**PAIM TESTING REPORT**

**GitHub URL :** <https://github.com/Giridharan-Sankaran/PAIMProject.git>

**SUMMARY:**

This project aims to comprehensively test and automate the PetStore Swagger API, utilizing manual and automated testing methods, integrating with CI/CD tools, and implementing containerization. The PetStore API was chosen for its diverse endpoints and comprehensive documentation, allowing for thorough testing within a limited timeframe.

**INTRODUCTION:**

**Objectives:**

* Manually test the PetStore API using Postman and Swagger UI.
* Automate API tests using SoapUI.
* Integrate automated tests with Jenkins for continuous testing and reporting.
* Containerize SoapUI tests using Docker for portability and consistency.

**Scope:**

* Testing all available endpoints of the PetStore Swagger API.
* Implementing CRUD operations testing.
* Validating response structures and schemas.
* Verifying error handling and status codes.
* Performance testing under simulated high traffic.

**TEST PLAN:**

**Selected API:**

The PetStore Swagger API (<https://petstore.swagger.io/>) was selected for this project due to its comprehensive set of endpoints and well-documented API specifications. This API provides a robust platform for demonstrating various testing techniques and automation capabilities.

**Test Approach:**

**Manual Testing:**

* Utilize Postman for initial API exploration and validation.
* Use Swagger UI for interactive testing and documentation verification.

**Automated Testing:**

* Develop automated test suites using SoapUI.
* Implement test cases for all CRUD operations.
* Validate response structures and error handling.

**CI/CD Integration:**

* Integrate SoapUI tests with Jenkins for automated test execution.
* Configure Jenkins to generate test reports and logs.

**Containerization:**

* Dockerize SoapUI tests for consistent execution across environments.

**AUTOMATION FRAMEWORK:**

**Tools and Technologies used:**

* **Postman:** For manual API testing and initial test case development.
* **Swagger UI:** For API documentation and interactive testing.
* **SoapUI:** Primary automation tool for API testing.
* **Jenkins:** CI/CD tool for test execution and reporting.
* **Docker:** Containerization of SoapUI tests.

**Framework Architecture:**

The automation framework for the Petstore Swagger API is built using SoapUI and is designed to be modular and scalable. Here are the key components:

1. **Test Suite Organization:**

* The framework is divided into three main test suites: Pet Operations, Store Operations, and User Operations, each grouping related endpoints.

1. **Test Case Design:**

* Each test suite contains test cases for CRUD operations, response structure and schema validation, error handling, and edge cases.

1. **Load Testing:**

* Load tests are integrated into each test case to evaluate API performance under various loads.

**CI/CD Integration:**

Jenkins is configured to automate the SoapUI test execution process:

* Created a freestyle project in Jenkins
* Set up Git for source code management, linking to the GitHub repository
* Configured GitHub hook trigger for automatic builds on code changes
* Added build steps:
  1. Invoke Ant to generate SoapUI reports
  2. Execute Windows batch command for Docker operations and test suite execution
* Configured post-build action to publish HTML reports

This setup ensures automated testing and reporting whenever changes are pushed to the repository, facilitating continuous integration and delivery

**TEST CASES:**

**CRUD Operations:**

A comprehensive set of test cases has been developed to cover both positive and negative workflows for all CRUD (Create, Read, Update, Delete) operations in the Petstore API. These test cases include:

* Creating new pets, orders, and users with valid data
* Retrieving existing records using various parameters
* Updating pet information, order status, and user details
* Deleting pets, orders, and user accounts
* Handling edge cases such as invalid IDs or incomplete data

The complete test plan, detailing all scenarios, expected results, and actual outcomes, is available in the Excel sheet attached to the GitHub repository. This thorough approach ensures robust testing of the API's core functionalities across all endpoints.

**Performance Testing:**

Load tests have been implemented for selected API endpoints to evaluate their performance under stress. The configuration for these tests includes:

* Thread count: 50
* Duration: 300 seconds
* Strategy: Simple

Assertions were added to verify the API's ability to handle the simulated load. These values were chosen as they represent a nominal load for API testing.

