Supports current integration needs
3. Server Information Leakage
   Finding: Low risk
   Action Needed: Review server-side configurations
4. Cache-Control
   Finding: Informational - re-examine directives
   Our Config: `public, max-age=3600, must-revalidate`
   Rationale: Balances performance and content freshness
5. Additional Security Measures
   Implemented: X-Frame-Options, X-Content-Type-Options, HSTS, Referrer-Policy, Permissions-Policy

After implementing the recommendations that could be implemented as suggested from our OWASP ZAP scan, Proberly's free Security Headers Tool was used to scan our headers. We used two scan tools for headers as OWASP ZAP can sometimes give false positives.

Security Header Check Tool (Probely): An automated scanning tool that evaluates a website's HTTP security headers. It checks for the presence and correct implementation of important security headers like Content-Security-Policy, X-Frame-Options, and Strict-Transport-Security. The tool provides a quick assessment of a site's basic security posture and offers recommendations for improvement based on current best practices.





Scalability and Reliability

The system should maintain optimal performance as user load increases, ensuring a smooth experience during peak usage periods.

User Concurrency Tactics

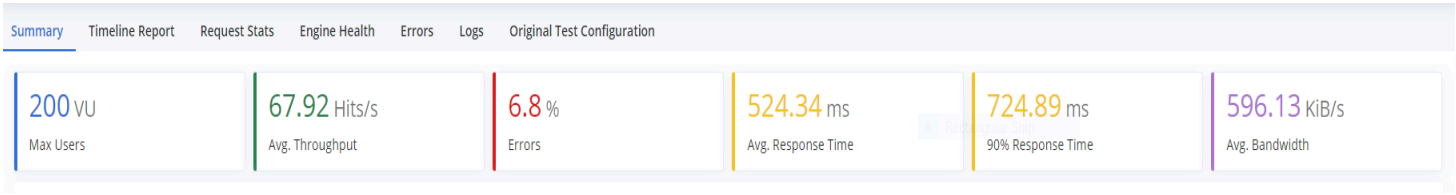
- Implement load balancing to distribute traffic evenly
- Utilize caching mechanisms for frequently accessed data
- Optimize database queries and indexing

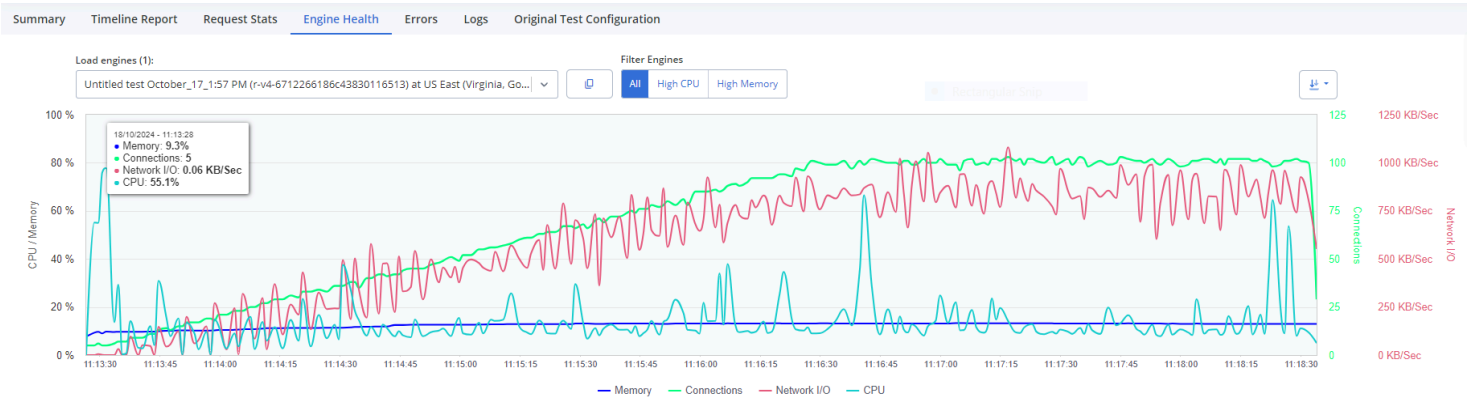
Quantification

- Response Time: Maintain response times under 10 seconds for 95% of requests with 100 concurrent users
- Scalability: Sustain performance for up to 200 concurrent users with minimal degradation

Load Testing Methodology

- Tool: EveryStep
- Users: 200 simulated users
- Ramp-up Time: 1 minute
- Test Scope: Adding inventory items, navigating pages (inventory, suppliers, reports), accessing dashboard





Key Findings

- Dashboard Performance: Highest error rate due to resource allocation issues
- Other Pages: Minor errors observed across various sections

Transaction Processing Tactics

- Implement asynchronous processing for non-critical operations
- Optimize resource allocation, particularly for dashboard functionality

Quantification

● Error Rate: Reduce dashboard errors by 50% under full load of 500 users

● Success Rate: Maintain a 95% success rate for all transactions under 200 concurrent users

Performance Optimization Tactics

● Implement more efficient resource allocation for dashboard components

● Enhance caching strategies for frequently accessed data

● Optimize database queries and indexing for common operations

By focusing on these areas, we aim to ensure that the system can handle the specified user loads while maintaining acceptable response times and minimizing errors, particularly in resource-intensive areas like the dashboard.

