Test Plan - IFTG_QA_VanPham

1. Test Plan Overview

This document outlines the test strategy and approach for testing selected endpoints from the Reqres API. The goal is to ensure that the API is functioning as expected, covering a wide range of scenarios, including positive, negative, and end-to-end workflows.

2. Objectives

- Validate the correct functioning of the selected API endpoints under various scenarios.
- Ensure the API handles both valid and invalid requests gracefully.
- Verify the correct implementation of authentication mechanisms.
- Confirm that the API performs well under different conditions and environments.
- Generate comprehensive test reports for analysis and decision-making.

3. Scope

The testing scope includes the following API endpoints:

- **POST:** `/api/login` - User Login
- **POST:** `/api/users` - Create a user
- **GET:** `/api/users/2` - Retrieve a single user
- **PUT:** `/api/users/2` - Update user information
- **DELETE:** `/api/users/2` - Delete a user

These endpoints will be tested across both Test and Stage environments.

4. Test Approach

4.1 Types of Testing

- **Functional Testing:** To verify that each endpoint performs its intended function. This involves verifying the expected outputs, response codes, and

data integrity for each API call. Test cases will cover both valid and edge-case scenarios to ensure comprehensive coverage.

- **Negative Testing:** To ensure the API handles invalid inputs and scenarios appropriately. This includes testing scenarios such as invalid input formats, missing required fields, and unauthorized access attempts. Negative testing helps identify potential vulnerabilities and ensures the API's robustness in handling unexpected situations.
- **End-to-End Testing:** To validate complete workflows across multiple endpoints. This includes testing scenarios such as creating a user, retrieving user details, updating user information, and deleting the user. End-to-end testing helps ensure that different API endpoints work together seamlessly and that the overall system functions as expected.
- **Performance Testing:** To assess the API's responsiveness and stability under load (optional if integrated into the test framework).
- **Compatibility Testing:** To verify that the API functions correctly across different platforms, browsers, and devices. This ensures that the API can be accessed and used effectively by a wide range of clients, maintaining consistency in functionality and performance across various environments.
- **Security Testing:** To ensure that authentication, authorization, and data handling are secure. Additionally, security testing will focus on validating proper implementation of authentication mechanisms, checking for potential vulnerabilities such as SQL injection, ensuring that sensitive data is properly encrypted and protected. This approach helps maintain the integrity and confidentiality of the API and its data.

4.2 Test Environments

- **Test Environment:** A controlled environment to test the API in isolation.
- **Staging Environment:** A pre-production environment that closely mirrors the production setup.

4.3 Tools & Frameworks

- **Playwright with TypeScript:** For writing and executing automated tests.
- **dotenv:** For managing environment variables.
- **Reporting Tools:** Playwright's built-in reporting, with potential integration of HTML reports for better visualization.

5. Test Scenarios

5.1 User Login (POST: `/api/login`)

- **Positive Test:** Log in with valid credentials. Validate that a token is returned.
- **Negative Test:** Attempt to log in with an incorrect password. Validate that the API returns a 400 status with an error message "user not found."
- **Negative Test:** Attempt to log in with a non-existent email. Validate that the API returns a 400 status with an appropriate error message.

5.3 Get Single User (GET: `/api/users/2`)

- **Positive Test:** Retrieve details for user ID 2. Validate that the correct user details are returned.
- **Negative Test:** Attempt to retrieve details for a non-existent user ID (e.g., `/api/users/23`). Validate that the API returns a 404 status.
- **Negative Test:** Attempt to retrieve user details with an invalid ID format (e.g., `/api/users/abc`). Validate that the API returns a 400 status.

5.4 Update User (PUT: `/api/users/2`)

- **Positive Test:** Update user information with valid data (e.g., name, job). Validate that the changes are reflected in the response.
- **Negative Test:** Attempt to update a user with incomplete data (e.g., missing name). Validate that the API returns a 400 status.
- **Negative Test:** Attempt to update a non-existent user ID. Validate that the API returns a 404 status.

5.5 Delete User (DELETE: `/api/users/2`)

- **Positive Test:** Successfully delete the user with ID 2. Validate that the API returns a 204 status with no content.
- **Negative Test:** Attempt to delete a non-existent user ID (e.g., `/api/users/23`). Validate that the API returns a 404 status.
- **Negative Test:** Attempt to delete a user with an invalid ID format (e.g., `/api/users/abc`). Validate that the API returns a 400 status.

