GitHub API Automation Testing Assignment

Overview:

In this assignment, you will write automated tests to validate key GitHub API endpoints, focusing on both authenticated and unauthenticated scenarios. This will help you explore various API flows and demonstrate the testing techniques you are familiar with.

The exercise objective is to play and showcase your automation knowledge interacting with:

- `users` API endpoints, to interact with Github registered users.
- `repos` API endpoints, to interact with the repository information present on the API.
- `commits` API endpoints, to obtain the commits information.

Feel free to extend this exercise by integrating the outcome of this exercise into automation flows that you consider useful or would like to showcase for this assignment.

Assignment Tasks.

Task 1: External user public profile

- **Objective:**
  - Retrieve the public profile information of a specific user without authentication.
  - Endpoint: `https://api.github.com/users/{username}`
- **Expected Outcome:**
  - Validate the possible status codes matching endpoint spec.
  - Ensure key public fields are returned.

Task 2: Logged user own profile

- **Objective:**
  - Retrieve the profile information of the logged-in user using an access token.
  - Endpoint: `https://api.github.com/user`
- **Expected Outcome:**
  - Validate the possible status codes matching endpoint spec.
  - Ensure additional user-specific fields (such as private details) are included.

Task 3: List repositories

- **Objective:**
  - Retrieve a list of public repositories for a user.
  - Endpoint: `https://api.github.com/users/{username}/repos`

- **Expected Outcome:**
    - Validate the possible status codes matching endpoint spec.
    - Ensure key fields are returned.

Task 4: List repositories for logged user

- **Objective:**
    - Retrieve a list of repositories (both public and private) for the logged-in user.
    - Endpoint: `https://api.github.com/user/repos`
- **Expected Outcome:**
    - Validate the possible status codes matching endpoint spec.
    - Validate that both public and private repositories are listed.
    - Ensure key fields are returned.

Task 5: List Commits of a repository

- **Objective:**
    - Retrieve the list of commits for a public repository without authentication.
    - Endpoint: `https://api.github.com/repos/{owner}/{repo}/commits`
- **Expected Outcome:**
    - Validate the possible status codes matching endpoint spec.
    - Validate all commits are listed, with proper handling of pagination if needed.
    - Validate the list of commits includes fields such as `sha`, `author`, `message`, and `date`.

Task 6: Update logged user metadata

Objective: Update the metadata of the logged-in user using an access token. This task tests the ability to modify user information and ensure that the updates are reflected accurately.

Endpoint: `https://api.github.com/user`

Expected Outcome:

- Validate the possible status codes matching endpoint spec.
- Ensure that user-specific fields, such as `name`, `bio`, or `blog`, can be updated.
- Ensure the updated fields are reflected in subsequent requests for the user profile.
- Verify that an unauthorized request (missing or invalid token) results in the appropriate error status code.

Task 7: Implement testing workflows

Objective: Create workflows that demonstrate the ability to chain multiple API interactions, including both successful and failed requests. The goal is to simulate real-world usage patterns and validate how different API endpoints work together.

Example Workflow:

1. **Step 1**: Try to retrieve the user's profile without a Bearer token and validate that access is denied (e.g., 401 Unauthorized).
2. **Step 2**: Set the Bearer token and retry fetching the profile, this time validating that access is granted (e.g., 200 OK).
3. **Step 3**: Update a field in the logged-in user's profile, such as the `bio` or `name`.
4. **Step 4**: Retrieve the profile again and validate that the field has been successfully updated.
5. **Step 5**: Obtain the list of repositories for the logged-in user (both public and private). Ensure that the repositories are listed correctly.
6. **Step 6**: Attempt to list commits for a non-existent repository and validate that the appropriate error is returned (e.g., 404 Not Found).
7. **Step 7**: List commits from the first repository of the logged-in user and validate the key fields (`sha`, `author`, `message`, `date`) in the response.
8. **Step 8**: List commits from the last repository of the logged-in user, again validating key fields.

Expected Outcome:

- Ensure status codes are validated at every step.
- Verify that the appropriate error handling is in place for failed requests.
- Validate the correct sequence of operations and responses, ensuring proper flow through the API endpoints.
- Ensure that each request and response in the workflow behaves as expected according to the GitHub API documentation.

GitHub API Documentation

For detailed information on the API endpoints, refer to the official GitHub REST API Documentation (https://docs.github.com/en/rest)

Submission Guidelines:

- Use any test automation framework of your choice.
- Implement authentication for the authenticated tests using a GitHub personal access token.
- Ensure the code is maintainable, modular, and follows best practices.
- Include assertions for status codes, response fields, and error handling.
- Document any challenges or assumptions in a README file.
- Provide guidelines for running the assignment and validating it works as expected.
- Push the assignment with git over GitHub. If privacy is required, a private repository can be provided to the candidate, just share with us the github username.

Evaluation Guidelines:

- **Meet Objectives:** Ensure all task requirements are strictly met and do not hesitate to exceed expectations if you consider it necessary, show us how good you are!
- **Long-Term Focus:** Design solutions that not only solve the immediate problem but also consider scalability and future needs. Write reusable and comprehensive tests that cover a wide range of scenarios and are adaptable to code evolution.
- **Best Practices:** Apply best practices like modularization and code reuse to ensure maintainable and efficient development.
- **Efficient Development:** Avoid quick fixes that lead to technical debt in the long run; focus on writing affordably efficient and maintainable code.
- **Architecture:** Build solutions with a clear structure that supports future changes and easy maintenance and extendibility.
- **Technology Choices:** Select technologies that enhance long-term development efficiency, not just short-term problem-solving. Think about how you would do it in a real work environment.