Al Hussein Technical University
Centre of Professional Development and Community Outreach
ICT Upskilling Program

Final Software Testing & Quality Assurance Project Report

Software Testing - Quality Assurance Track

Student Name: Yara Alzaghah

Student ID: 202508103

Instructor: Eng. Ashraf Al-Smadi

10th February 2026

Table of Contents

Abstract

This project presents a comprehensive Quality Assurance (QA) assessment of a web based e-commerce system and its associated APIs, conducted as part of a Software Testing and Quality Assurance capstone assignment. The objective of the project is to demonstrate a professional, industry aligned testing approach by combining manual testing, API functional automation, API performance testing, and UI automation into a single, coherent quality evaluation.

The testing activities covered functional correctness, validation and error handling, API reliability, performance under load, and regression stability through automation. Manual testing was used to validate end to end user workflows and identify functional defects. API testing was implemented using Postman and Newman to validate REST endpoints, assertions, and data driven execution. Performance testing was conducted using k6 to evaluate system behaviour under smoke and load conditions, focusing on key performance indicators such as response time, throughput, and failure rate. UI automation was implemented using Selenium WebDriver and TestNG to ensure regression stability.

Test results, defect analysis, performance metrics, and execution evidence are documented and analyzed to assess system quality and identify improvement areas. The project demonstrates a structured QA strategy, practical tool usage, and professional reporting aligned with real world software testing practices.

1. Introduction

Software quality is a critical factor in the success of modern digital systems, particularly for web-based applications and APIs that serve end users at scale. Defects, performance issues, or unstable releases can lead to financial loss, reputational damage, and poor user experience. As a result, structured and well documented testing processes are essential throughout the software development lifecycle.

This project focuses on applying professional Quality Assurance practices to a sample e-commerce application and its associated APIs. Rather than relying on a single testing technique, the project adopts a multi layered testing approach, combining manual testing, automation, API testing, and performance testing to achieve broad and deep coverage.

The scope of the work includes:

- Designing a formal test plan and test scenarios
- Executing and documenting manual functional testing
- Identifying and reporting defects with severity and priority
- Automating API testing using Postman and Newman
- Evaluating API performance using k6 under smoke and load conditions
- Implementing UI automation using Selenium WebDriver and TestNG
- Managing source code and collaboration artifacts through GitHub

The primary goal of this report is to document the testing strategy, implementation, results, and analysis in a clear, professional, and evidence-based manner. Each section of the report corresponds directly to executed work and includes supporting artifacts such as screenshots, logs, reports, and charts. Together, these sections provide a complete reflection of the testing effort and its outcomes.

2. System Under Test and Assumptions

2.1 System Under Test

The System Under Test (SUT) consists of two main components:

2.1.1 Web Application

- **Application Name:** SauceDemo
- **URL:** https://www.saucedemo.com
- **Application Type:** Web-based e-commerce demo application

The SauceDemo application simulates a typical online shopping platform and includes the following core features:

- User authentication (login and logout)
- Product inventory display
- Add/remove products from shopping cart
- Shopping cart validation
- Checkout process with form validation

This application was used primarily for manual testing and UI automation testing, focusing on user facing functionality and end to end workflows.

2.1.2 API System

- **API Name:** DummyJSON API
- **Base URL:** https://dummyjson.com
- **API Type:** Public REST (Mock API)

The DummyJSON API provides mock endpoints that simulate real world backend services commonly used in e-commerce systems.

During this project, functional API testing using Postman and Newman covered multiple endpoints, including authentication, user management, product operations, cart operations, and data driven scenarios. For performance testing using k6, a representative subset of critical endpoints was selected to simulate realistic e-commerce user behaviour and to keep load scenarios focused and controlled.

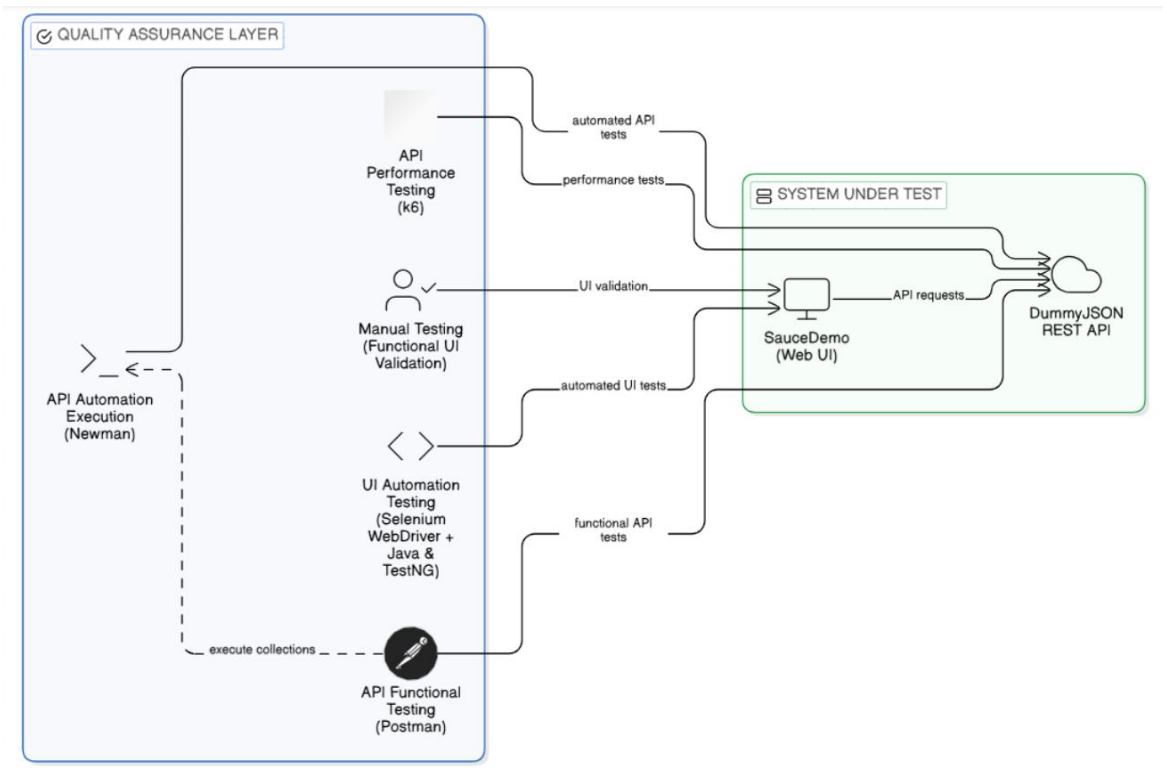


Figure 2.1 System Under Test and Quality Assurance Architecture

Figure 2.1 illustrates the relationship between the System Under Test and the different Quality Assurance layers applied during this project, including manual testing, API testing, automation, and performance testing.

2.2 Assumptions

The following assumptions were made during the testing activities:

- The SauceDemo application and DummyJSON API are externally hosted, and their internal implementations are not accessible.
- The application behaviour remains functionally stable during the testing period.
- Valid test credentials and test data are available and reusable.
- The DummyJSON API represents a mock environment, and its performance results do not fully reflect production grade backend systems.
- UI automation tests are executed on supported desktop browser (Microsoft Edge).
- Network connectivity during testing is stable and does not introduce significant external latency.
- Performance tests are executed locally and are constrained by local system resources.

These assumptions were documented to clearly define the boundaries and limitations of the testing results.

3. Test Strategy and Scope

3.1 Test Strategy

The test strategy for this project follows a risk based, layered testing approach, designed to validate system quality from multiple perspectives. Instead of relying on a single testing method, the strategy combines complementary testing techniques to maximize coverage and confidence.

The testing strategy includes:

- **Manual functional testing** to validate core business workflows and user interactions
- **API testing** to verify backend behaviour, and data integrity
- **API performance testing** to assess system responsiveness and stability under load
- **UI automation testing** to support regression testing and repeatability

Black box testing techniques were applied throughout the project, as the internal implementation of the systems under test was not accessible.

