Software Testing & Quality Assurance Final Report (SauceDemo & DummyJSON)

Prepared by: Asma Khasawneh

Track: Software Testing – Quality Assurance

Date: February 2026

1.Executive Summary

This report presents a concise and valuable summary of the Quality Assurance activities performed as part of the final QA project. The project focused on testing a retail web application and its supporting backend APIs using a combination of manual testing, test automation, and performance testing techniques.

The web application, SauceDemo, was tested through comprehensive manual test cases and UI automation to validate critical user flows such as login, product browsing, cart management, and checkout. In parallel, the DummyJSON API was tested using API functional testing with Postman and performance testing using k6 to ensure the reliability, stability, and responsiveness of backend services under different load conditions.

The testing activities were conducted in a controlled and well-prepared environment where system access, test data, and requirements were clearly defined and available. This enabled the execution of effective and structured test cycles, resulting in a successful and well-covered testing process. The outcomes demonstrate that the system behaves as expected under both functional and non-functional testing scenarios.

Through this project, the essential role of Quality Assurance was clearly demonstrated. Regardless of how strong or innovative an idea or system may be, Quality Assurance remains a critical factor in delivering reliable, high-quality products. QA ensures that software is released with confidence, meets user expectations, and supports the overall success of any project or organization.

2. System Under Test + Assumptions

2.1 SUT:

This project was conducted on a web-based retail system and its supporting backend API to simulate a real-world e-commerce environment.

Web UI:
The web application under test is SauceDemo, which provides basic e-commerce features such as login, product browsing, cart management, and checkout. The application was used for manual testing and UI automation to validate core user flows.
Link**:** **https://www.saucedemo.com**

REST API:
The backend system under test is DummyJSON, a RESTful API used to test authentication, products, carts, and users. It was used for API functional testing with Postman and performance testing with k6.
Link: **https://dummyjson.com**

Both systems were tested using test data only and within the scope defined for this project.

2.2. Assumptions & Constraints

The following assumptions and constraints were considered during the testing activities of this project:

- All testing was performed using test data only. No real personal or payment information was used.
- The system requirements and test scope were provided and clearly defined before testing started.
- The DummyJSON API simulates create, update, and delete operations, and data changes may not persist on the server.
- Security testing was out of scope for this project.
- Performance testing was executed within reasonable load limits to avoid impacting the availability of the target system.
- Testing was conducted in a stable environment with consistent access to the system under test.

3. Test Strategy & Scope

This project followed a structured Quality Assurance approach that combines manual testing, test automation, and performance testing to validate both functional and non-functional aspects of the system.

The testing scope included:

- Manual functional testing of core user flows on the web application.
- UI automation testing to validate critical scenarios such as login, cart operations, and checkout.
- API functional testing using Postman to verify backend behavior, authentication, and data handling.

- API performance testing using k6 to evaluate system stability and response times under load.

The testing scope excluded:

- Security and penetration testing.
- Third-party integrations not directly related to the core system.
- Real payment processing or sensitive data validation.

Testing activities were executed based on defined test scenarios and supported by automated assertions and performance metrics to ensure reliable and repeatable results.

4. Manual Testing

4.1 Summary

Manual testing was conducted on the web application to validate core functional flows and user interactions. A comprehensive set of test cases was designed and executed to ensure adequate coverage of both positive and negative scenarios.

4.2 Test Coverage

The manual test cases covered the following functional areas:

- User authentication (valid and invalid login scenarios).
- Product listing and product details validation.
- Cart operations (add to cart, remove items, reset app state).
- Checkout process and form validation.
- User-specific behaviors (standard user and problem user scenarios).

4.3 Test Scenarios

A total of 59 manual test cases were designed and executed, covering:

- Happy path scenarios to validate normal system behavior.
- Negative scenarios to validate error handling and input validation.
- Edge cases related to checkout and cart functionality.

4.4 Test Execution Summary

- Total Test Cases Executed: 59
- Passed: Majority of test cases passed successfully
- Failed: Test cases associated with known system defects (documented separately in the defect log)

Execution evidence, including screenshots and recorded results, was captured and documented for all executed test cases to ensure traceability and validation.

The results of manual testing indicate that the system performs as expected in most core functional scenarios, while several defects were identified and reported to highlight areas requiring improvement.

