Janeeb
Solutions

TESTING POLICY DOCUMENT

Extended Planning Instrument for Unpredictable Spaces and Environments

University Of Pretoria
janeeb.solutions@gmail.com

Introduction

Testing is a crucial part of ensuring the quality, performance, and reliability of the logistics optimization system. This system, designed to improve the placement of goods in logistics trucks, leverages machine learning and dynamic algorithms. The following guide provides an overview of the testing procedures, including unit tests using Vitest and integration tests using Postman API.

Testing Overview

The logistics optimization system uses a combination of layered architecture, API Gateway, and service-oriented architecture (SOA). The backend is developed using TypeScript with Supabase for database interactions, while the frontend is built with Vue and Three.js for visualization.

The two primary types of testing performed in this system are:

- **Unit Tests**: Ensure the accuracy of isolated functions, algorithms, and components.
- **Integration Tests**: Ensure the proper interaction between system components, especially API functionality and data flow.

Unit Testing

Setup

Unit tests verify the correctness of individual components such as algorithms, database queries, and business logic. For unit testing, we use Vitest.

Ensure the following packages are installed:

```
npm install vitest --save-dev
npm install @types/node --save-dev
```

Writing Unit Tests

1. Create a test file in the `__tests__` directory, with a `_test.ts` suffix. Example: `packing_algorithm_test.ts`.
2. Use Vitest to mock external services and verify that components like the dynamic packing algorithm work correctly.

Example test with Vitest:

```
import { describe, expect, test } from 'vitest';
import { calculateOptimalPacking } from '../src/packing_algorithm';

describe('Packing Algorithm', () => {
  test('calculates optimal packing correctly', () => {
    const input = {/* mock input data */};
    const result = calculateOptimalPacking(input);
    expect(result).toEqual(expectedOutput);
  });
});
```

Running Unit Tests

To run the unit tests using Vitest, use the following command:

```
npx vitest
```

Integration Testing

Setup

Integration tests ensure the proper functioning of multiple components together, focusing on APIs and workflows. These tests are executed using Postman, which tests API routes by simulating real-world scenarios.

1. Create a collection in Postman that includes all relevant API endpoints (e.g., `POST /pack-items`, `GET /status`).
2. Set up different test cases for each endpoint, testing both normal and edge cases for requests and responses.

Writing Integration Tests

- In Postman, create requests for each API endpoint.
- Use the built-in "Tests" tab in Postman to write assertions. For example, verify response codes, data formats, and values.

