**Test Strategy**

**Objective:**

To ensure robust testing coverage for the API and provide recommendations for a stable test environment.

**Testing Layers:**

1. **Unit Tests**:
   * Verify the functionality of isolated units (e.g., functions, services).
   * **Framework**: Jest (already common in TypeScript projects like NestJS).
2. **Integration Tests**:
   * Validate interaction between modules
   * Mock external services where applicable.
   * **Framework**: Super test with Jest for HTTP-level tests.
3. **API Contract Tests**:
   * Ensure the API adheres to a predefined schema and doesn’t break consumer expectations.
   * **Framework**: Postman
4. **End-to-End (E2E) Tests**:
   * Simulate real-world scenarios, covering user workflows.
   * Focus initially on critical endpoints (POST /campaigns and GET /campaigns).
   * **Framework**: Cypress

**Environment Recommendations (for DevOps):**

1. **Stable Staging Environment**:
   * Use **Docker** to containerize the API and database for consistent testing.
   * Implement a CI/CD pipeline for automated deployment of test environments (e.g., GitHub Actions).
2. **Database Management**:
   * Create reproducible datasets using database seeding scripts.
   * Reset database state before each test run.
3. **Mocking External Dependencies**:
   * Use a library like **WireMock** for external APIs to ensure consistent responses.

**2. Documenting the API**

**Tool Selection:**

* Use **Swagger (OpenAPI)** or postman for detailed documentation, enabling developers and Product Owners to interact with the API directly.

**Implementation Steps:**

**1. Overview of API Testing Approach**

Provide a brief overview of the testing process for the Campaign API, which includes the testing of the following CRUD operations:

1. **Create Campaign** (POST request)
2. **Get Campaign** (GET request)
3. **Update Campaign** (PUT request)
4. **Delete Campaign** (DELETE request)

Explain how these tests will be implemented using **Postman** and how automation will be used to ensure reliable and repeatable tests.

**2. Pre-requisites**

Before running the tests, the following pre-requisites should be in place:

1. **Postman** installed and configured.
2. Environment variables set up in Postman:
3. baseUrl: API base URL (e.g., http://localhost:8081)

campaignId: This will store the dynamically created campaign ID.

campaignData: JSON object with campaign details for creating the campaign.

UpdatedCampaignData: JSON object for updating the campaign.

**3. Postman Request and Test Details**

* 1. **Create Campaign (POST Request)**

**Method**: POST

**Endpoint**: {{baseUrl}}/campaign

Use environment variables to pass campaign data in JSON format (e.g., campaignData).

**Test Implementation**:

1. **Verify the response status code is 201** (Created).
2. **Verify the response contains a campaign ID**.
3. **Verify the response body matches the input data** (i.e., the campaign name, client, category, etc.).

**3.2 Get Campaign (GET Request)**

1. **Method**: GET
2. **Endpoint**: {{baseUrl}}/campaign/{{campaignId}}
3. **Test Implementation**:
4. **Verify the response status code is 200** (OK).
5. **Verify the campaign name matches the original name** passed in the creation request.
6. Example test script in Postman:

**3.3 Update Campaign (PUT Request)**

1. **Method**: PUT
2. **Endpoint**: {{baseUrl}}/campaign/{{campaignId}}
3. **Request Body**:
4. Use environment variables to pass updated campaign data (e.g., updatedCampaignData).
5. **Test Implementation**:
6. **Verify the response status code is 200** (OK).
7. **Verify the updated campaign name** is as expected.

**3.4 Delete Campaign (DELETE Request)**

1. **Method**: DELETE
2. **Endpoint**: {{baseUrl}}/campaign/{{campaignId}}
3. **Test Implementation**:
4. **Verify the response status code is 200** (OK) indicating successful deletion.

**3.5 Verify Campaign Deletion**

1. **Method**: GET
2. **Endpoint**: {{baseUrl}}/campaign/{{campaignId}} (to verify that the campaign no longer exists).
3. **Test Implementation**:
4. **Verify the response status code is 404** (Not Found) after deletion.

**4. Test Automation Execution**

Once the above tests are set up in Postman, they can be executed manually or automatically in a continuous integration (CI) pipeline.

**Manual Execution**: Run the collection of tests directly from Postman.

**Automated Execution**:

Use **Newman**, a command-line collection runner for Postman, to execute the collection in an automated manner.