Load Test Report: JMeter Results Analysis

Test Overview

Test Type: Load Test
Tool: Apache JMeter 5.6.3
Date: October 18, 2024

Test Configuration

- **Number of Threads (Users):** 500
- **Ramp-up Period:** 30 seconds
- **Loop Count:** 2
- **Total Samples:** 10,000 (1,000 per HTTP request)

Test Results

Response Times

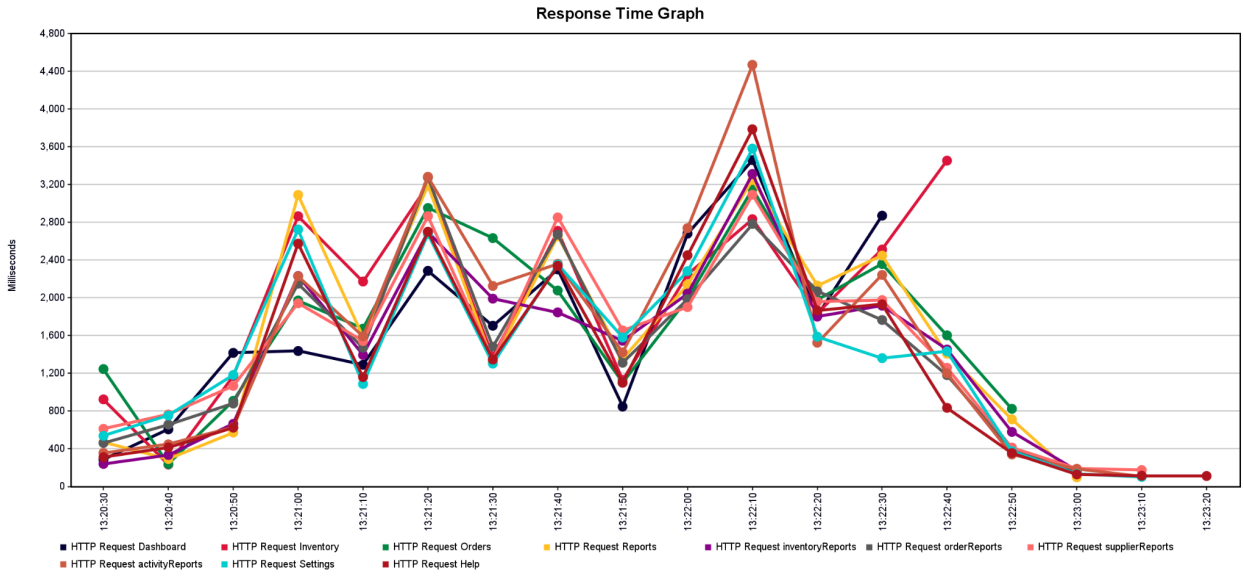


Image 1: Response Time per Page

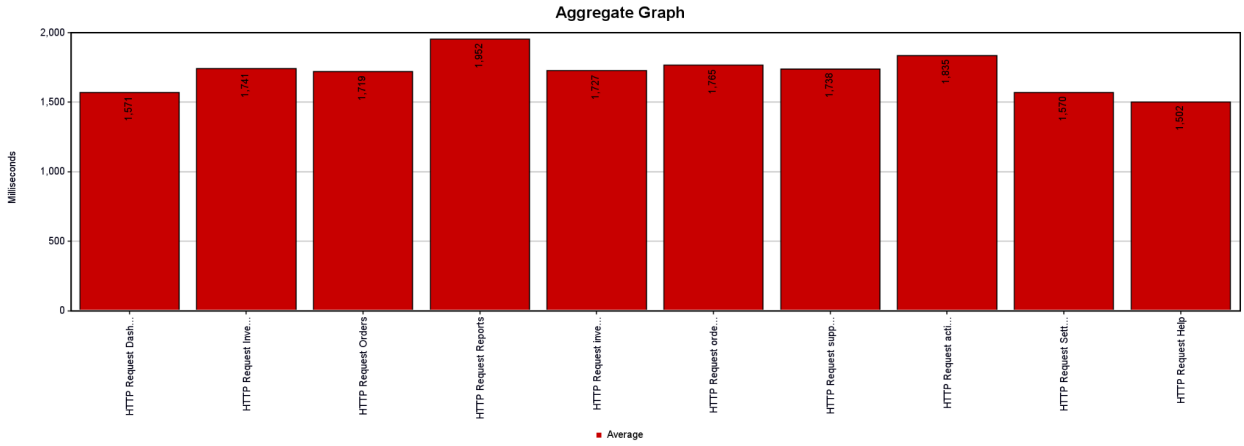


Image 2: Average Response Time per Page

Metric	**Value (ms)**
Average	1712
90% Line	794
95% Line	4674
99% Line	5530
	7048