Testing Area	Tool	Justification
Manual Testing	Browser + Excel	Clear documentation, traceability, and defect tracking
UI Automation	Selenium WebDriver (Java) + TestNG	Industry standard automation with maintainable structure
API Testing	Postman	Efficient REST API testing and scripting
API Automation	Newman	Command line execution and detailed reporting
Performance Testing	k6	Script based, scalable, and lightweight load testing
Version Control	GitHub	Centralized test artifacts and automation management

Table 3.1 Testing Areas, Tools, and Selection Rationale

Table 3.1 summarizes the testing areas covered in this project, the tools selected for each area, and the justification behind each tool choice.

3.2 Testing Types and Levels

The following testing levels and types were applied:

Testing Levels

- **System Testing:** Validation of complete application functionality
- **End to End Testing:** Verification of full user journeys from login to logout

Testing Types

- Smoke Testing
- Functional Testing
- Negative Testing
- Regression Testing
- Performance Testing (Smoke and Load)

Each testing type served a specific purpose and collectively contributed to a holistic quality assessment.

3.3 Test Scope

In Scope

- Authentication workflows
- Product inventory and search
- Cart management
- Checkout and validation
- API functional correctness
- API performance metrics
- UI regression automation

Out of Scope

- Security and penetration testing
- Backend database validation
- Mobile application testing
- Advanced scalability and stress testing beyond defined limits

Defining scope ensured that testing efforts remained focused, realistic, and aligned with the project objectives and time constraints.

4. Manual Testing

4.1 Overview and Objectives

Manual testing was conducted to validate the functional correctness, usability, and validation behaviour of the SauceDemo web application from an end user perspective. This phase focused on verifying that core e-commerce workflows operate as expected when interacted with through the user interface, before introducing automation or performance testing.

The objectives of manual testing were to:

- Validate critical user journeys from login to logout
- Verify correct system behaviour for both happy path and negative scenarios
- Identify functional defects and validation issues
- Provide execution evidence to support test results
- Serve as a reliable baseline for later UI automation and regression testing

Manual testing was intentionally executed before automation to ensure that automated scripts were built on top of stable and well understood application behaviour.

4.2 Test Basis and Test Design

4.2.1 Test Basis

Manual test cases were derived from:

- The approved Test Plan
- Defined Happy and Negative Path Test Scenarios
- Functional requirements inferred from application behaviour

The test scenarios covered all major functional areas of the SauceDemo application, including authentication, product handling, cart management, checkout, and logout.

4.2.2 Test Design Techniques

The following test design techniques were applied during manual test case creation:

- **Black Box Testing**
  Internal implementation details were not considered; testing was based solely on inputs, outputs, and observable behaviour.

- **Equivalence Partitioning**
  Input fields (e.g., login credentials and checkout form fields) were tested using representative valid and invalid values.
- **Boundary and Validation Testing**
  Mandatory fields and validation rules were tested to ensure proper error handling and messaging.
- **Happy Path and Negative Path Coverage**
  Each major user flow was validated under both valid and invalid conditions to ensure system robustness.

These techniques ensured broad coverage while keeping the test suite focused and maintainable.

4.3 Test Scenarios and Coverage

High level test scenarios were defined to represent real user interactions with the system. These scenarios formed the foundation for detailed manual test cases.

The scenarios included:

- Login with valid and invalid credentials
- Product selection and cart operations
- Viewing and validating cart contents
- Checkout with valid information
- Checkout with missing or invalid input data
- Logout functionality

TS Number	Scenario Title	Description	Expected Result	Scenario Type
TS-01	Login with valid credentials	User logs in using valid username and password	User is successfully logged in and redirected to product page	Happy
TS-02	Login with invalid credentials	User attempts login using invalid credentials	Login is denied and error message is displayed	Negative
TS-03	Add product to cart	User adds a product from product list to cart	Product is added and visible in cart	Happy
TS-04	View shopping cart	User opens the cart after adding products	Cart displays correct products and quantities	Happy
TS-05	Checkout with valid information	User completes checkout using valid information	Checkout proceeds successfully	Happy

TS-06	Checkout with empty cart	User attempts checkout without adding products	System prevents checkout and shows validation message	Negative
TS-07	Checkout with missing first name	User submits checkout without first name	System prevents checkout and shows validation message	Negative
TS-08	Checkout with missing last name	User submits checkout without last name	System prevents checkout and shows validation message	Negative
TS-09	Checkout with missing postal code	User submits checkout without postal code	System prevents checkout and shows validation message	Negative
TS-10	Checkout with invalid first name	User submits checkout with invalid first name format	System prevents checkout and shows validation message	Negative
TS-11	Checkout with invalid last name	User submits checkout with invalid last name format	System prevents checkout and shows validation message	Negative
TS-12	Checkout with invalid postal code	User submits checkout with invalid postal code format	System prevents checkout and shows validation message	Negative
TS-13	Logout successfully	User logs out from the application	User is logged out and redirected to login page	Happy

Table 4.1 Manual Test Scenarios (Happy and Negative Paths)

Table 4.1 summarizes the high level manual test scenarios designed to cover both happy and negative user flows for the SauceDemo application.

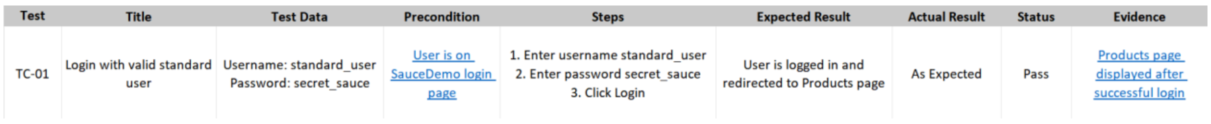
4.4 Manual Test Case Implementation

4.4.1 Test Case Structure

Manual test cases were documented in a structured Excel format to ensure clarity, traceability, and repeatability. Each test case included the following attributes:

- Test Case ID and Title
- Preconditions and Test Data
- Step by step execution steps
- Expected Result
- Actual Result
- Execution Status (Pass/Fail)
- Evidence reference

This structure aligns with industry standard manual testing practices and enables efficient defect identification and reporting.



Test	Title	Test Data	Precondition	Steps	Expected Result	Actual Result	Status	Evidence
TC-01	Login with valid standard user	Username: standard_user Password: secret_sauce	User is on SauceDemo login page	1. Enter username standard_user 2. Enter password secret_sauce 3. Click Login	User is logged in and redirected to Products page	As Expected	Pass	Products page displayed after successful login

Figure 4.1 Sample Manual Test Case Structure and Execution Result

Figure 4.1 illustrates a sample manual test case, including test data, execution steps, expected and actual results, execution status, and supporting evidence.

4.4.2 Test Execution

Manual test execution was performed using supported desktop browsers (Google Chrome and Microsoft Edge). Each test case was executed according to its documented steps, and results were recorded immediately to avoid discrepancies.

Test execution focused on:

- Verifying expected navigation and UI behaviour
- Confirming correct data display (products, cart contents)
- Validating form error messages and system feedback
- Ensuring state consistency across user actions

Both happy path and negative scenarios were executed to ensure that the system behaves correctly under valid usage as well as incorrect or incomplete input conditions.

4.5 Test Execution Summary

A total of 20+ manual test cases were executed, covering all critical functional areas of the application. The execution results provided a clear view of system stability and areas requiring improvement.

User Type	Total Test Cases Executed	Passed	Failed	Key Observations
Standard User	16	12	4	Core flows executed successfully
Problem User	7	1	6	Functional inconsistencies observed
Performance Glitch User	3	0	3	UI delays and unstable behaviour identified
Overall	26	13	13	Issues identified and logged as defects

Table 4.2 Manual Test Execution Summary

Table 4.2 provides a summary of manual test execution results across different user types, highlighting pass/fail distribution and key observations.