5. Defects

5.1 Defects Summary

During the manual testing phase, multiple defects were identified while executing functional and negative test scenarios. All detected issues were documented in a defect log with clear reproduction steps, severity, priority, and supporting evidence.

5.2 Severity Distribution

A total of 8 defects were reported and categorized based on their impact on system functionality:

- High Severity: 3 defects
    - Issues that directly affect core user flows such as checkout or cart functionality.
- Medium Severity: 3 defects
    - Issues that impact usability or cause unexpected behavior but do not completely block the user.
- Low Severity: 2 defects
    - Minor issues related to UI behavior or non-critical validations.

This distribution highlights that most critical defects were concentrated around core shopping and checkout workflows.

5.3 Top Issues Identified

The most significant defects identified during testing include:

- Checkout process failing under specific input conditions.
- Inconsistent behavior when adding or removing products from the cart.
- UI inconsistencies when interacting with certain product-related actions.
- Validation messages not displayed correctly in some negative scenarios.

5.4 Defect Example

One of the reported defects **(BUG-03)** was related to the checkout process. When mandatory checkout fields were left empty, the system allowed the user to proceed without displaying proper validation messages. This behavior may lead to incomplete or invalid order data being submitted. This defect was classified as **High Severity** due to its direct impact on a critical user flow. Detailed reproduction steps and supporting screenshots were included in the defect log to ensure clear understanding and traceability.

6. API Testing (Postman):

6.1 Collection Design

The Postman collection was designed in a structured and organized manner to improve readability, maintainability, and ease of execution. The collection was created under the name **Final Project API Testing** and was divided into logical folders based on system functionality.

The collection consists of five main folders:

- **Authentication:**
  Contains all requests related to user authentication, including login and token validation.
- **Products (Big File):**
  This folder was further divided into two sub-folders:
  - **Products:** Includes requests for retrieving and searching products.
  - **CRUD:** Covers create, update, and delete operations related to products.
- **Carts (Big Project):**
  This folder was also divided into two sub-folders:
  - **Cart:** Includes requests related to cart retrieval and user cart operations.

    - **CRUD:** Covers create, update, and delete cart operations.
- **Categories:**
  Contains requests related to product categories and category-based retrieval.
- **Negative Scenarios:**
  Includes negative and validation test cases such as invalid IDs, missing authentication, wrong HTTP methods, and invalid payloads to verify proper error handling.

Each folder contains only the requests relevant to its specific functionality, ensuring a clear separation of concerns and a clean collection structure. This design supports easier execution, better traceability, and efficient automated testing using Postman and Newman.

6.2 Assertions & Automated Tests

Automated assertions were added to the Postman requests to validate the correctness and reliability of API responses. These assertions were designed to verify both functional behavior and response quality across positive and negative scenarios.

The following key validations were implemented:

- **Status Code Validation:**
  Verifying that each request returns the expected HTTP status code based on the scenario (e.g., success or failure cases).
- **Response Time Validation:**
  Ensuring that API responses are returned within an acceptable response time to confirm basic performance expectations.
- **Response Structure Validation:**
  Validating the structure and format of the response body, such as checking whether the returned data is an object or an array when required.
- **Critical Data Presence:**
  Verifying the existence of mandatory fields in the response, such as IDs, authentication tokens, and key attributes required for subsequent requests.
- **Negative Scenario Validation:**
  Confirming that appropriate error messages are returned for invalid requests, missing authentication, incorrect payloads, or invalid resource identifiers.

These assertions helped ensure that the APIs not only return correct responses but also handle invalid inputs properly, contributing to a more reliable and robust testing process.

6.3 Newman Report Summary

The Postman collection was executed using Newman to validate the automated API tests in a command-line environment. The execution covered **69 API requests** across **3 iterations**, resulting in a total of **429 automated assertions**.

The Newman execution results showed that all API requests were executed successfully, with **no failed requests** reported. A total of **18 assertions failed**, which were expected and related to predefined negative test scenarios designed to validate error handling and response validation behavior. No tests were skipped during execution.

Overall, the Newman results confirm the stability of the API execution flow and the correctness of the implemented assertions for both positive and negative scenarios. The generated Newman HTML report was used as execution evidence and included detailed request-level results.

7. Performance Testing (k6)

7.1 k6 Test Plan

The performance testing activities were conducted using k6 to evaluate the stability and responsiveness of the backend APIs under different load conditions. The test plan was designed based on realistic usage scenarios derived from the API functional testing phase.