Example in Postman:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response contains correct packing results", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property('packedItems');
    pm.expect(jsonData.packedItems).to.be.an('array');
});
```

Running Integration Tests

- Run individual tests or collections using Postman's interface or via the **Newman CLI** (Postman's command-line companion) to automate test execution.

To run using Newman:

newman run <your_postman_collection.json>

Integration Tests for Performance, Scalability, and Security

In addition to the standard integration tests, we conducted specific tests to ensure the logistics optimization system meets key quality requirements: **performance**, **scalability**, and **security**. These tests focused on evaluating the system's behavior under high loads and its resilience to unauthorized access attempts.

Performance Testing

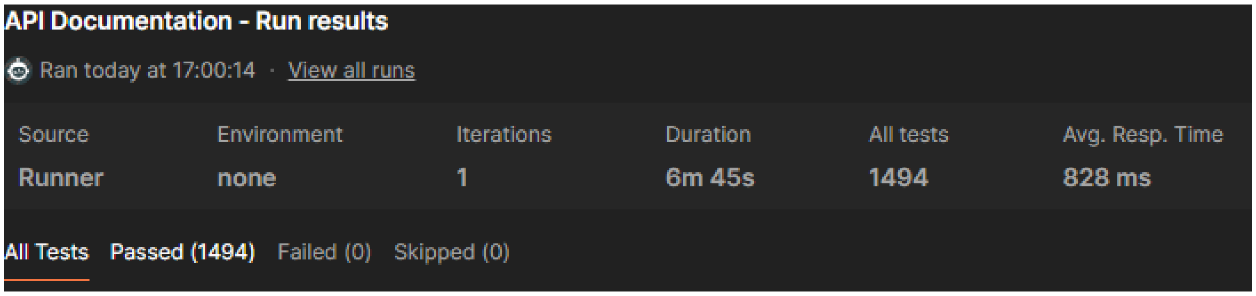
To assess the performance of the system, we performed a **stress test** that simulated **200 concurrent users** accessing the system. The primary objective was to observe the API's response times and identify any potential bottlenecks. The test involved multiple API routes, with a large volume of data being processed by the packing algorithm.