4.6 Defect Identification During Manual Testing

Defects identified during manual test execution were logged in a structured defect report. Each defect included:

- Clear reproduction steps
- Expected vs actual behaviour
- Severity and priority classification
- Environment details
- Supporting evidence

Defect severity was assigned based on functional impact and user experience, ensuring that critical issues were clearly distinguished from minor validation or UI related problems.

4.7 Key Observations

The following observations were made during manual test execution:

- Core user flows such as login, product browsing, and cart management worked as expected for the standard user.
- Testing with Problem User and Glitch User accounts revealed functional inconsistencies and unstable UI behaviour.
- Checkout validation was not consistently enforced, as the system allowed progression in some scenarios where invalid data was entered.
- Manual testing was critical in identifying these issues prior to automation and provided a reliable basis for defect reporting and further regression testing.

4.8 Conclusion

Manual testing successfully validated the functional behaviour of the SauceDemo web application and ensured that critical user workflows were tested under both valid and invalid conditions. The structured approach to test design, execution, and documentation enabled effective defect detection and traceability.

This phase played a crucial role in establishing confidence in the application's baseline quality and directly informed subsequent automation testing activities.

5. Defects

5.1 Overview

Defects were identified during the execution of manual test cases on the SauceDemo web application. All defects were logged in a structured defect report to ensure clear traceability between test cases, observed behaviour, and reported issues.

The purpose of defect reporting in this project was to:

- Document functional and validation related issues discovered during testing
- Assess the impact and severity of each issue
- Provide reproducible steps and evidence to support defect analysis
- Highlight risk areas that affect system reliability and user experience

Defect analysis in this section is based only on executed manual tests and their corresponding defect records.

5.2 Defect Reporting Process

Each identified defect was documented using a standardized defect reporting format. This ensured consistency and clarity across all reported issues.

Each defect record includes:

- Defect ID and Title
- Affected Module
- Preconditions and Test Data
- Steps to Reproduce
- Expected Result vs Actual Result
- Severity and Priority
- Execution Environment
- Supporting Evidence

This approach aligns with industry standard defect management practices and supports effective communication between QA and development stakeholders.

5.3 Defect Severity Classification

Defects were classified based on their impact on functionality and user experience, rather than frequency alone.

The following severity levels were applied:

- **High:** Issues that allow incorrect system behaviour or bypass critical validation (e.g., checkout validation failures)
- **Medium:** Functional inconsistencies that affect usability or correctness but do not fully block user progress
- **Low:** Minor UI or behaviour issues with limited functional impact

This classification helped prioritize issues and identify high risk areas within the application.

Severity Level	**Number of Defects**	**Percentage (%)**
High	4	30.8%
Medium	9	69.2%
Low	0	0%
Total	**13**	**100%**

Table 5.1 Defect Severity Distribution

Table 5.1 presents the distribution of reported defects by severity level, providing an overview of the overall risk profile identified during manual testing.

5.4 Defect Distribution by Functional Area

Analysis of the reported defects revealed that issues were not evenly distributed across the application. A higher concentration of defects was observed in specific functional areas.

The main affected areas included:

- Checkout form validation
- User type specific behaviour (Problem User and Glitch User)
- UI stability and consistency

This distribution highlights areas of higher functional risk and provides insight into where additional testing and stabilization would be required.

Functional Area	Number of Defects	Primary Impact
Checkout / User Information	4	Input validation failures and data integrity risks
Cart / Products	2	Inability to add or remove products
Products / UI	1	Incorrect product image rendering
Products / Sorting	1	Sorting functionality not applied
Product Details	1	Incorrect or mismatched product information
Authentication / Performance	1	Delayed login response
Product Details / Navigation / Performance	1	Delayed navigation response
Products / Sorting / Performance	1	Slow sorting response
Cart / Checkout	1	Checkout allowed with empty cart
Total	**13**	—

Table 5.2 Defect Distribution by Functional Area

Table 5.2 summarizes the distribution of identified defects across functional areas of the application, highlighting modules with higher defect concentration and potential quality risk.

5.5 Representative Defect Examples

To illustrate the nature and impact of the identified issues, selected representative defects are summarized below in **Table 5.3**. These examples reflect the most significant findings during manual testing.

Defect ID	Defect Title	Severity	Summary Description
Bug-01	Checkout allowed with empty cart	Medium	The system allows users to complete checkout even when no products are added to the cart, bypassing expected validation.
Bug-05	All products display the same image for problem user	Medium	Product images are incorrectly rendered as identical when logged in using the problem_user account.
Bug-06	Products cannot be added to cart for problem user	High	Certain products fail to be added to the cart when using the problem_user account, blocking core shopping functionality.

Defect ID	Defect Title	Severity	Summary Description
Bug-09	Incorrect product details displayed	High	Product details pages display mismatched or incorrect information for some products.
Bug-10	Last name field overwrites first name	High	Input entered in the last name field overwrites the first name field during checkout, causing data corruption.

Table 5.3 Sample Reported Defects

5.6 Key Defect Observations

The following observations were derived from defect analysis:

- Critical validation gaps were identified in the checkout process, allowing progression with invalid data.
- Application behaviour varied depending on user type, with Problem User and Glitch User exposing functional inconsistencies not present for the standard user.
- Several defects directly impacted user experience and data integrity, increasing the overall functional risk of the application.

5.7 Conclusion

The defect reporting and analysis phase provided valuable insight into the functional stability and risk areas of the SauceDemo application. While core flows functioned correctly for the standard user, defects related to validation logic revealed important quality gaps.

These findings emphasize the importance of structured manual testing and defect analysis as a foundation for automation, regression testing, and quality improvement recommendations discussed in later sections.

6. API Testing (Postman)

6.1 Overview and Objectives

API testing was conducted to validate the functional correctness, reliability, and response integrity of the DummyJSON REST API using Postman. While the SauceDemo web application was validated primarily through UI-based testing, API testing was used to independently verify backend style functionality and data behaviour through direct REST endpoint interaction.

The objectives of API testing were to:

- Verify that key endpoints return correct HTTP status codes and valid response structures
- Validate response content, required fields, and data types
- Confirm error handling and negative behaviour where applicable
- Improve repeatability by designing reusable collections and environments
- Enable automated execution and reporting through Newman

6.2 System Under Test

- **API Under Test:** DummyJSON Public REST API
- **Base URL:** https://dummyjson.com
- **Testing Tool:** Postman
- **Execution Support:** Newman CLI

The API test suite was structured to cover a broad set of functional areas including authentication workflows, product operations, user operations, cart operations, and data-driven scenarios.

The scope of API coverage is summarized below in **Table 6.1**

Category	Key Endpoints Tested	Purpose
Authentication	Login, Token Refresh	Access control and session handling
User Management	Get Current User, User Search	User data validation
Products	Product List, Product Search	Data retrieval and filtering
Cart Operations	Add Cart (Data-Driven), Get Cart	Transactional behaviour
Negative Scenarios	Invalid ID, Invalid Token	Error handling validation