Key aspects of the performance tests:

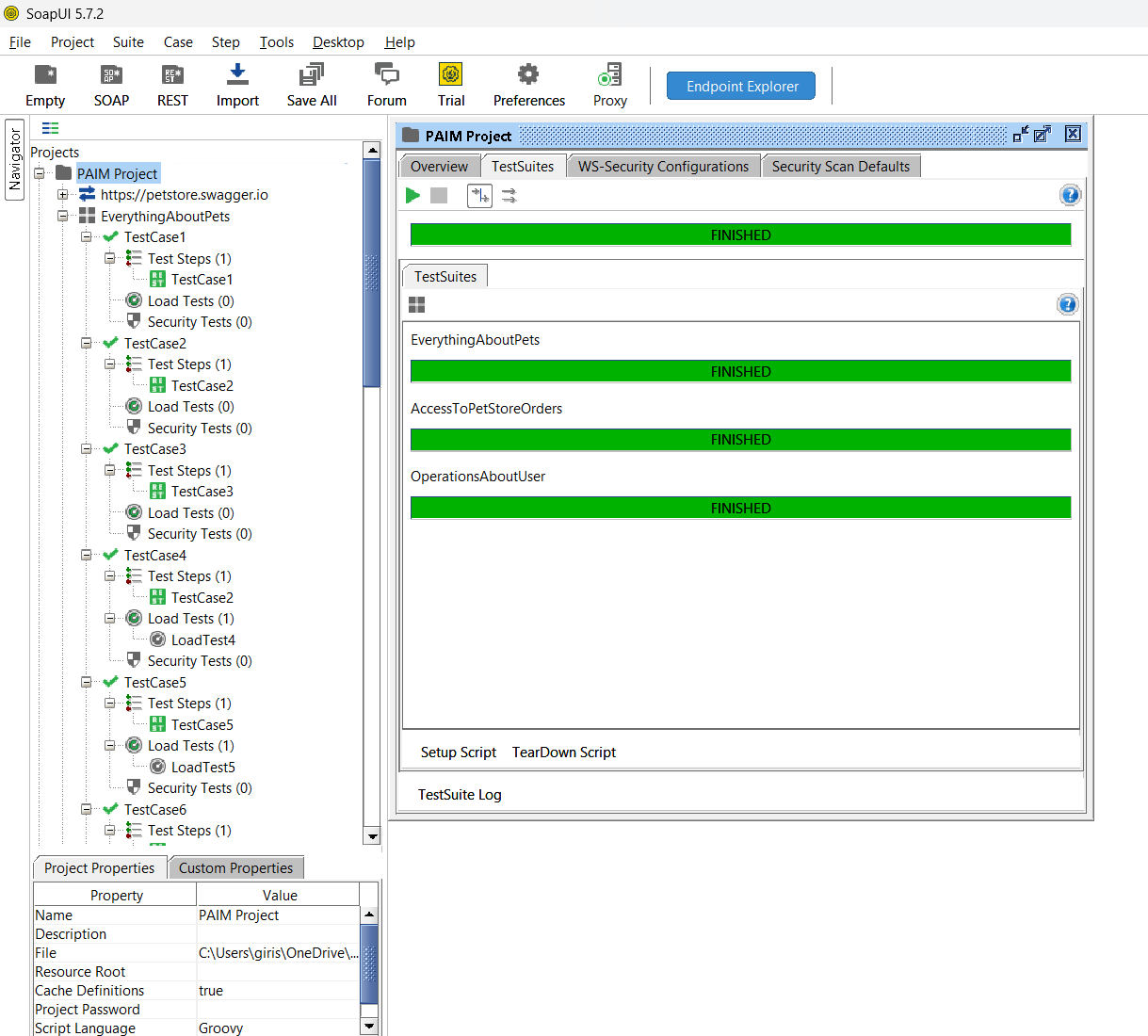
1. Concurrent users: The 50 threads simulate 50 simultaneous users accessing the API.
2. Test duration: The 300-second duration allows for observing the API's behavior over an extended period.
3. Simple strategy: This approach maintains a constant load throughout the test.
4. Assertions: These validate that the API maintains acceptable performance under load, checking metrics such as response time and error rates.