5.6 End-to-End Workflow Tests

- **Workflow 1: Test Login, Create Then Delete User**.
  - Preconditions: Valid user credentials
  - Steps & Expected Result:
    1. Send a `POST /login` request with valid user credentials
    2. Verify that the response contains an auth token
    3. Send a `POST /api/users` request to create a user
    4. Verify user created
    5. Send a `GET /api/users/{userId}` to get created user
    6. Verify user information
    7. Send a `DELETE /api/users/{userId}` to delete the created user
    8. Verify user is deleted successfully and get user returns 404 not found.
- **Workflow 2: Test Login, Create, View And Update User Details**
  - Preconditions: Valid user credentials
  - Steps & Expected Result:
    1. Send a `POST /login` request with valid user credentials
    2. Verify that the response contains an auth token
    3. Send a `POST /api/users` request to create a user
    4. Verify user created
    5. Send a `GET /api/users/{userId}` to get created user
    6. Verify user information
    7. Send a `PUT /api/userss/{userId}` request to update user
    8. Verify user update successfully. Verify new user information.

6. Test Data Management

- **Test Data Source:** Static data for testing will be defined in configuration files, with dynamic data generation where necessary.

- **Data Cleanup:** Ensure that any test data created during testing (e.g., user creation) is properly cleaned up after tests are executed, especially in persistent environments.

7. Test Execution Strategy

- **Execution Frequency:** Tests will be executed nightly in the Test environment and on-demand in the Stage environment.
- **Parallel Execution:** Tests will be run in parallel to reduce execution time, leveraging Playwright's capabilities.

8. Test Resources Matrix

Roles	Responsibility	Availability	Dependencies
QA Lead	Oversee test strategy, coordinate team efforts, review test results	Full-time during analysis and testing phases	Support from management for resource allocation
QA Member	Design and execute test cases, report bugs, maintain test documentation. Maintain test environments, manage CI/CD pipeline	Full-time during analysis and testing phases	Test feature and environment ready
Developer	Provide technical support, fix reported issues	Part-time during testing, implementation phase	Access to up-to-date project requirements and specifications
Business Analyst	Clarify requirements, assist in test case creation	Part-time	Timely input from stakeholders on requirements
Project Manager	Monitor progress, manage resources, communicate with stakeholders	Part-time	Cooperation from team for development and verification

9. Reporting & Metrics

- **Report Format:** HTML reports will be generated to visualize test results, including pass/fail rates, execution time, and error logs.

- **Metrics:** Key metrics like test coverage, defect density, and execution time will be tracked.

10. Risks & Mitigation

- **Flaky Tests:** Address flaky tests by improving test reliability through proper synchronization and data handling.
- **Time Constraints:** Manage time effectively by prioritizing critical test cases and automating repetitive tasks. Implement parallel test execution to reduce overall test duration without compromising coverage.
- **Environment Instability:** Ensure environment stability by validating that test environments are correctly configured and maintained.
- **Data Inconsistency:** Use isolated data sets and proper cleanup to avoid data-related issues.
- **Tool Limitations**: Regularly assess and update the test automation framework to address any limitations in Playwright or other tools used. Stay informed about new releases and features that could enhance test coverage and efficiency. Implement workarounds or alternative solutions when necessary to ensure comprehensive testing despite tool constraints.
- **API Changes**: Stay informed about any updates or changes to the API specifications. Regularly review API documentation and communicate with the development team to ensure test cases remain aligned with the latest API versions. Implement a flexible test framework that can easily adapt to API changes without requiring significant rework. Implement a robust change management process to handle API modifications effectively. This includes maintaining a version control system for test scripts, establishing a clear communication channel with the development team for timely updates, and conducting regular reviews of API documentation. By staying proactive and adaptable, the testing team can minimize the impact of API changes on the overall testing process and maintain high-quality assurance standards.
- **Incomplete Features**: Ensure comprehensive testing of all API features, including those that may be partially implemented or in development. Coordinate with the development team to identify and prioritize testing of incomplete features to catch potential issues early in the development cycle.