Table 6.1 API Test Coverage Summary

6.3 Postman Collection Design

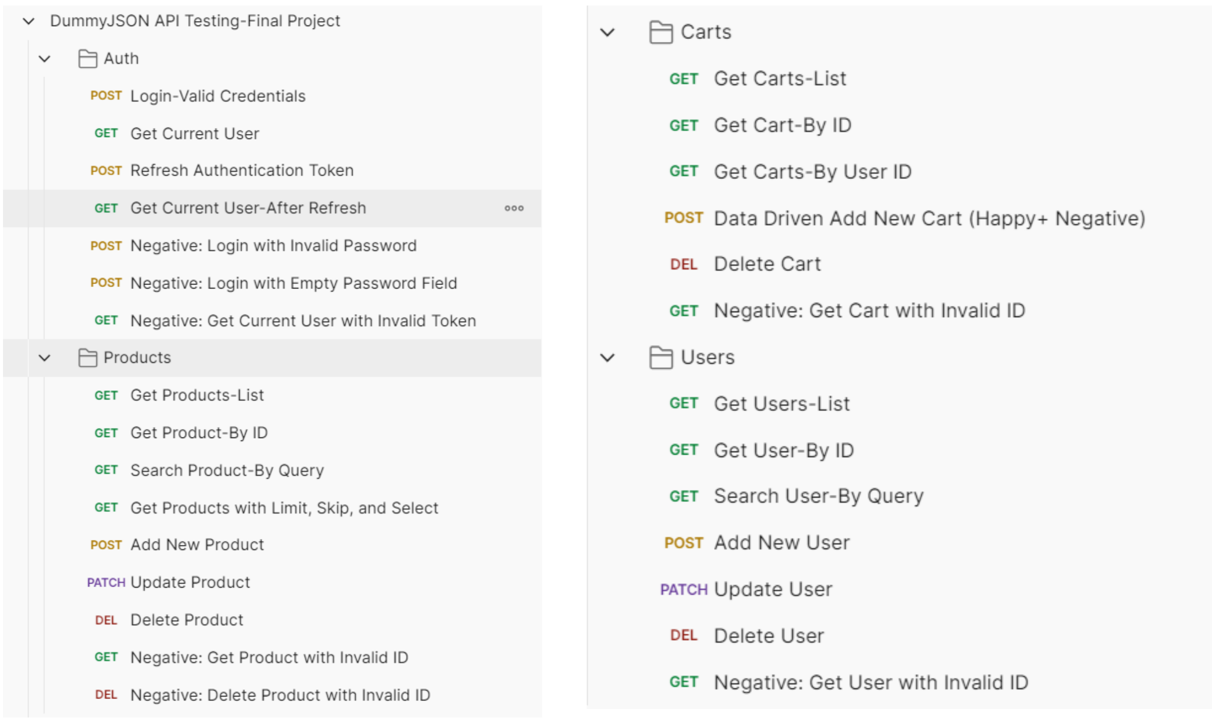
6.3.1 Collection Structure

The Postman test suite was organized as a structured collection to support maintainability, readability, and reuse. Requests were grouped by functional domain (for example: authentication, products, carts, users) with consistent naming conventions to reflect purpose and expected behaviour.

Each request includes:

- Clear request name and description
- Parameterized endpoint configuration using environment variables
- Post response test scripts for automated validation
- Consistent assertion patterns for correctness and reliability

The logical organization of the API tests is shown in **Figure 6.1**



- DummyJSON API Testing-Final Project
  - Auth
    - POST Login-Valid Credentials
    - GET Get Current User
    - POST Refresh Authentication Token
    - GET Get Current User-After Refresh
    - POST Negative: Login with Invalid Password
    - POST Negative: Login with Empty Password Field
    - GET Negative: Get Current User with Invalid Token
  - Products
    - GET Get Products-List
    - GET Get Product-By ID
    - GET Search Product-By Query
    - GET Get Products with Limit, Skip, and Select
    - POST Add New Product
    - PATCH Update Product
    - DEL Delete Product
    - GET Negative: Get Product with Invalid ID
    - DEL Negative: Delete Product with Invalid ID
  - Carts
    - GET Get Carts-List
    - GET Get Cart-By ID
    - GET Get Carts-By User ID
    - POST Data Driven Add New Cart (Happy+ Negative)
    - DEL Delete Cart
    - GET Negative: Get Cart with Invalid ID
  - Users
    - GET Get Users-List
    - GET Get User-By ID
    - GET Search User-By Query
    - POST Add New User
    - PATCH Update User
    - DEL Delete User
    - GET Negative: Get User with Invalid ID

Figure 6.1 Postman API Collection Structure

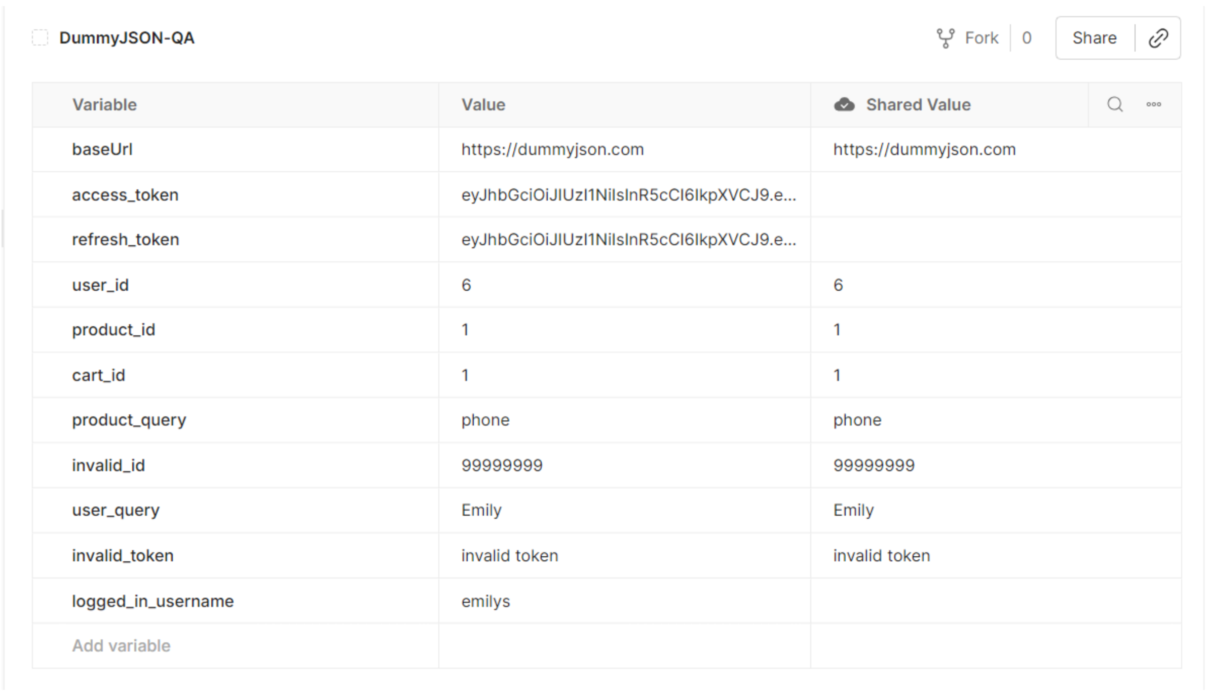
6.3.2 Environment Configuration and Reusability

A dedicated Postman environment was created to centralize configuration and enable consistent execution across different machines and runs. Key environment variables included:

- baseUrl
- IDs used for request parameterization (e.g., user, cart, product IDs)
- Authentication tokens (where applicable)
- Search inputs and negative test values

This approach improved maintainability by allowing test execution without editing request URLs or scripts.

Postman environment variables and there values is shown in **Figure 6.2**



Variable	Value	Shared Value
baseUrl	https://dummyjson.com	https://dummyjson.com
access_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.e...	
refresh_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.e...	
user_id	6	6
product_id	1	1
cart_id	1	1
product_query	phone	phone
invalid_id	99999999	99999999
user_query	Emily	Emily
invalid_token	invalid token	invalid token
logged_in_username	emilys	
Add variable		