- **Test Scenario:** Simulate 200 concurrent users sending requests to the system.
- **Expected Outcome:** The system should handle the load efficiently, with minimal degradation in response time.
- **Result:** API response times averaged below the threshold of 2000ms under load, with no critical errors or failures.

Scalability Testing

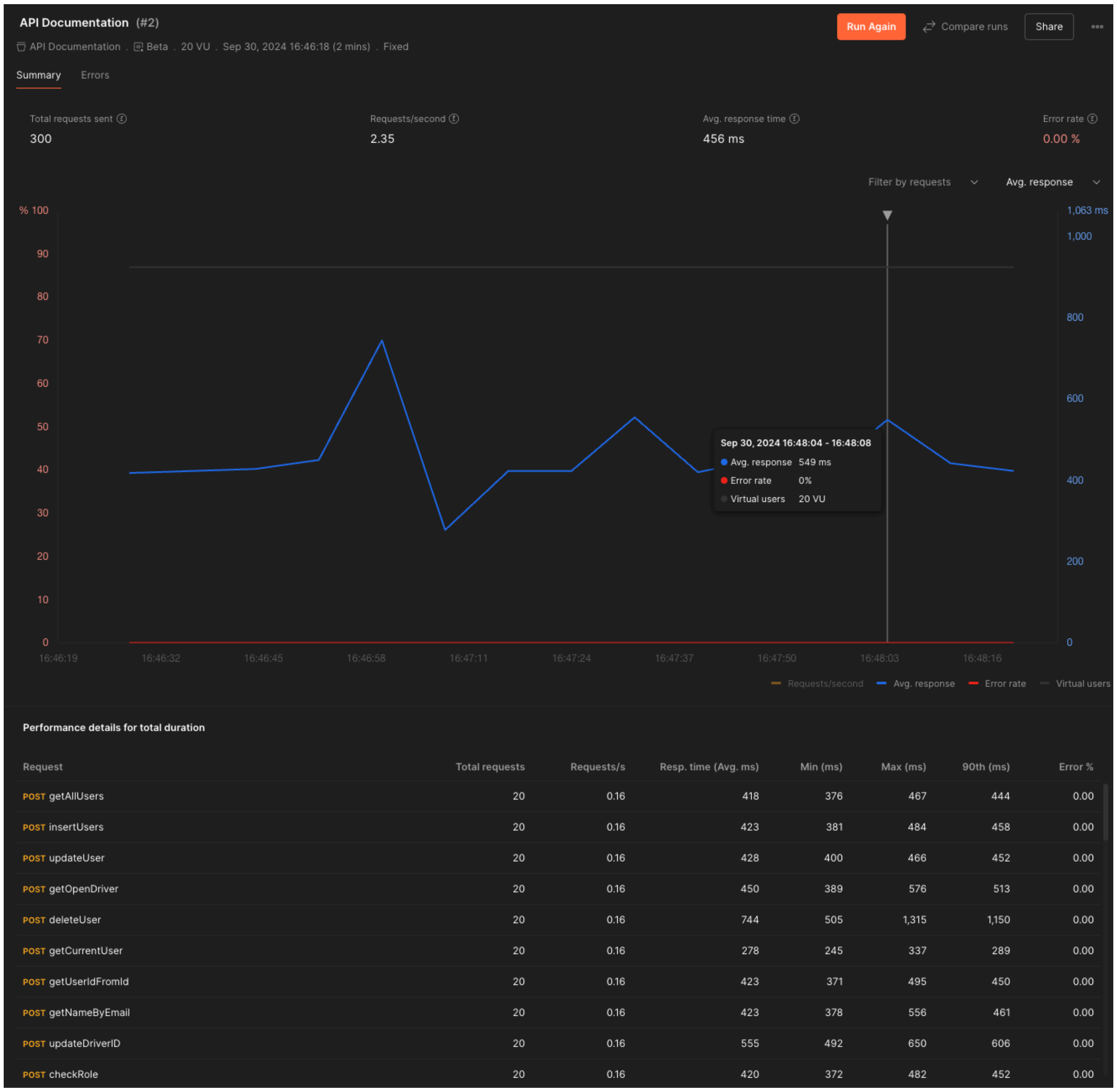
The scalability of the system was validated by monitoring the system's ability to maintain performance while handling increasing user loads. The stress test described above was also used to observe the system's horizontal scaling behavior, ensuring that it could handle an expanding number of requests without performance degradation.

- **Test Scenario:** Gradually increase the number of concurrent users to test scaling limits.
- **Expected Outcome:** The system should automatically scale to accommodate higher user volumes without compromising response times or data processing capabilities.


API Documentation - Run results

Ran today at 17:00:14 · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	1	6m 45s	1494	828 ms



API Documentation (#2)

API Documentation . Beta . 20 VU . Sep 30, 2024 16:46:18 (2 mins) . Fixed

Total requests sent	Requests/second	Avg. response time	Error rate
300	2.35	456 ms	0.00 %

Performance details for total duration

