Approach to testing API's

I started by familiarising myself with the API's authentication endpoint and reviewing the headers. Once I realised that the header was marked as: "no store, no cache, must re-validate", I created a collection level pre-script that would run before each request, authenticating via the relevant endpoint and continually updating the AUTH_TOKEN collection variable.

From then on, I sought to create a minimal amount of requests, using programmatic handling of collection variables to allow my test to be run either in the postman collection runner or newman. Each request URL is constructed using collection variables to ensure they are easy to modify quickly.

Each request, where applicable, is split to handle requests for successful scenarios, and requests for unsuccessful scenarios, ensuring that all tests are "Successful" with respect to handling either success or failure.

For example, when testing the DELETE endpoints, I split the requests by:

- Delete valid id
    - Testing for an expected 204 code
    - Testing for an empty or undefined body
- Delete invalid id
    - Testing for an expected 404 code indicating that the post was not found
    - Testing for a valid error message that present and was not zero-length

This approach allowed me to ensure that error handling is properly validated across all end points.

This method allowed me to run 63 requests, with 80 total assertions, utilising only 10 core requests, as demonstrated by the screenshot of the htmlextra summary below:

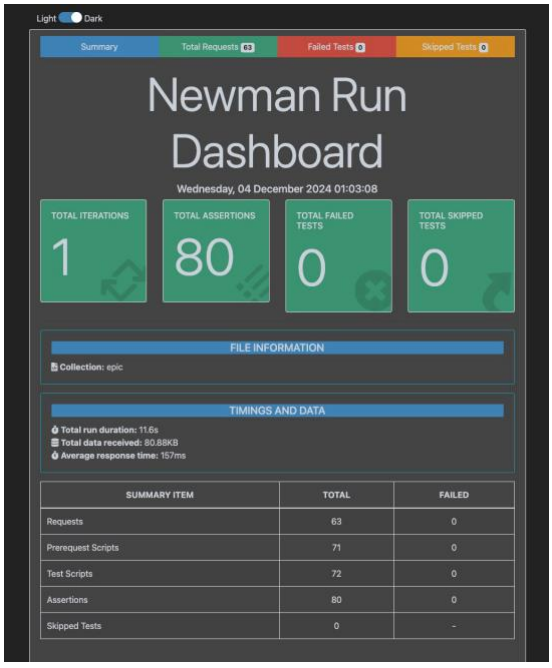

Figure 1. Newman html extra summary

Boundary and edge case testing

An edge case that I have not yet fully implemented in the testing suite is the sanitisation of inputs, ensuring that no malicious code can be sent via the API and executed on the server.

As an example, I was attempting to pass a function to the /authuserapi/post/{id} endpoint, by encoding it and then seeing what came in the response, but I was having some difficulty handling this and null testing for a null value response at the same time, so I stuck with the null and malformed URL tests.

Though technically passing an encoded function string may also fall under the malformed URL tests, I was specifically checking to see if the response body was containing any log messages or indicating that there was some sort of code running as a result on the server.

Instead, I threw in a test:

```
// Test: No sensitive data is exposed
pm.test("Response does not expose sensitive information", function () {
    pm.expect(pm.response.text()).to.not.match(/stack trace|exception|error at/i);
});
```

Figure 2. Testing for sensitive data exposed by the server

This test checks to see if the response text contains any mention of a stack trace (record of function calls), exceptions (indicating that code is attempting to be run unsuccessfully) or error at which might reveal a file name or other important information about the server storage/config.

Though this was not applied to the POST or update methods, as it was not explicitly requested, I wanted to try it with the get requests and see if there was something unprotected there instead. Considering that they are generally not meant to create anything, I thought they might serve as a weak-point where rules may not usually be in place to protect against such methods. That being said, my knowledge of enterprise API construction is still growing, and I am still gaining a deeper knowledge regarding the best practices of edge case testing.

Another edge case I would like to test is storing functions as post/user parameters. E.g a function used to grab the folder structure or other sensitive info in the email field, or in the username field that then passes sensitive information about the server to the user in a welcome email. There are many other scenarios, I would like to test, given the time. For instance, SQL or NoSQL injection in all fields, massive payloads etc.

I have also not implemented any sort of load testing yet, that would certainly be useful, perhaps by utilising a python script with multiple threads and assertions in pytest.