Performance Metrics

- **Throughput:** 54.6 requests/second
- **Error Rate:** 0% (no errors reported)
- **Received KB/sec:** 9547.79
- **Sent KB/sec:** 7.53

Performance by Request

Request Type	Average Response Time (ms)
HTTP Request Reports	1952 (slowest)
HTTP Request Dashboard	1571 (fastest)

Analysis

1. **Response Times:** The average response time (1712 ms) exceeds Google's recommended 1-second threshold for optimal user experience. While the median response time (794 ms) is acceptable, the 90th percentile and above show significant slowdowns that could impact user satisfaction.
2. **Throughput:** The system handled 54.6 requests per second without errors, indicating good stability under load.
3. **Error Rate:** Zero errors reported suggest robust error handling and system stability.
4. **Resource Utilization:** High received KB/sec (9547.79) compared to sent KB/sec (7.53) indicates asymmetric data flow, typical for content-heavy responses.
5. **Request Performance:** There is notable variation in response times across different request types, with HTTP Request Reports being the slowest and HTTP Request Dashboard being the fastest. This suggests potential areas for targeted optimization.
6. **Scalability Considerations:** While the system performed without errors, the increasing response times at higher percentiles indicate potential scalability challenges under heavier loads.
7. **User Experience Impact:** The 95th percentile response time of 5530 ms suggests that a significant portion of users may experience noticeable delays, potentially affecting overall user satisfaction and engagement.

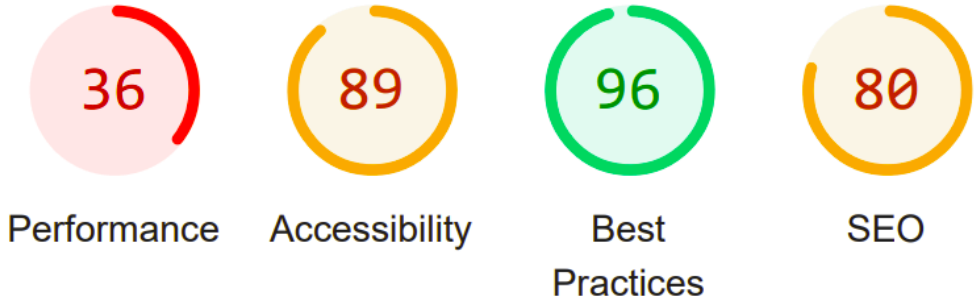
Motivation for the use of JMeter and EveryStep

JMeter simulates heavy loads on systems, helping them identify performance issues and determine capacity. EveryStep complements this by enabling easy test script creation

through browser recording. Together, these tools allow us to create and execute user scenarios at scale. This approach ensures applications can handle expected loads, resulting in more robust and reliable software.

Performance and Reliability

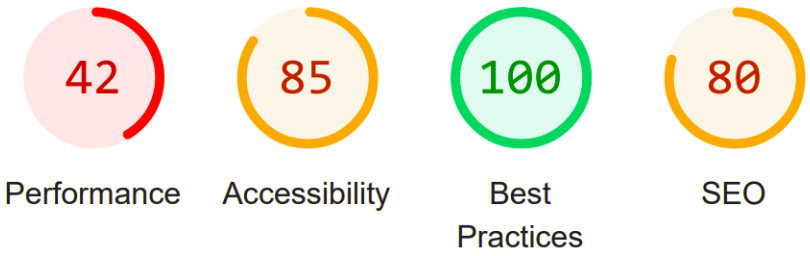
Dashboard Performance



The performance testing results for our dashboard and reports pages reveal some interesting insights. Both pages demonstrate varied performance across key metrics, all of which contribute to overall system performance in different ways.

For the dashboard, we see a direct performance score of 36, indicating room for optimization. Interestingly, the accessibility score of 89 suggests that the page's structure and design choices aimed at improving accessibility are having a positive impact on performance. The best practices score of 96 is particularly noteworthy, as following web development best practices often leads to more efficient code execution and resource utilization, boosting overall performance. The SEO score of 80 implies that our search engine optimization efforts are also contributing to a leaner, more performant page structure.

Reports Performance



Moving to the reports page, we observe a slightly higher direct performance score of 42. This improvement could be attributed to the enhanced accessibility score of 85, which likely indicates more efficient DOM structures and better resource loading. The perfect best practices score of 100 is especially impressive, suggesting that this page is leveraging cutting-edge performance optimization techniques. The SEO score remains consistent at 80, maintaining a performance-friendly page organization.

It's crucial to understand that while these metrics may seem distinct, they all interconnect to influence the overall performance of our pages. By focusing on improving these scores across the board, we can expect to see compounding benefits in loading times, responsiveness, and overall user experience.

Test Review and Sign-off

Conduct peer reviews of test cases and scripts Hold regular test results review meetings with stakeholders. Establish clear criteria for test pass/fail and release readiness Obtain formal sign-off from relevant stakeholders before major releases.