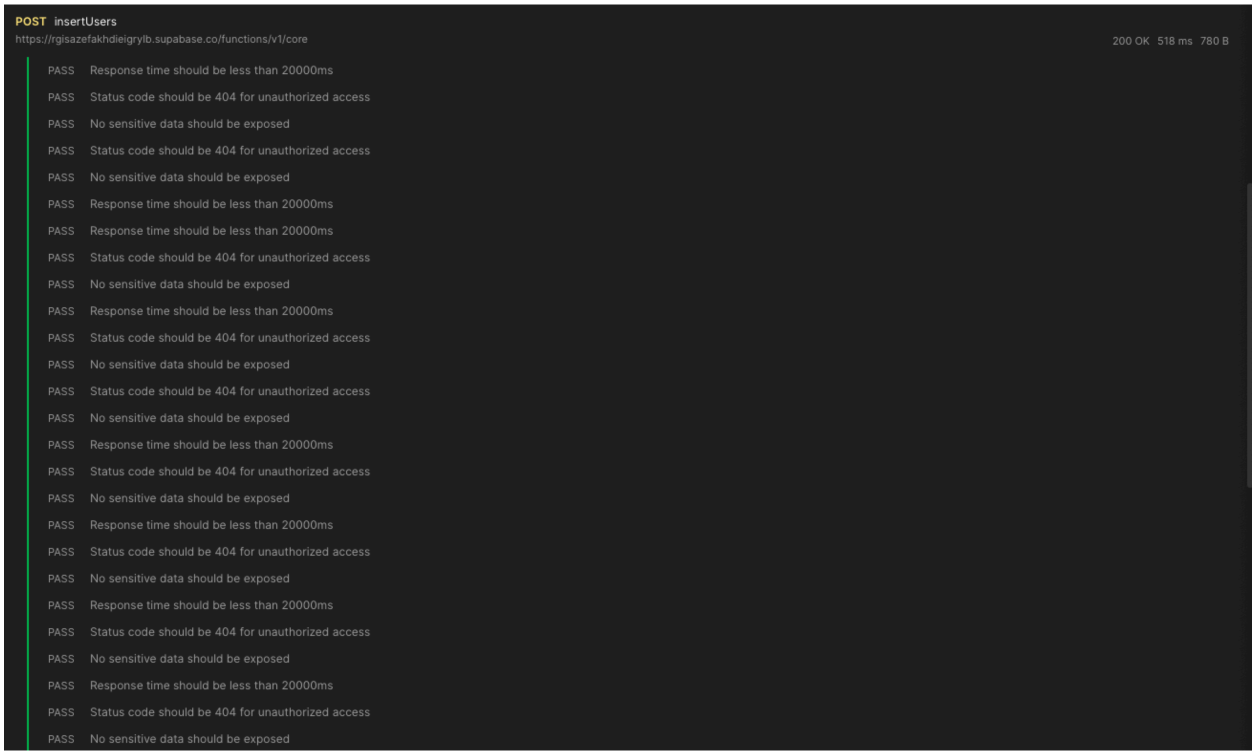
Request	Total requests	Requests/s	Resp. time (Avg. ms)	Min (ms)	Max (ms)	90th (ms)	Error %
POST getAllUsers	20	0.16	418	376	467	444	0.00
POST insertUsers	20	0.16	423	381	484	458	0.00
POST updateUser	20	0.16	428	400	466	452	0.00
POST getOpenDriver	20	0.16	450	389	576	513	0.00
POST deleteUser	20	0.16	744	505	1,315	1,150	0.00
POST getCurrentUser	20	0.16	278	245	337	289	0.00
POST getUserIdFromId	20	0.16	423	371	495	450	0.00
POST getNameByEmail	20	0.16	423	378	556	461	0.00
POST updateDriverID	20	0.16	555	492	650	606	0.00
POST checkRole	20	0.16	420	372	482	452	0.00

Security Testing

To ensure the system's security, we conducted a test where API requests were sent **without authorization headers** to evaluate whether any sensitive data could be accessed or leaked.

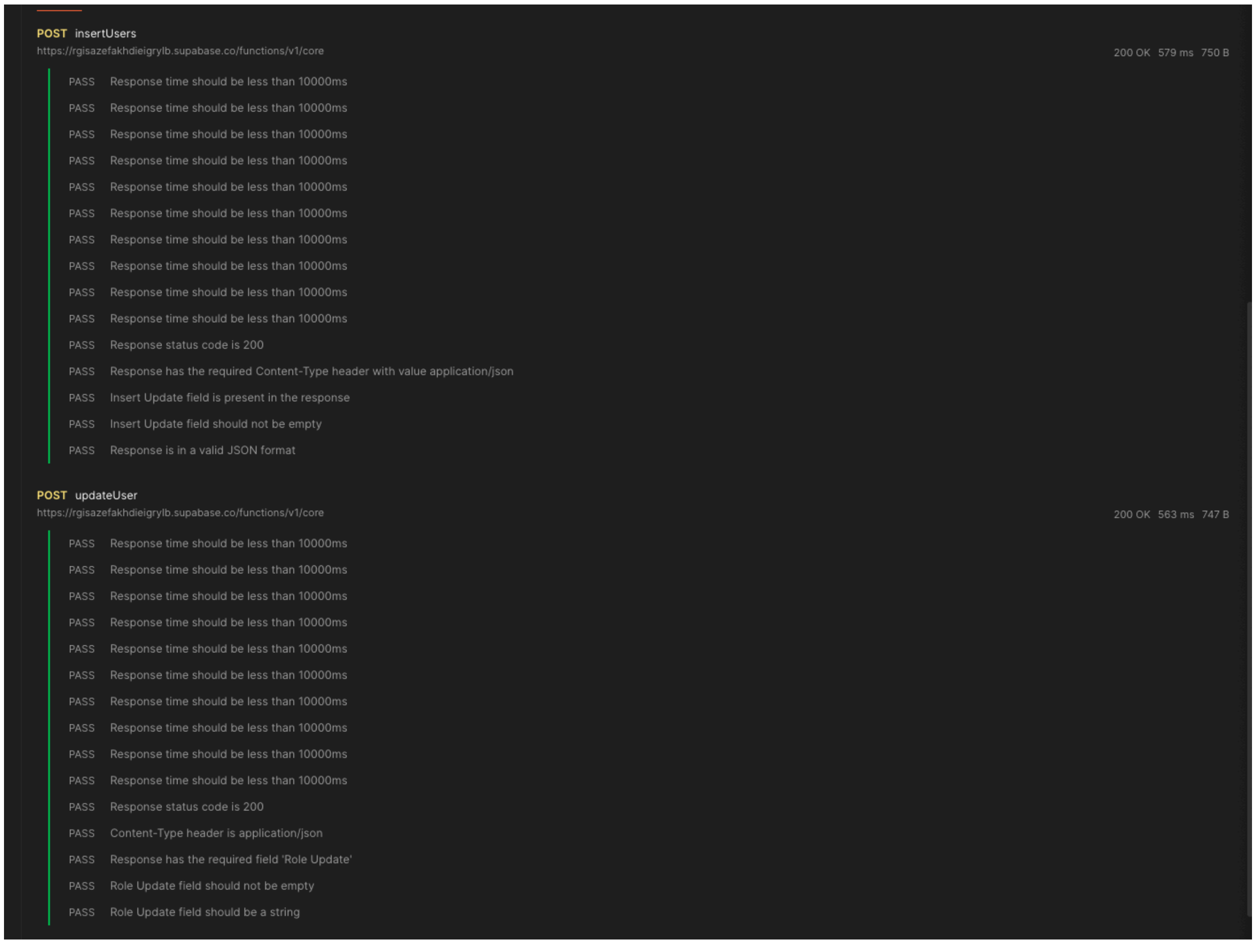
This test was essential to verify that the API strictly enforced authentication and authorization mechanisms.

- **Test Scenario:** Send unauthorized API requests to key endpoints.
- **Expected Outcome:** The system should reject all unauthorized requests and return appropriate error messages (e.g., HTTP 401 Unauthorized).
- **Result:** The API rejected all unauthorized requests, and no sensitive data was exposed, confirming the robustness of the system's security protocols