Example command to run Postman tests via Newman:

newman run campaign-api-collection.json -e environment.json

**5. Continuous Integration (CI) Integration**

To integrate the tests into a CI pipeline (such as Jenkins, GitLab CI, or GitHub Actions), follow these steps:

**Install Newman** (if running tests in CI):

npm install -g newman

**Run Postman Collection**: In your CI configuration file, add a step to run the Postman collection using Newman. Example in a **Jenkins pipeline**:

node --max\_old\_space\_size=8192 $(npm bin)/newman run campaign-api-collection.json -e environment.json

**6. Reporting and Logs**

During test execution, generate detailed logs for each request. Use **Newman** to generate HTML or JSON reports:

Example command to generate an HTML report:

newman run campaign-api-collection.json -e environment.json -r html

The output report will provide insights into the status of each request, allowing for easier troubleshooting and validation.

**7. Additional Considerations**

**Data Cleanup**: Ensure that any created data is cleaned up after each test run to avoid conflicts with subsequent tests.

**Environment and Configuration**: Ensure that the API environment (e.g., staging, production) and configurations are set correctly before running tests.

**Error Handling**: Implement robust error handling in the Postman scripts to ensure that the tests fail gracefully when unexpected conditions arise.

**3. Implementing Cypress Automated Tests**

**Framework: Cypress**

* Cypress provides a good balance of simplicity and power for API testing.
  + Test for edge cases like missing or invalid fields.

**Test Data Preparation**:

* + Define the input data required to create a campaign (e.g., id, name, client, category, type, status).

1. **Test Steps**:
   * **Step 1**: Make a POST request to /campaign with the defined campaign data.
   * **Step 2**: Verify that the response status code is **201 Created**.
   * **Step 3**: Check that the response body contains the campaign details and that the id field matches the generated ID.
   * **Step 4**: Ensure the response body contains the correct name, client, category, and status.
2. **Error Handling**:
   * Include tests to simulate invalid input (e.g., missing required fields) to ensure that the API responds with the correct error status codes (e.g., **400 Bad Request**).
3. **Test Automation Execution**:
   * Use npx cypress run for headless test execution.
   * Run the tests in a continuous integration pipeline to ensure consistent testing after every code change.

**5. Continuous Integration (CI) Integration**

* Integrate the Cypress tests into the CI pipeline (e.g., using Jenkins, GitLab CI, or GitHub Actions) to automate test execution with every code push or merge request.

**6. Conclusion**

This strategy focuses on validating the functionality of the **POST /campaigns** endpoint by automating the test scenario for campaign creation. Using **Cypress** will streamline test execution and ensure that the API behaves as expected in both development and production environments.

**DevOps Recommendations for API Test Automation**

1. **Test Environment Management**:
   * Set up isolated and consistent test environments that mimic production.
   * Use environment variables for configuration (e.g., API keys, base URLs).
   * Ensure proper database management (e.g., test databases, data seeding, and cleanup).
2. **CI/CD Integration**:
   * Integrate automated tests into the CI/CD pipeline to run tests on every code change (pull requests, commits).
   * Use parallel test execution to reduce the overall test runtime.
   * Fail builds early if critical tests fail (e.g., regression failures, API errors).
3. **Test Reporting and Monitoring**:
   * Generate detailed reports with logs, screenshots, and videos for test failures.
   * Set up alerts for test failures and notify the relevant stakeholders.
   * Continuously monitor test health and coverage to ensure reliable results.
4. **Version Control and Test Data Management**:
   * Maintain versioning for test scripts and ensure that they are in sync with application changes.
   * Use mock or test-specific data for tests to avoid data pollution in production.
   * Ensure test data is cleaned up post-test execution.
5. **Security and Compliance**:
   * Avoid hardcoding sensitive data (e.g., credentials) in test scripts; use secure storage (e.g., Vault, AWS Secrets Manager).
   * Implement security scans within the CI pipeline to identify vulnerabilities.
   * Ensure API security headers and rate limits are validated in automated tests.
6. **Test Automation Maintenance**:
   * Regularly review and update test cases to reflect changes in the API (new/removed endpoints, modified data structures).
   * Address flaky tests promptly to maintain test reliability.
   * Clean up test data to prevent conflicts during subsequent test runs.
7. **Collaboration and Feedback Loop**:
   * Foster communication between DevOps, Development, and QA teams to align on test case creation and updates.
   * Provide quick feedback to developers about test results to allow them to take immediate corrective actions.
   * Implement a test impact analysis to assess the effect of code changes on existing test cases.