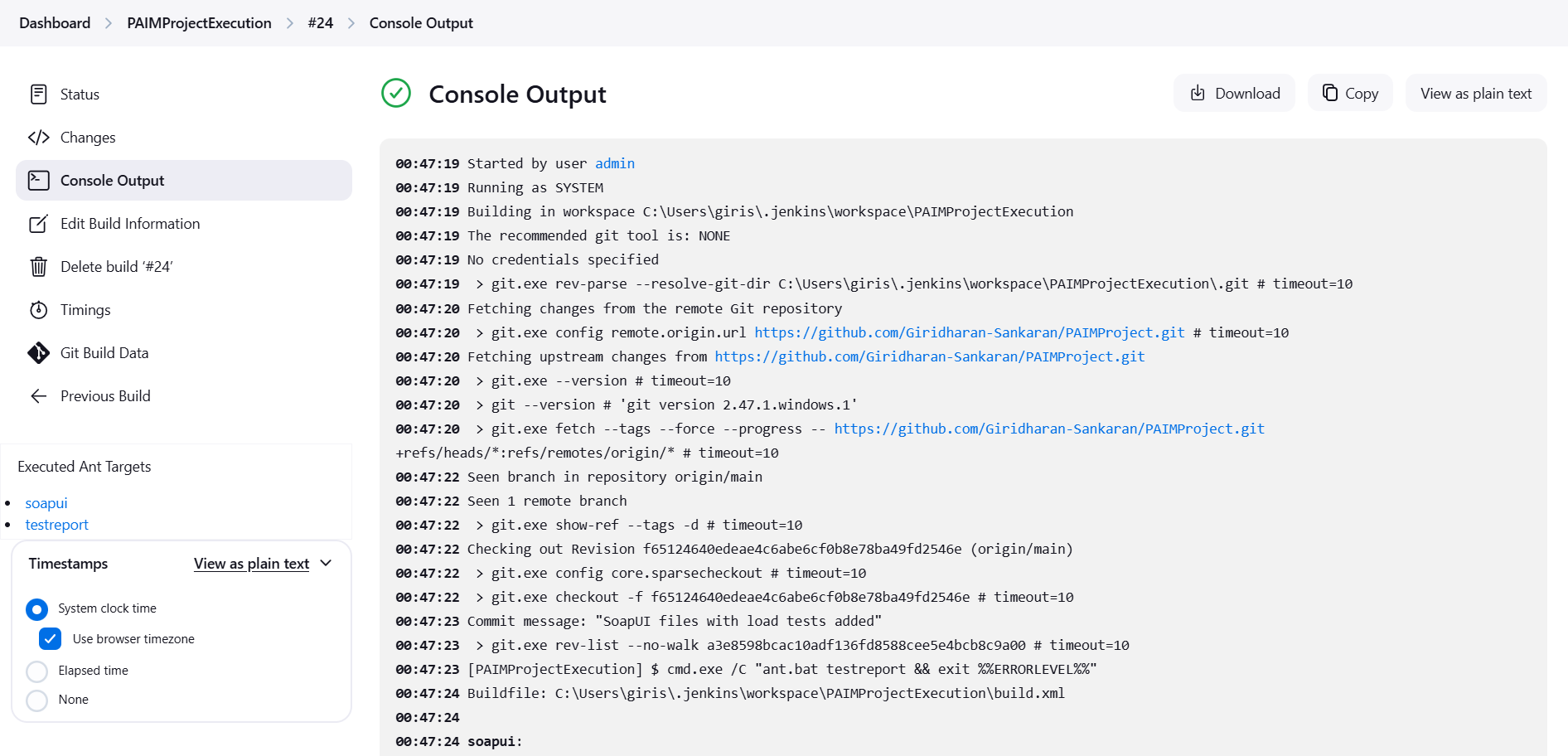
This configuration provides a baseline for assessing the API's performance characteristics and identifying potential bottlenecks or scalability issues.