```
POST insertUsers
https://rgisazefakhdieigrylb.supabase.co/functions/v1/core                200 OK  518 ms  780 B

    PASS  Response time should be less than 20000ms
    PASS  Status code should be 404 for unauthorized access
    PASS  No sensitive data should be exposed
    PASS  Status code should be 404 for unauthorized access
    PASS  No sensitive data should be exposed
    PASS  Response time should be less than 20000ms
    PASS  Response time should be less than 20000ms
    PASS  Status code should be 404 for unauthorized access
    PASS  No sensitive data should be exposed
    PASS  Response time should be less than 20000ms
    PASS  Status code should be 404 for unauthorized access
    PASS  No sensitive data should be exposed
    PASS  Status code should be 404 for unauthorized access
    PASS  No sensitive data should be exposed
    PASS  Response time should be less than 20000ms
    PASS  Status code should be 404 for unauthorized access
    PASS  No sensitive data should be exposed
    PASS  Response time should be less than 20000ms
    PASS  Status code should be 404 for unauthorized access
    PASS  No sensitive data should be exposed
    PASS  Response time should be less than 20000ms
    PASS  Status code should be 404 for unauthorized access
    PASS  No sensitive data should be exposed
    PASS  Response time should be less than 20000ms
    PASS  Status code should be 404 for unauthorized access
    PASS  No sensitive data should be exposed
```



```
POST insertUsers
https://rgisazefakhdieigrylb.supabase.co/functions/v1/core                200 OK  579 ms  750 B

    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response status code is 200
    PASS  Response has the required Content-Type header with value application/json
    PASS  Insert Update field is present in the response
    PASS  Insert Update field should not be empty
    PASS  Response is in a valid JSON format

POST updateUser
https://rgisazefakhdieigrylb.supabase.co/functions/v1/core                200 OK  563 ms  747 B

    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response time should be less than 10000ms
    PASS  Response status code is 200
    PASS  Content-Type header is application/json
    PASS  Response has the required field 'Role Update'
    PASS  Role Update field should not be empty
    PASS  Role Update field should be a string
```

Conclusion

The logistics optimization system uses Vitest for unit tests to verify the accuracy of individual components and Postman API for integration tests to ensure the system’s components work together as expected. Both types of tests will run automatically using GitHub Actions upon code commits and pull requests, ensuring the system remains robust and scalable. Our testing shows that our system is able to handle large amounts of requires proving it is able to handle the performance,scalability and security require for South Africa's logistics sector