Error Handling Validation

Some strategies I employed to ensure that the API handles errors consistently and predictably were:

1. Splitting my test cases in to success and failure cases (testing for success or testing for expected failure)
2. Using preset test values in collection variables that ensured test values of multiple data types were considered.
3. Redundancy functions like making a request to get a limited number of ids from existing posts when looking to test the /authapi/posts/{id} endpoint.

```
async function fetch_valid_ids() {
    const valid_posts_url = `
        ${pm.collectionVariables.get("BASE_URL")}
        ${pm.collectionVariables.get("POSTS")}
        ?limit=3
    `.replace(/\s+/g, '').trim();

    return new Promise((resolve, reject) => {
        pm.sendRequest({
            url: valid_posts_url,
            method: 'GET',
            header: {
                "Authorization": `Bearer
${pm.collectionVariables.get("AUTH_TOKEN")}`
            }
        }, function (err, response) {
            if (err) {
                console.error("Error fetching valid IDs:", err);
                reject(err);
            } else {
                const ids = response.json().map(post => post.id);
                resolve(ids);}
        });
    });
}
(async function () {
    let valid_ids = JSON.parse(pm.collectionVariables.get("VALID_IDS") || "[]");
    if (valid_ids.length === 0) {
        console.log("Valid IDs are empty. Fetching new valid IDs...");
        valid_ids = await fetch_valid_ids();
        pm.collectionVariables.set("VALID_IDS", JSON.stringify(valid_ids));
    }
    const curr_id = valid_ids.shift();
    pm.collectionVariables.set("CURRENT_ID", curr_id);
    pm.collectionVariables.set("VALID_IDS", JSON.stringify(valid_ids));
})();
```

Figure 3. Ensuring valid ids are fetched if none are set

This function flow allows for valid ids to be asynchronously fetched if none are set in the collection variables. This is easier to do for cases where we are testing for success, less so in cases where I am trying to test invalid payloads, etc.

Automation Strategy

I set out to ensure all my tests could be automated from the beginning of this challenge. The core design choices that facilitate this are:

I designed my API testing framework with automation as a key focus. The following design choices and tools ensure scalability, efficiency, and maintainability:

Dynamic Variable Management:

The consistent use of collection variables ensures tests are flexible and adaptable across environments (e.g., dev, staging, production). This also enables dynamic request handling for valid and invalid payloads, IDs, and endpoints.

Folder and Request Organization:

Requests are grouped by functionality (e.g., CREATE, DELETE) and scenarios (success vs. failure), improving readability and supporting incremental test suite expansion.

Pre and Post-Scripting:

Scripts are used to fetch data, dynamically construct requests, and validate responses, allowing for highly automated and reusable test cases.

Controlled Test Execution:

Using the pm.setNextRequest method, I programmatically control the execution flow to iterate through multiple test cases in a single run, saving time and reducing manual intervention.

CI/CD Integration with Newman:

The test suite is designed to run headlessly using Newman, supporting automation workflows in tools like GitHub Actions. For example:

- Upon a pull request, GitHub Actions runs the test suite using Newman.
- Reports (e.g., HTML Extra) are generated, providing detailed feedback for developers and stakeholders.
- Code merging is blocked if critical test assertions fail.

Future Enhancements:

- **Load and Stress Testing**: Integration with performance tools like Artillery or JMeter to simulate high-traffic scenarios.
- **Enhanced Monitoring**: Using Postman Monitors or Datadog to provide proactive API health alerts.
- **Advanced Reporting**: Automating Newman results into dashboards or Slack notifications.

Performance Testing

Response Time

- Average – Tracks the overall average response time, providing a baseline for the API's general performance. A high average may indicate underlying inefficiencies.
- Median – Gives a performance measure that reduces the impact of outliers (e.g., rare slow requests) and is often more representative of typical user experience.
- Max – Identifies the slowest response time, highlighting performance bottlenecks or resource contention under specific conditions.

Availability

- Uptime – Measures how often the API is available over a given period, typically represented as a percentage (e.g. 99.9%). This metric helps determine reliability and adherence to SLAs.
- Error Rate – Tracks the percentage of requests resulting in errors, offering insights into how often the API fails compared to successful responses.

Throughput

- Requests per second – Measures the volume of requests the API can handle within a second, a critical metric for determining scalability under load.
- Concurrent connections – Tracks the number of simultaneous connections the server can handle before experiencing issues, indicating potential memory or CPU bottlenecks.

Timeouts and Error Spread

- Timeouts – Measures the frequency of timeouts when the API fails to respond within the expected timeframe, which can be caused by server overload or inefficient code.
- Error spread – Tracks the ratio of 4xx (client-side) to 5xx (server-side) errors to help identify whether failures stem from client misuse or server-side problems.

Security

- Number of failed login attempts – Tracks how often users attempt to log in with incorrect credentials, which can help detect brute-force attacks.
- Rate of invalid tokens or API keys – Monitors how often expired, revoked, or tampered tokens are used, signalling unauthorized access attempts or token misuse.
- Unusual IP activity – Tracks login attempts or requests from unusual or unexpected IP addresses, which may indicate a targeted attack or suspicious activity.