The scope of performance testing included three core cart-related endpoints: retrieving all carts, retrieving a single cart using a dynamic cart ID, and retrieving carts by user ID. These endpoints were selected to represent common backend operations and user interactions.

Two load profiles were defined. A **Smoke Test** was executed using 5 virtual users for a short duration to validate basic system stability and ensure that the APIs were functioning correctly. This was followed by a **Load Test** using 30 virtual users for a longer duration to simulate normal expected system load. A start delay was applied to separate the smoke and load phases.

Dynamic data handling was implemented by randomly selecting a cart ID from the response of the carts list endpoint, ensuring realistic and non-static request execution. Think time was added between requests to simulate real user behavior.

Performance thresholds were defined to monitor key performance indicators, including response time percentiles, error rate, and check success rate. These thresholds ensured that response times remained within acceptable limits and that the system maintained reliable behavior under load.

7.2 Load Profiles

The performance tests were executed using two load profiles to evaluate system behavior under different usage levels.

- **Smoke Test:**
  The smoke test was executed using **5 virtual users** for a duration of **20 seconds**. This profile was used to verify basic system stability and ensure that the targeted API endpoints were functioning correctly under minimal load.
- **Load Test:**
  The load test was executed using **30 virtual users** for a duration of **40 seconds**, starting after a short delay following the smoke test. This profile simulated normal expected user traffic to evaluate system performance and response stability under sustained load

  Both load profiles were designed to reflect realistic usage patterns and were executed sequentially to ensure controlled and reliable performance evaluation.

7.3 Tables

Table 1: Test Execution Overview

Item	Value
Test Tool	k6
Execution Type	Local
Scenarios	1.Smoke Test 2. Load Test
Total Scenarios	2
Max Virtual Users	35 VUs
Total Test Duration	~1m 35s

Table 2: Load Profiles Summary

Profile	Virtual Users	Duration	Purpose
Smoke Test	5 VUs	20 seconds	Verify basic system stability
Load Test	30 VUs	40 seconds	Evaluate performance under normal load

Table 3: Thresholds Validation

Threshold	Expected	Actual Result	Status
Checks Success Rate	> 99%	100%	Passed
P95 Response Time	< 1000 ms	310.84 ms	Passed
Error Rate	< 1%	0.00%	Passed

Table 4: Key Performance Indicators (KPIs)

Metric	Result
Average Response Time	189.63 ms
Minimum Response Time	148.48 ms
Maximum Response Time	599.44 ms
P90 Response Time	220.85 ms
P95 Response Time	310.84 ms
Error Rate	0.00%
Total HTTP Requests	1170
Iterations Executed	390

Table 5: Validation Checks Summary

Validation	Result
Status Code Validation	Passed
Response Time Validation	Passed
JSON Response Validation	Passed
Carts Array Exists	Passed
Single Cart Retrieval	Passed
User Carts Retrieval	Passed

7.4 Results & Analysis

The k6 performance test results indicate that the backend APIs demonstrated stable and reliable performance under both smoke and load testing conditions. All defined thresholds were successfully met, with a 100% check success rate and zero request failures.

Response times remained well within acceptable limits, with an average response time of approximately 190 ms and a P95 response time of 310 ms, which is significantly below the defined threshold of 1000 ms. This indicates efficient request handling and consistent performance under the tested load.

No HTTP request failures were observed during execution, confirming the stability of the targeted endpoints. Overall, the performance testing results show that the system can handle the tested load levels effectively without performance degradation.

8. UI Automation

8.1Automation Approach

UI automation testing was implemented using Selenium WebDriver with Java as a Maven project and executed using the TestNG framework. The automation focused on end-to-end functional testing of critical user journeys rather than isolated UI checks.

The automation covered multiple user types, including standard users, performance users, blocked users, and negative login scenarios. A dedicated login test class was created to validate

authentication behavior for different user roles, with test data separated into a dedicated data package.

Following successful login validation, full end-to-end user flows were automated for different user types, including product sorting, adding products to the cart, removing items, and completing the checkout process. This approach ensured that core business workflows were validated from login until order completion.

A data-driven approach was applied by separating test data from test logic, allowing the same test flows to be reused across multiple user scenarios. This improved test coverage while keeping the automation maintainable and reusable.

Test execution was centralized using a TestNG test suite to ensure that all tests run in a controlled manner with a single browser session per execution.