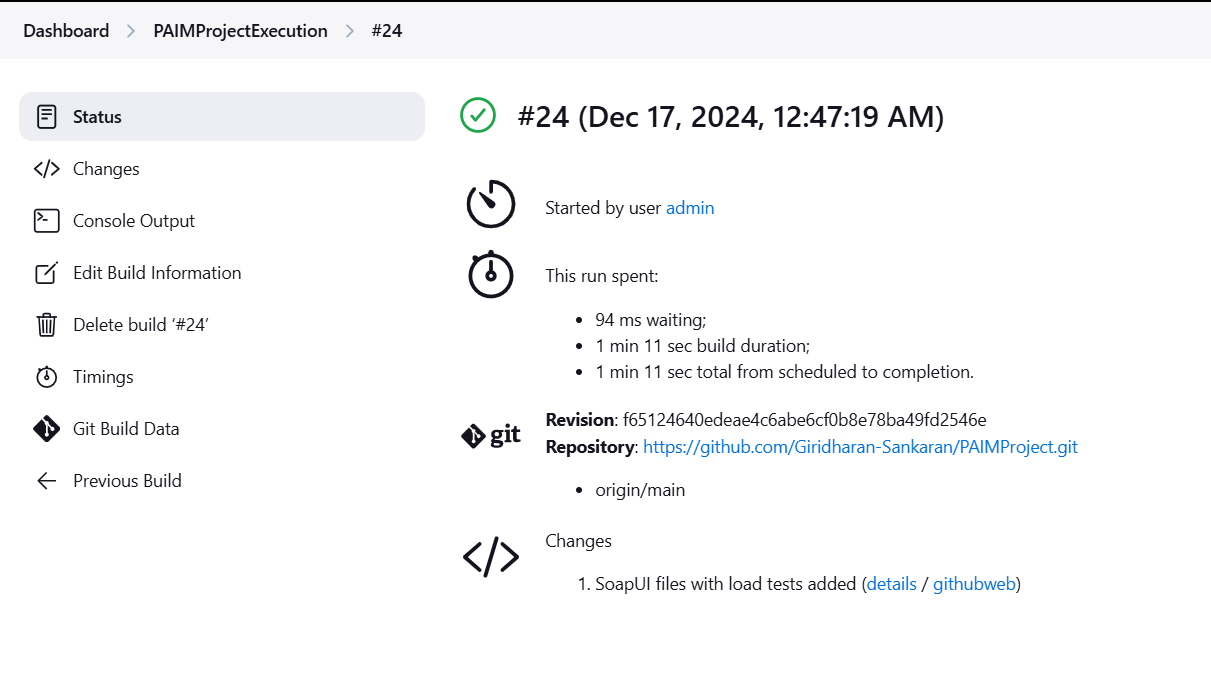
**TEST RESULTS:**

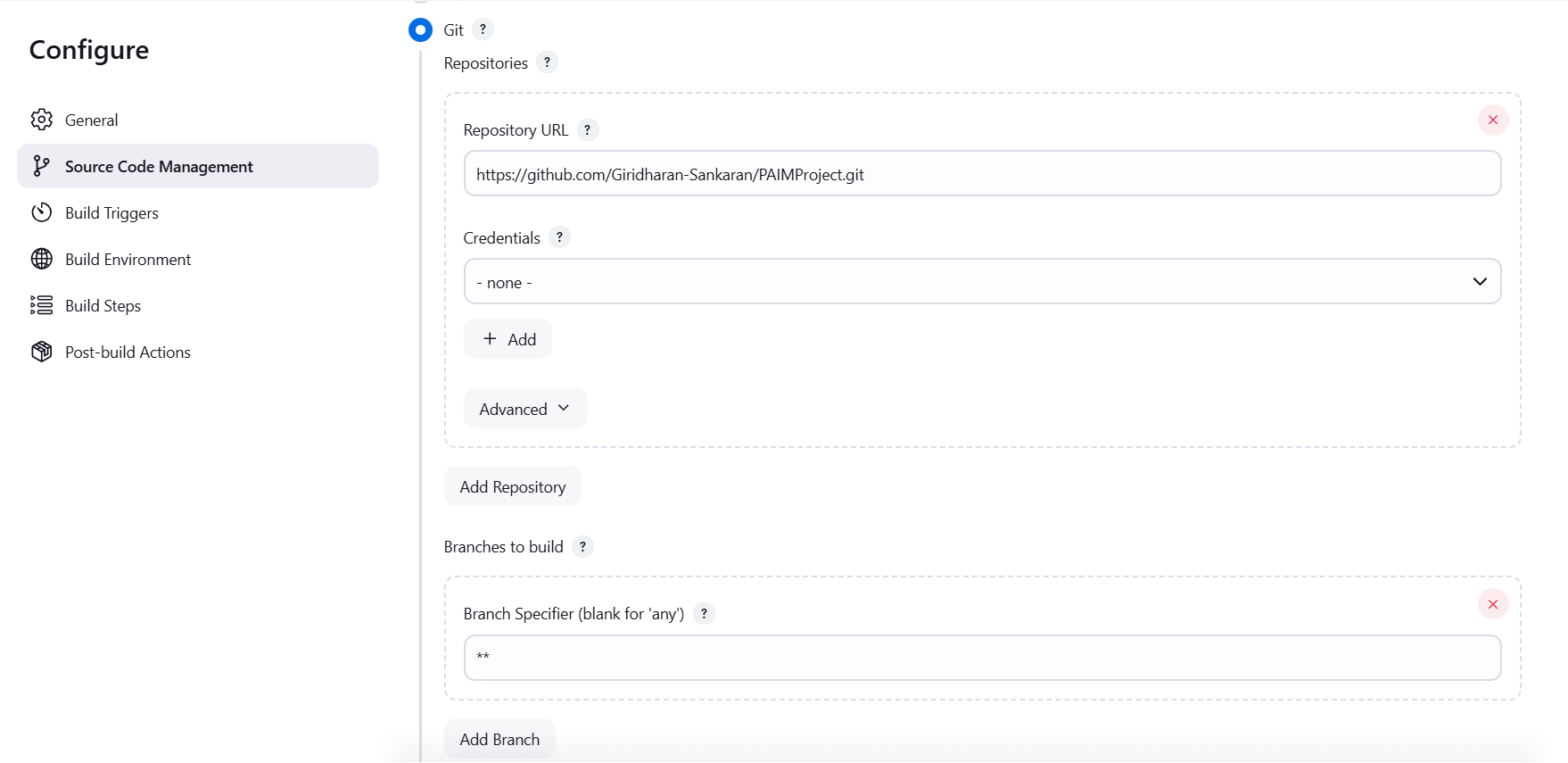
The test results have been documented comprehensively to provide a clear overview of the API's performance across different testing methods:

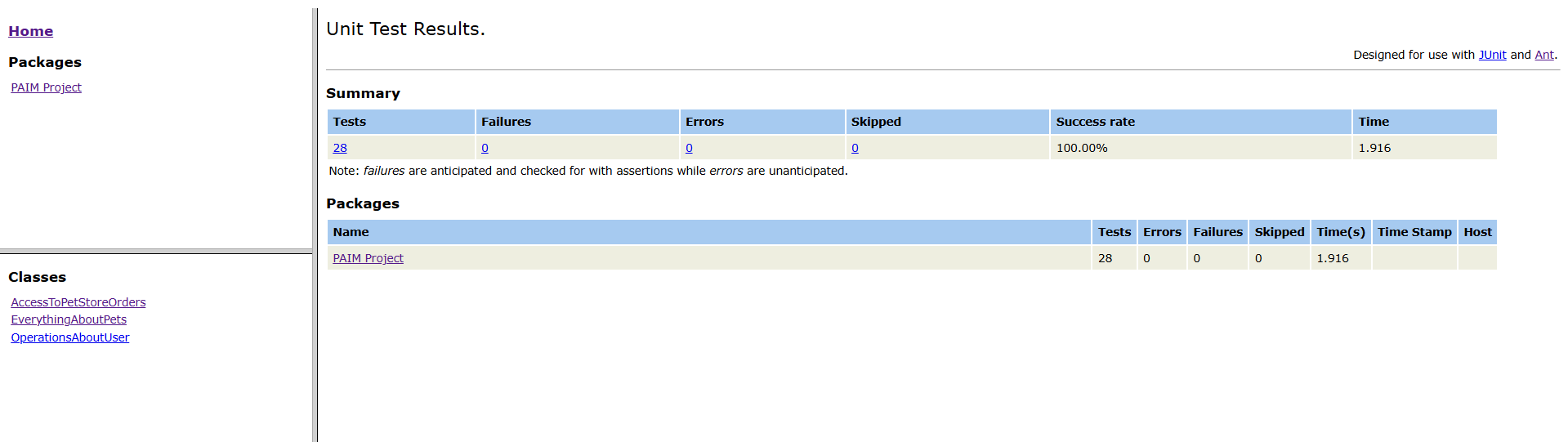
1. Manual Testing:
   * Screenshots of API tests conducted using Postman and Swagger UI are included in the test results document.
   * This document has been uploaded to the GitHub repository for reference.
2. Automated Testing:
   * Screenshots of SoapUI test executions are attached below in this report.
   * These images showcase the results of automated test cases, including CRUD operations and performance tests.