Figure 6.2 Postman Environment Variables

6.4 Assertions and Validation Strategy

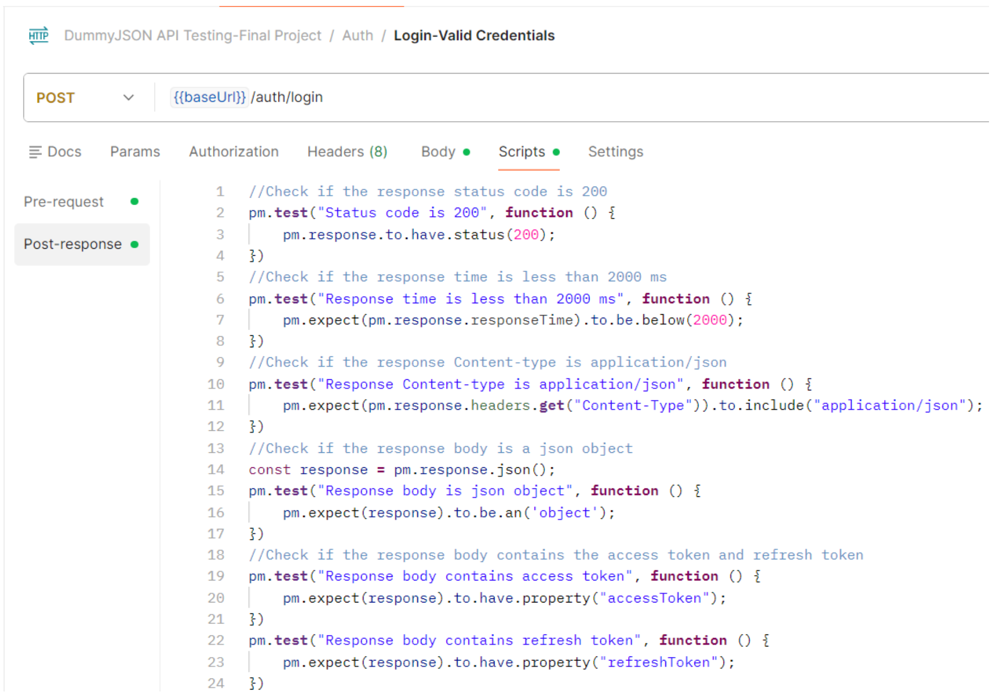
A core strength of this API testing phase is the use of systematic assertions to validate not only status codes, but also response content and structure.

Assertions implemented across the collection included:

- **Status code validation** (e.g., 200 OK, 201 Created)
- **Response time checks** (to ensure responsiveness under normal conditions)
- **Content-Type validation** (JSON response confirmation)
- **Schema/structure validation** (presence of expected objects/arrays)
- **Field level validation**
  - Required fields exist (e.g., id, title, price)
  - Correct data types are returned (number, string, array)
- **Data integrity checks**
  - Ensuring returned objects match expected request intent (e.g., search returns results)

This validation strategy ensured the API suite behaves as a true automated verification system rather than a simple request collection.

An example of automated API assertions is shown in **Figure 6.3**

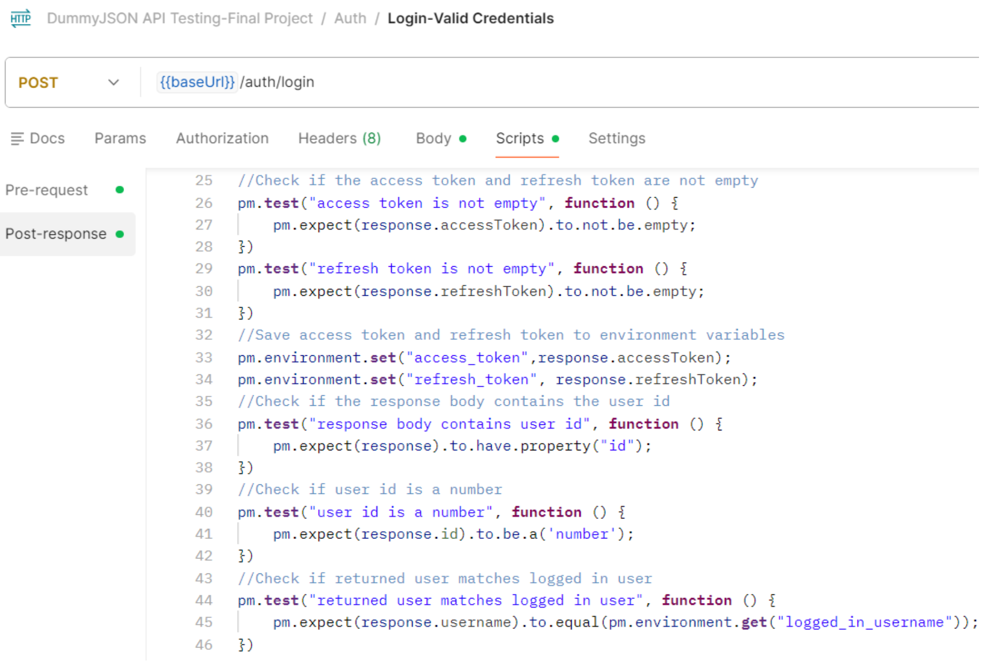

DummyJSON API Testing-Final Project / Auth / **Login-Valid Credentials**

POST {{baseUrl}} /auth/login

Docs · Params · Authorization · Headers (8) · Body · Scripts · Settings

Pre-request · Post-response

```
//Check if the response status code is 200
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
})
//Check if the response time is less than 2000 ms
pm.test("Response time is less than 2000 ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(2000);
})
//Check if the response Content-type is application/json
pm.test("Response Content-type is application/json", function () {
    pm.expect(pm.response.headers.get("Content-Type")).to.include("application/json");
})
//Check if the response body is a json object
const response = pm.response.json();
pm.test("Response body is json object", function () {
    pm.expect(response).to.be.an('object');
})
//Check if the response body contains the access token and refresh token
pm.test("Response body contains access token", function () {
    pm.expect(response).to.have.property("accessToken");
})
pm.test("Response body contains refresh token", function () {
    pm.expect(response).to.have.property("refreshToken");
})
```


DummyJSON API Testing-Final Project / Auth / **Login-Valid Credentials**

POST {{baseUrl}} /auth/login

Docs · Params · Authorization · Headers (8) · Body · Scripts · Settings

Pre-request · Post-response

```
//Check if the access token and refresh token are not empty
pm.test("access token is not empty", function () {
    pm.expect(response.accessToken).to.not.be.empty;
})
pm.test("refresh token is not empty", function () {
    pm.expect(response.refreshToken).to.not.be.empty;
})
//Save access token and refresh token to environment variables
pm.environment.set("access_token",response.accessToken);
pm.environment.set("refresh_token", response.refreshToken);
//Check if the response body contains the user id
pm.test("response body contains user id", function () {
    pm.expect(response).to.have.property("id");
})
//Check if user id is a number
pm.test("user id is a number", function () {
    pm.expect(response.id).to.be.a('number');
})
//Check if returned user matches logged in user
pm.test("returned user matches logged in user", function () {
    pm.expect(response.username).to.equal(pm.environment.get("logged_in_username"));
})
```

Figure 6.3 Automated Assertions Implemented in Postman

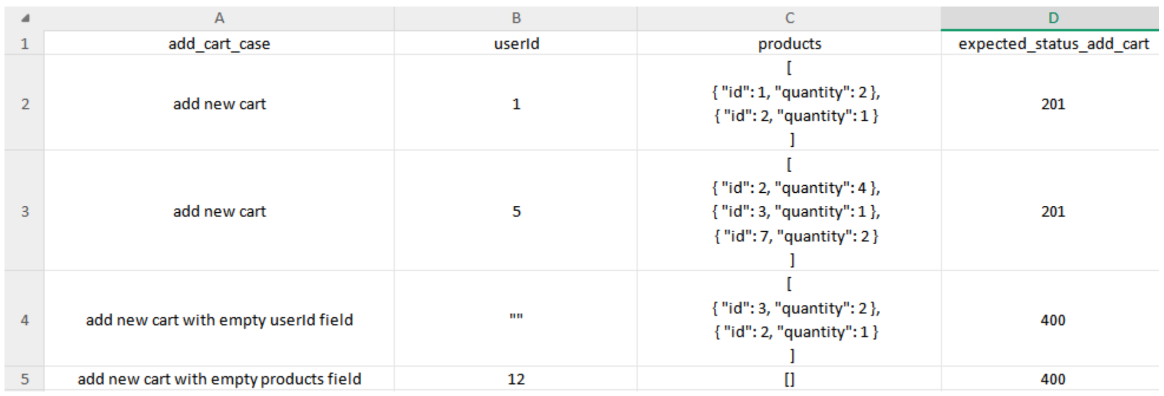
6.5 Data-Driven API Testing

To increase coverage and simulate varied input combinations, data-driven execution was implemented for a selected request using external CSV input. This enabled multiple iterations of the same request with different data values while maintaining a single test definition.