8.2 Test Structure

The UI automation project was organized into two main packages:

- **Data Package:**
  Contains all test data classes, including login credentials and user-specific data for standard, problem, error, and performance users.
- **Test Package:**
  Contains test classes responsible for executing automation scenarios. This includes a dedicated login test class for authentication validation, as well as separate test classes for standard, problem, and error users covering full end-to-end user flows.

A base test class was implemented to manage WebDriver initialization and teardown. This class ensures that the browser is opened only once per test suite execution, preventing multiple browser instances from being launched unnecessarily.

TestNG was used as the main execution framework, with dependencies managed through Maven. A central **testng.xml** file was configured to define the test suite and control execution flow, allowing all tests to be executed in a single run using the TestNG runner.

8.3 How to Run the UI Automation Tests

The UI automation tests can be executed using two different methods, depending on the desired execution behavior.

Method 1: Running via TestNG Suite (Recommended)
The primary and recommended execution method is running the test suite through the testng.xml file. This can be done by right-clicking on the testng.xml file and selecting **Run As → TestNG Suite**.
In this approach, the test suite is executed in a controlled sequence, where a single WebDriver instance is initialized and reused across all test classes. This ensures efficient execution, avoids opening multiple browser instances, and allows all test classes to run sequentially as part of one test suite.

Method 2: Running Individual Test Packages or Classes
Alternatively, tests can be executed by right-clicking on the test package (e.g., ui.tests) or individual test classes and selecting **Run As → TestNG Test**.
In this approach, each test class is executed independently, and a new WebDriver instance is created for each class. This method is useful for debugging or running specific tests but is less efficient for full regression execution.

The test suite execution via testng.xml was used as the main execution approach for this project to ensure stability, consistency, and optimized browser usage.

8.4 sample results

- Video For the Run: *Execution evidence is provided in the Appendix section.*
- Results for the Run: *Execution evidence is provided in the Appendix section.*

The UI automation test suite was executed successfully using the TestNG framework. The execution validated all targeted end-to-end user scenarios, including login, product sorting, cart operations, and checkout flows for different user types.

Execution evidence was captured in the form of **screenshots,** and a **screen-recorded video** demonstrating the test run process. The recorded video shows the automated execution of the test suite and confirms that the tests were executed in sequence without manual intervention. Screenshots were also taken to highlight successful test execution and result outputs.

9. Risks, Limitations, and Recommendations

9.1 Risks

- The tested systems are demo and mock environments, which may not fully represent real production behavior.
- Changes to the demo application or API may affect test stability and result consistency.
- Performance test results may vary depending on network conditions and local machine resources.

9.2 Limitations

- Security and penetration testing were out of scope for this project.
- API write operations are simulated, and data changes may not persist on the server.
- Performance testing was limited to defined load levels and did not include stress or endurance testing.
- UI automation was executed in a local environment and not integrated into a continuous integration pipeline.

9.3 Recommendations

- Integrate UI automation into a CI pipeline for continuous validation.
- Expand performance testing to include stress and endurance scenarios.
- Increase negative and edge-case coverage for both API and UI testing.
- Enhance reporting by adding centralized test result dashboards.
- Consider adding security-focused testing in future testing phases.

10.Conclusion

This project delivered a comprehensive Quality Assurance testing portfolio covering manual testing, API testing, performance testing, and UI automation. The applied testing activities validated both functional and non-functional aspects of the system, ensuring reliable behavior across critical user flows.

Manual testing provided broad functional coverage and enabled the identification of key defects. API testing confirmed backend correctness and stability through automated assertions and Newman execution, while performance testing using k6 demonstrated that

the system performs reliably under defined load conditions. UI automation further strengthened test coverage by validating end-to-end user journeys using a structured and maintainable automation framework.

Overall, the project highlights the value of Quality Assurance in delivering high-quality software. Through structured testing, automation, and performance validation, the system was tested with confidence, risks were identified early, and actionable recommendations were provided to support continuous improvement and future scalability.

11. Appendix

Appendix A: Screenshots

- UI automation TestNG execution ClickToSeeTheResults

Appendix B: Reports

- Newman HTML report
- k6 performance execution output
- TestNG automation execution report

Appendix C: Links

- SauceDemo: https://www.saucedemo.com
- DummyJSON API: https://dummyjson.com
- UI Automation Execution Video (Google Drive): *(*ClickToRunTheVideo*)*