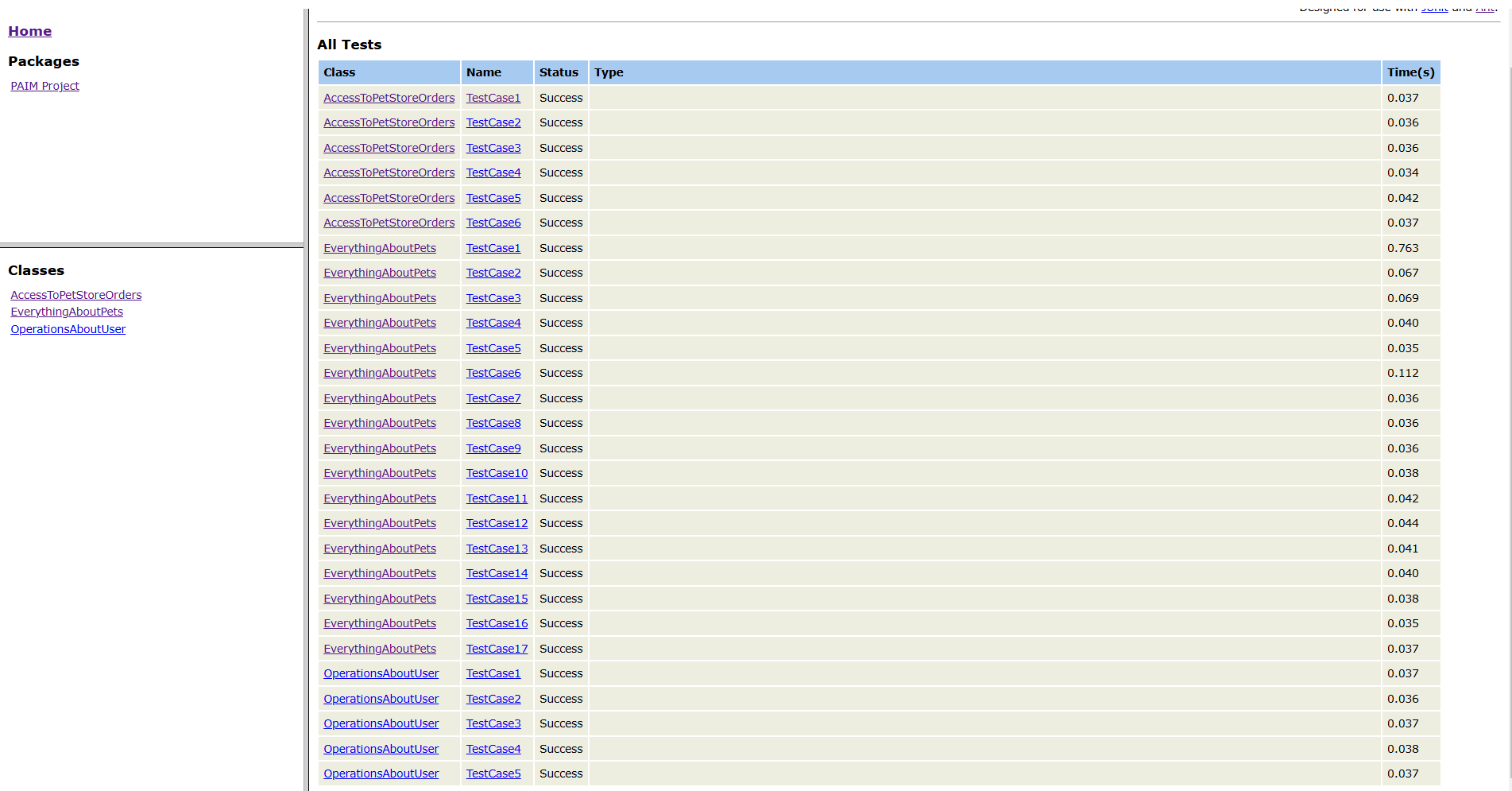












**OBSERVATIONS AND FINDINGS:**

During the testing process, several issues were identified that required attention. Here's a summary of the key findings:

1. Order ID Inconsistency
   * Issue: When placing an order without explicitly providing an ID, the system auto-generates one. However, subsequent GET requests fail for these auto-generated IDs.
   * Observation: The error message suggests IDs should be less than 10, conflicting with the auto-generation logic.
2. Unexpected User Creation on PUT Request
   * Issue: PUT requests for non-existent usernames create new records instead of returning an error.
   * Impact: This behavior could lead to unintended data creation and potential security risks.
3. Incorrect Status Code for Repeated DELETE Requests
   * Issue: Attempting to delete an already deleted username returns a 200 OK status instead of an appropriate error code.
   * Concern: This may mislead clients about the success of their delete operation.
4. Login Authentication Vulnerability
   * Issue: The login functionality accepts incorrect email IDs and passwords, still granting access.
   * Security Risk: This significantly compromises the system's security and user data protection.

These findings highlight areas where the API's behavior deviates from expected standards, potentially affecting data integrity, security, and user experience. Addressing these issues should be prioritized to enhance the overall reliability and security of the API.

In the test plan document attached to the GitHub repository, the test cases associated with these identified issues have been highlighted in red to draw attention to the problematic areas. This color-coding provides a quick visual reference for the development team to focus on the most critical issues that require immediate attention.

**DOCKERIZATION:**

To execute the SoapUI test suites within a Docker container, the following steps were implemented in the Jenkins configuration:

1. Run the SoapUI Docker Container: A Docker container was created to host the SoapUI test runner, mapping the project directory for access to test files.
2. Execute Test Suites: Curl commands were used to trigger the execution of specific test suites defined in the SoapUI project file.
3. Clean Up: After executing the tests, the Docker container was stopped and removed to free up resources.

The primary challenge encountered was generating HTML reports, as SoapUI does not natively support this feature. To address this, ngrok was utilized along with Ant framework configurations in the Jenkins file to facilitate report generation and access.

This approach ensures that all test suites in the project are executed in a consistent and portable manner within a containerized environment.

**CONCLUSION:**

This project successfully developed a robust testing framework for the PetStore Swagger API, achieving key objectives in both manual and automated testing. Comprehensive API testing validated all CRUD operations and response handling across multiple endpoints, while a modular SoapUI framework facilitated efficient, repeatable testing. Load tests assessed the API's scalability and reliability under stress, and Jenkins integration enabled continuous testing and reporting. Dockerization improved portability and consistency in test execution.

Throughout the process, several critical issues were identified, including order ID inconsistencies and security vulnerabilities, highlighting the importance of thorough testing. To enhance the API's reliability and security, it is essential to address these issues, expand performance testing scenarios, and improve reporting mechanisms.

In summary, this project laid a solid foundation for ongoing quality assurance of the PetStore API, ensuring its functionality and performance are maintained moving forward.