Key benefits of this approach:

- Improved coverage without duplicating requests
- Supported repeatable validation across multiple input sets

The data-driven inputs used during API execution are summarized in **Figure 6.4**



	A	B	C	D
1	add_cart_case	userId	products	expected_status_add_cart
2	add new cart	1	[{ "id": 1, "quantity": 2 }, { "id": 2, "quantity": 1 }]	201
3	add new cart	5	[{ "id": 2, "quantity": 4 }, { "id": 3, "quantity": 1 }, { "id": 7, "quantity": 2 }]	201
4	add new cart with empty userId field	""	[{ "id": 3, "quantity": 2 }, { "id": 2, "quantity": 1 }]	400
5	add new cart with empty products field	12	[]	400

Figure 6.4 Data-Driven Cart Test Parameters

6.6 Execution Outcome Summary

API functional execution results were validated through Postman runs and then automated with Newman for repeatable execution. Overall execution demonstrated stable API behaviour with successful assertions across multiple iterations.

6.7 Conclusion

API testing using Postman established functional confidence in the DummyJSON endpoints by verifying correct status codes, response structures, field integrity, and automated assertions. The collection was designed with reusability and maintainability in mind through the use of environment variables and structured request grouping. This work directly enabled automated execution using Newman and provided a foundation for performance testing using k6 in later sections.

6.8 API Automation Using Newman

To automate the execution of the Postman API test suite, Newman, the official command-line runner for Postman, was used. Newman enables repeatable, non-interactive execution of API collections and is commonly used in continuous integration and automated testing workflows.

The objectives of using Newman in this project were to:

- Execute the Postman API test collection in an automated manner
- Validate the stability and consistency of API behaviour across multiple runs
- Support data-driven execution using external CSV input
- Generate detailed execution reports as formal testing evidence

The same Postman collection and environment used during interactive API testing were reused during Newman execution, ensuring consistency between manual and automated API validation.

Execution was performed locally using the Newman CLI, with parameters specifying the collection file, environment file, data file, and report format.

6.9 Newman Execution Results and Reporting

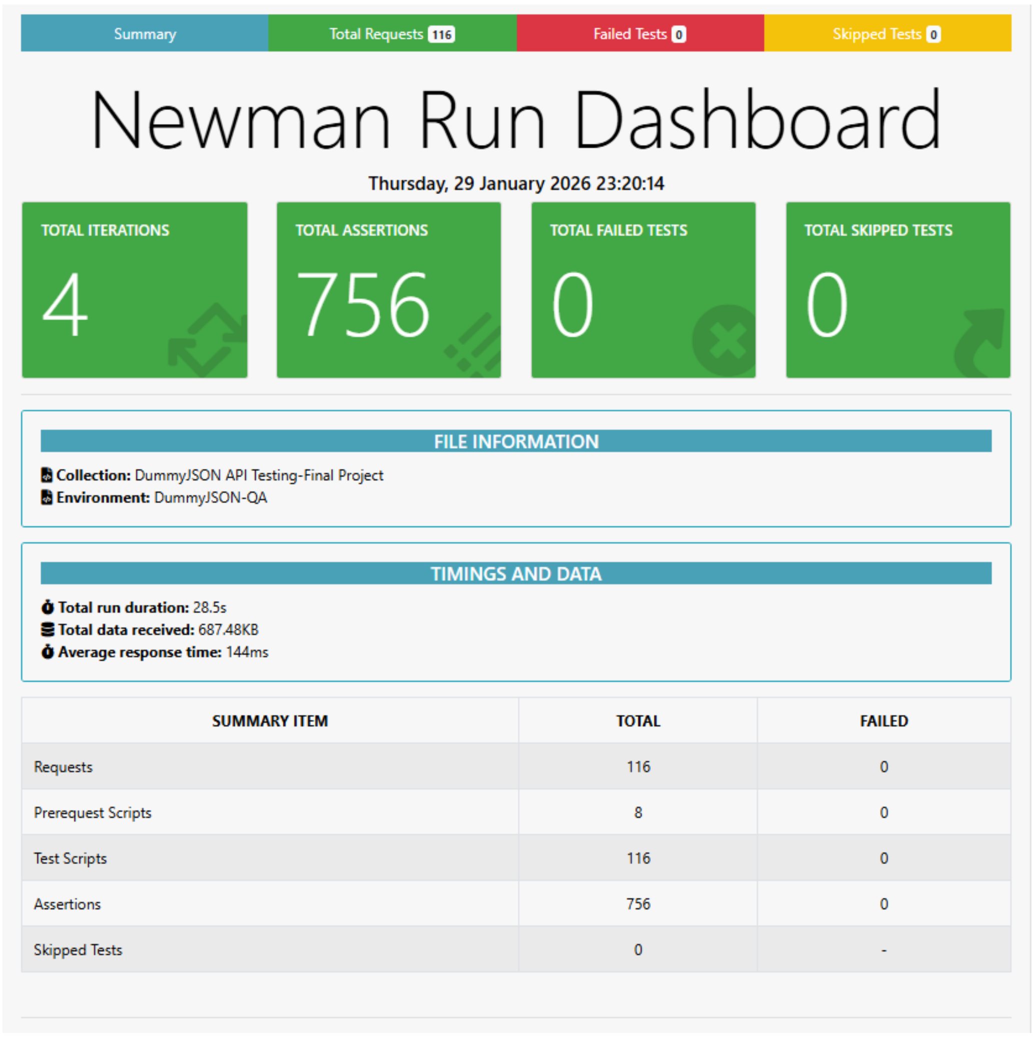
The API test collection was executed multiple times using Newman, including data-driven iterations. During these runs, all requests and assertions were executed automatically without manual intervention.

The execution results demonstrated:

- Successful execution of all API requests across all iterations
- Consistent pass results for all defined assertions
- Stable API behaviour across repeated automated runs
- No failed requests or assertion errors

To provide clear and professional evidence of execution, Newman reports were generated in HTML and PDF formats. These reports include detailed information such as request level results, assertion outcomes, response times, and overall execution summaries.

The overall automation results are visualized in **Figure 6.5**



Newman Run Dashboard

Thursday, 29 January 2026 23:20:14

TOTAL ITERATIONS 4

TOTAL ASSERTIONS 756

TOTAL FAILED TESTS 0

TOTAL SKIPPED TESTS 0

FILE INFORMATION

Collection: DummyJSON API Testing-Final Project
Environment: DummyJSON-QA

TIMINGS AND DATA

Total run duration: 28.5s
Total data received: 687.48KB
Average response time: 144ms

SUMMARY ITEM	TOTAL	FAILED
Requests	116	0
Prerequest Scripts	8	0
Test Scripts	116	0
Assertions	756	0
Skipped Tests	0	-

Figure 6.5 Newman HTML Execution Report Summary

7. Performance Testing (k6)

7.1 Overview and Objectives

Performance testing was conducted to evaluate the responsiveness, stability, and reliability of selected DummyJSON API endpoints under different load conditions. Unlike functional API testing, which focuses on correctness of responses, performance testing aims to assess how the system behaves when subjected to concurrent user activity.

The performance testing phase was implemented using k6, an open-source, script based load testing tool designed for modern APIs.

The objectives of this phase were to:

- Validate API availability and stability under minimal load (smoke testing)
- Evaluate system behaviour under increasing concurrent users (load testing)
- Measure key performance indicators (KPIs) such as response time, throughput, and error rate
- Identify potential performance risks or bottlenecks
- Provide quantitative evidence to support system performance analysis

7.2 Performance Test Scope

Performance testing focused on the DummyJSON REST API only. The SauceDemo web UI was excluded from this phase, as UI performance testing was outside the project scope.

A representative subset of API endpoints was selected for performance testing. These endpoints were chosen because they reflect common e-commerce operations and provide meaningful insight into system behaviour under load.

The scope included:

- Retrieve product list
- Search products by query
- Retrieve carts by user ID
- Add a new user

The subset approach ensured realistic performance testing while keeping load scenarios controlled and interpretable.

7.3 Test Environment and Tooling

- **Tool Used:** k6
- **Execution Mode:** Local execution
- **Test Type:** API level performance testing
- **Script Language:** JavaScript (k6 scripting model)

The k6 test scripts were generated by converting an existing Postman collection using the postman-to-k6 tool and then refined to include appropriate load profiles, thresholds, and performance checks.

This approach ensured consistency between functional API testing and performance testing.

7.4 Performance Test Design

Two distinct performance test types were implemented to evaluate the API under different conditions.

7.4.1 Smoke Test

The smoke test was designed to verify basic API availability and stability under a very small number of virtual users.

Characteristics:

- Constant and low number of virtual users
- Short execution duration
- Focus on detecting critical failures before load testing

This test served as a validation gate before executing more intensive load tests.

7.4.2 Load Test

The load test was designed to simulate increasing user activity and evaluate system behaviour under sustained load.

Characteristics:

- Gradual ramp-up of virtual users
- Sustained load period
- Controlled ramp-down phase
- Think time included to simulate realistic user behaviour

This test provided insight into how the API handles concurrent requests and whether performance degrades under load.

Test Type	Virtual Users (VUs)	Duration	Load Pattern	Purpose
Smoke Test	3	30 seconds	Constant	Validate basic API stability and correctness under minimal load
Load Test	0 → 50	~8 minutes	Ramping	Evaluate performance and scalability under increasing concurrent load

Table 7.1 Performance Test Types and Load Profiles

Table 7.1 outlines the performance test types executed in this project along with their corresponding load profiles and objectives.

7.5 Key Performance Indicators (KPIs) and Thresholds

To objectively evaluate performance results, specific KPIs and thresholds were defined within the k6 scripts.

The primary KPIs included:

- **Average response time**
- **95th percentile (p95) response time**
- **Request failure rate**
- **Throughput (requests per second)**
- **Check pass rate**

Thresholds were configured to ensure:

- Response times remain within acceptable limits
- Error rates remain minimal
- Functional checks continue to pass under load

KPI	Threshold	Purpose
Average Response Time	< 500 ms	Ensure acceptable average API responsiveness
95th Percentile (p95) Response Time	< 1000 ms	Control response time for most requests
HTTP Request Failure Rate	< 1%	Detect instability or request errors
Check Pass Rate	> 95%	Ensure functional correctness under load
Throughput	Observational	Measure API capacity under load

Table 7.2 Defined Performance KPIs and Thresholds

Table 7.2 summarizes the key performance indicators monitored during testing and the thresholds used to evaluate test success.

7.6 Test Execution Results

Performance tests were executed successfully for both smoke and load scenarios. Execution results were captured in structured summary files generated by k6.

7.6.1 Smoke Test Results

The smoke test confirmed that:

- All requests executed successfully
- No failed requests were observed
- Response times were stable under minimal load
- All defined checks passed

This validated the readiness of the system for load testing.

7.6.2 Load Test Results

During the load test:

- The API handled increasing concurrent users without request failures
- Response times remained within defined thresholds
- Throughput scaled consistently with user load
- No errors or instability were observed during peak load

Aspect	Smoke Test	Load Test
Virtual Users	3	Up to 50
Duration	30 seconds	~8 minutes
Goal	Functional validation	Performance & scalability
Avg Response Time	~196 ms	~126 ms
p(95) Response Time	~461 ms	~220 ms
Failure Rate	0%	0%
Result	Passed	Passed

Table 7.3 Smoke Test vs Load Test Results

Table 7.3 Compare the results of smoke test and load test

7.7 Results Analysis

Analysis of the performance results indicates that the tested DummyJSON API endpoints demonstrate stable and reliable behaviour under both minimal and moderate load conditions.

Key observations include:

- Consistent response times across smoke and load tests
- No increase in error rate under concurrent access
- Efficient handling of repeated API requests
- Predictable scaling behaviour under increased load

It is important to note that DummyJSON is a mock API, and therefore results do not represent production scale backend systems. However, the tests effectively demonstrate correct performance testing methodology and tool usage.

7.8 Limitations

The following limitations apply to the performance testing results:

- Tests were executed locally and constrained by local system resources
- The API under test is a mock service and not a production backend
- Stress and endurance testing were outside the defined scope
- Network conditions were not artificially varied

These limitations were acknowledged to ensure transparent interpretation of results.

7.9 Conclusion

Performance testing using k6 successfully validated the stability and responsiveness of selected DummyJSON API endpoints under smoke and load conditions. The use of script-based testing, defined thresholds, and quantitative KPIs enabled objective performance evaluation.

This phase complements functional API testing and demonstrates how performance testing can be integrated into a structured QA strategy. The results provide confidence in the tested endpoints while highlighting a repeatable methodology applicable to real world systems.

8. UI Automation Testing

8.1 Overview and Objectives

UI automation testing was implemented to validate core functional user flows of the SauceDemo web application in a repeatable and automated manner. This phase complements manual testing by enabling consistent re-execution of critical scenarios and demonstrating the use of industry-standard UI automation tools.

The main objectives of UI automation testing were to:

- Automate core functional scenarios validated during manual testing
- Verify application behaviour through the user interface
- Reduce reliance on repetitive manual execution
- Demonstrate structured UI automation using Selenium and TestNG

8.2 Automation Scope

UI automation focused on essential and stable user scenarios to ensure reliable execution and meaningful results. Negative scenarios and defect-prone user types (such as Problem User and Glitch User) were intentionally excluded from automation, as their unstable behaviour could lead to inconsistent or misleading automation results.

This scope selection aligns with common industry practices, where automation prioritizes high value regression scenarios.

8.3 Tools and Technologies

The following tools and technologies were used for UI automation:

- **Programming Language:** Java
- **IDE:** Eclipse
- **Test Framework:** Selenium WebDriver
- **Test Engine:** TestNG
- **Build and Dependency Management:** Maven

These tools were selected due to their wide adoption, flexibility, and suitability for functional UI automation.

8.4 Test Implementation Approach

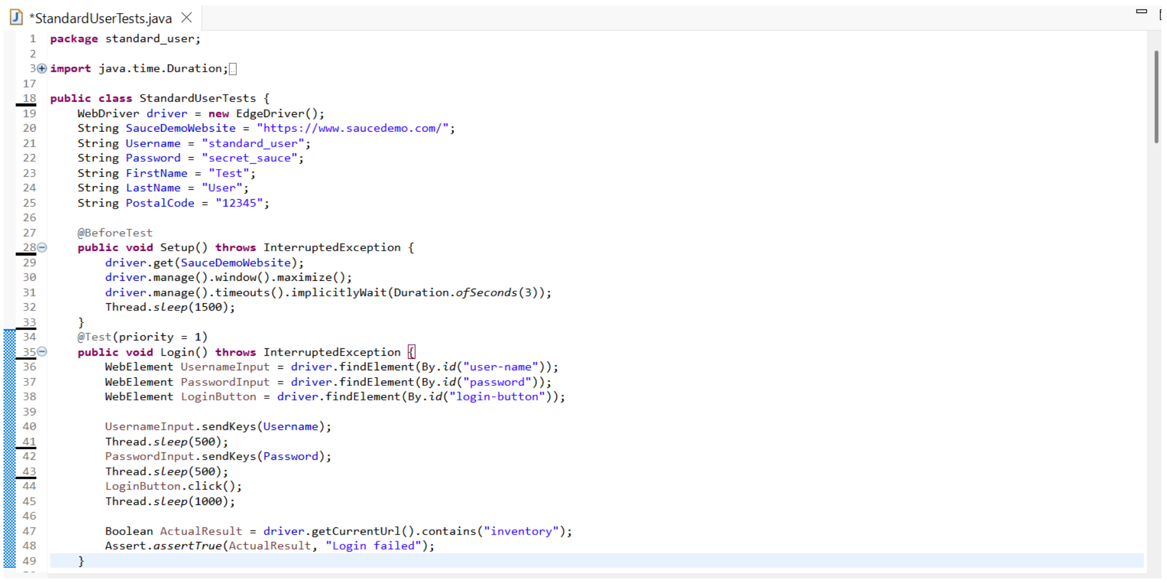
UI automation was implemented using TestNG based test classes, with Selenium WebDriver handling browser interaction. The test logic directly interacts with web elements by locating them and performing user actions such as input and button clicks.

Each automated test includes:

- Browser setup and initialization
- Navigation to the application under test
- Execution of user actions (login)

- Assertions to validate expected UI behaviour
- Browser cleanup after execution

The automation code is structured to ensure clarity and straightforward execution, making it suitable for small-scale regression testing.



*StandardUserTests.java

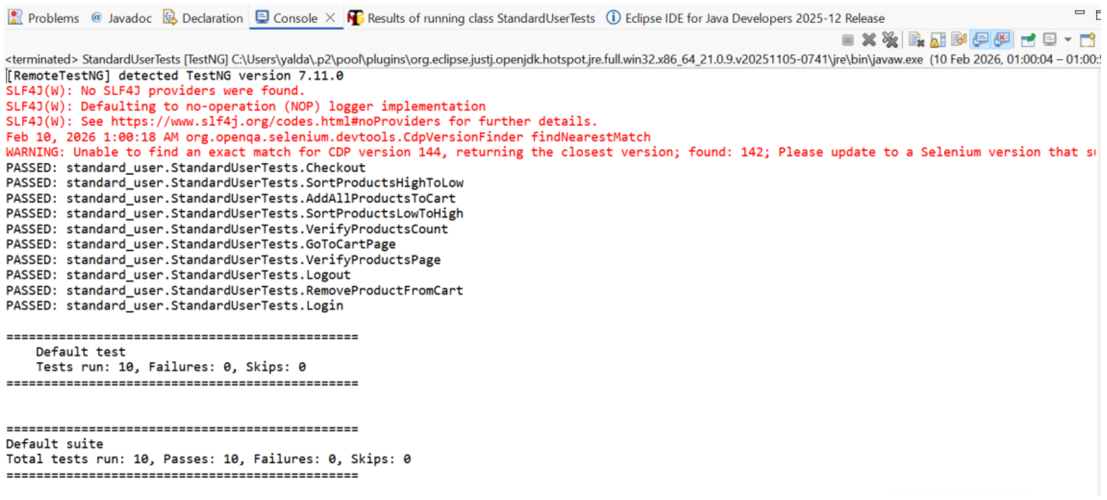
```java
package standard_user;

import java.time.Duration;

public class StandardUserTests {
    WebDriver driver = new EdgeDriver();
    String SauceDemoWebsite = "https://www.saucedemo.com/";
    String Username = "standard_user";
    String Password = "secret_sauce";
    String FirstName = "Test";
    String LastName = "User";
    String PostalCode = "12345";

    @BeforeTest
    public void Setup() throws InterruptedException {
        driver.get(SauceDemoWebsite);
        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(3));
        Thread.sleep(1500);
    }
    @Test(priority = 1)
    public void Login() throws InterruptedException {
        WebElement UsernameInput = driver.findElement(By.id("user-name"));
        WebElement PasswordInput = driver.findElement(By.id("password"));
        WebElement LoginButton = driver.findElement(By.id("login-button"));

        UsernameInput.sendKeys(Username);
        Thread.sleep(500);
        PasswordInput.sendKeys(Password);
        Thread.sleep(500);
        LoginButton.click();
        Thread.sleep(1000);

        Boolean ActualResult = driver.getCurrentUrl().contains("inventory");
        Assert.assertTrue(ActualResult, "Login failed");
    }
```

Figure 8.1 Sample UI Automation Test Implementation Using Selenium and TestNG

Figure 8.1 shows a sample UI automation test implementation written using Selenium WebDriver and TestNG, illustrating browser setup, user interaction, and assertion logic.

8.5 Test Execution

Test execution is managed using a TestNG suite configuration (testng.xml), which defines the execution of a single test class containing all automated test cases. Tests were executed locally and validated using TestNG execution output and reports.


Problems
Javadoc
Declaration
Console
Results of running class StandardUserTests
Eclipse IDE for Java Developers 2025-12 Release

```
<terminated> StandardUserTests [TestNG] C:\Users\yalda\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.9.v20251105-0741\jre\bin\javaw.exe (10 Feb 2026, 01:00:04 – 01:00:
[RemoteTestNG] detected TestNG version 7.11.0
SLF4J(W): No SLF4J providers were found.
SLF4J(W): Defaulting to no-operation (NOP) logger implementation
SLF4J(W): See https://www.slf4j.org/codes.html#noProviders for further details.
Feb 10, 2026 1:00:18 AM org.openqa.selenium.devtools.CdpVersionFinder findNearestMatch
WARNING: Unable to find an exact match for CDP version 144, returning the closest version; found: 142; Please update to a Selenium version that s
PASSED: standard_user.StandardUserTests.Checkout
PASSED: standard_user.StandardUserTests.SortProductsHighToLow
PASSED: standard_user.StandardUserTests.AddAllProductsToCart
PASSED: standard_user.StandardUserTests.SortProductsLowToHigh
PASSED: standard_user.StandardUserTests.VerifyProductsCount
PASSED: standard_user.StandardUserTests.GoToCartPage
PASSED: standard_user.StandardUserTests.VerifyProductsPage
PASSED: standard_user.StandardUserTests.Logout
PASSED: standard_user.StandardUserTests.RemoveProductFromCart
PASSED: standard_user.StandardUserTests.Login

===============================================
    Default test
    Tests run: 10, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 10, Passes: 10, Failures: 0, Skips: 0
===============================================
```

Figure 8.2 UI Automation Execution Results Using TestNG

Figure 8.2 presents the TestNG execution output for the automated UI tests, confirming successful execution of all defined test cases without failures.

8.6 Source Control

All UI automation source code and configuration files were stored in a GitHub repository, ensuring version control and traceability. This setup allows changes to automation code to be tracked and supports future extension of the test suite.

8.7 Limitations

The following limitations apply to the UI automation phase:

- Automation coverage is limited to a small set of core scenarios
- No abstraction layer (such as Page Object Model) was implemented
- Cross browser execution was limited

These limitations were acknowledged to ensure transparency and realistic evaluation of automation results.

8.8 Conclusion

UI automation testing successfully demonstrated the use of Selenium WebDriver and TestNG to automate a core functional scenario of the SauceDemo application. While the automation scope is limited, it provides a solid foundation for future expansion and illustrates how UI automation can support regression testing when combined with manual and API testing.

9. Risks, Limitations, and Recommendations

9.1 Risks and Limitations

The following risks and limitations were identified during the project:

- UI automation coverage is limited to a small set of stable scenarios and does not cover negative or defect-prone flows.
- API performance testing was conducted against a mock API (DummyJSON), which does not fully represent real production backend behaviour.
- Performance tests were executed locally and are constrained by local system resources and network conditions.
- Security, penetration, and cross-browser performance testing were outside the project scope.

These limitations were acknowledged to ensure transparent interpretation of results.

9.2 Recommendations

Based on the testing results, the following recommendations are proposed:

- Improve input validation during the checkout process to prevent incorrect data submission.
- Extend UI automation coverage to additional core flows once identified defects are resolved.
- Apply performance testing to a production-like backend environment for more realistic scalability assessment.
- Integrate automated tests into a CI pipeline to support continuous quality validation.

10. Appendix

The appendix contains supporting artifacts and execution evidence referenced throughout this report, including:

- Manual test cases and test scenarios
- Defect reports and screenshots
- Postman collections and environment files
- Newman execution commands and reports
- k6 scripts, summaries, and performance reports
- UI automation source code, TestNG configuration, and execution outputs
- GitHub repository links for version-controlled artifacts

All appendix materials provide traceable evidence for the testing activities and results documented in this report.