1.How do you validate and differentiate UI error responses and API error responses ?

To validate and differentiate UI error responses and API error responses, you'll need to understand the differences in how these responses are delivered and how they should be handled in automation testing. Let's break it down:

1.1 UI Error Responses:

- **Nature**: UI error responses are presented directly on the user interface. They may appear as pop-ups, error messages, or other visual elements like warnings or red banners.
- **Validation**: To validate these errors using automation tools like Selenium, you would typically interact with the UI elements where the error messages are displayed.

Example: Validating a UI Error Response in Selenium (Java)

```
import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.Assert;


public class UIErrorResponseValidation {

   public static void main(String[] args) {

      WebDriver driver = new ChromeDriver();

      driver.get("https://example.com/login");


      // Simulating a login failure

      driver.findElement(By.id("username")).sendKeys("wrongUser");

      driver.findElement(By.id("password")).sendKeys("wrongPass");

      driver.findElement(By.id("loginButton")).click();


      // Validate the UI error message
```

```
        WebElement errorMessage = driver.findElement(By.id("errorMessage"));
        Assert.assertEquals(errorMessage.getText(), "Invalid username or password.");

        driver.quit();
    }
}
```

1.2 API Error Responses:

- **Nature**: API error responses are typically delivered as HTTP status codes and JSON/XML payloads. These errors are more structured and can provide specific error codes, messages, and even suggestions for correction.
- **Validation**: Validating API error responses is done by verifying the status code, error message in the response body, and other relevant fields using tools like RestAssured.

Example: Validating an API Error Response in RestAssured (Java)

```
import io.restassured.RestAssured;
import io.restassured.response.Response;
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class APIErrorResponseValidation {
    public static void main(String[] args) {
        RestAssured.baseURI = "https://api.example.com";

        // Sending a request that triggers an error response
        Response response = given()
            .header("Content-Type", "application/json")
            .body("{ \"username\": \"wrongUser\", \"password\": \"wrongPass\" }")
            .post("/login");
```

```
        // Validate the status code and error message in the response
        response.then().statusCode(401)
            .body("error", equalTo("Unauthorized"))
            .body("message", equalTo("Invalid username or password."));
    }
}
```

Differentiating UI and API Error Responses:

1. **Location**:
    - **UI Errors**: Displayed on the web page and require interaction with UI elements.
    - **API Errors**: Delivered via HTTP responses and contain structured data (status codes, JSON/XML).
2. **Validation Approach**:
    - **UI Errors**: Use UI automation tools like Selenium to check for error messages or visual cues.
    - **API Errors**: Use API testing tools like RestAssured to validate the status code and response body.
3. **Technical Information**:
    - **UI Errors**: Usually more user-friendly and less technical, designed for end-users.
    - **API Errors**: Provide detailed technical information useful for debugging, such as error codes and stack traces.

Conclusion:

While both UI and API errors need to be validated, the method differs significantly. UI errors require interaction with the UI elements, whereas API errors are validated through HTTP responses. Knowing how to handle and validate each type is crucial in automation testing to ensure comprehensive test coverage.

2.How do you verify in testing whether it is a API error or UI error ?

To verify whether an issue is due to an API error or a UI error during testing, you can follow a systematic approach. This involves isolating the problem to determine if the error originates from the backend (API) or from the frontend (UI). Here's how you can do it:

2.1. Check the API Response Directly:

- **Use API Testing Tools**: Tools like Postman or RestAssured can be used to directly test the API endpoints. By sending the same request that your UI would send, you can check if the API itself is returning an error.
- **Look for Status Codes**: If the API returns a 4xx or 5xx status code, it's likely an API issue. For example, a 500 Internal Server Error would indicate a server-side issue, while a 400 Bad Request might indicate invalid input sent by the UI.

Example: Testing the API Directly with RestAssured (Java)

```
import io.restassured.RestAssured;

import io.restassured.response.Response;

import static io.restassured.RestAssured.*;


public class APIVerification {

    public static void main(String[] args) {

        RestAssured.baseURI = "https://api.example.com";


        // Directly sending the request to the API

        Response response = given()

            .header("Content-Type", "application/json")

            .body("{ \"username\": \"testUser\", \"password\": \"testPass\" }")

            .post("/login");


        // Check if the API is returning an error

        int statusCode = response.getStatusCode();

        if (statusCode >= 400) {

            System.out.println("API Error: " + response.getBody().asString());

        } else {

            System.out.println("API is working fine.");

        }
```

```
    }
}
```

2.2 Simulate the UI Action and Capture API Calls:

- **Browser Developer Tools**: Use the network tab in browser developer tools to monitor the API calls that are made when interacting with the UI. If an API call fails (e.g., returns a 4xx or 5xx status), the issue likely stems from the API.
- **Check for Errors in the Network Tab**: If the network tab shows that the API call failed, then the issue is on the backend side. If the API call succeeds but the UI behaves incorrectly, the issue is likely on the frontend.

Steps to Verify via Developer Tools:

1. Open your application in a browser (e.g., Chrome).
2. Press F12 to open Developer Tools and navigate to the Network tab.
3. Perform the action that triggers the error (e.g., submitting a form).
4. Look for any failed requests in the Network tab.
    - If you see a failed request (e.g., HTTP status 500), it indicates an API error.
    - If all requests succeed but the UI still displays an error, it's likely a UI issue.

2.3 Analyze UI Behavior:

- **Look for UI-Specific Issues**: If the API returns the correct response but the UI displays an error (e.g., misinterpreting the response or failing to render the data correctly), this indicates a UI problem.
- **Check for JavaScript Errors**: JavaScript errors can cause the UI to fail to handle valid API responses correctly. Use the Console tab in browser developer tools to look for such errors.

Example: Identifying a UI Error in Selenium (Java)

```
import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;


public class UIErrorVerification {

    public static void main(String[] args) {
```

```
        WebDriver driver = new ChromeDriver();
        driver.get("https://example.com/login");

        // Simulating a UI action
        driver.findElement(By.id("username")).sendKeys("testUser");
        driver.findElement(By.id("password")).sendKeys("testPass");
        driver.findElement(By.id("loginButton")).click();

        // Checking if the UI shows an error
        if (driver.findElements(By.id("errorMessage")).size() > 0) {
            System.out.println("UI Error Detected: " +
driver.findElement(By.id("errorMessage")).getText());
        } else {
            System.out.println("No UI Error Detected.");
        }

        driver.quit();
    }
}
```

2.4 Cross-Validation:

- **UI Test with API Mocking**: Use mocking tools to simulate API responses while running UI tests. If the mocked API response leads to a UI error, then the issue is likely in the UI. If the error disappears with mocked responses, then the API is likely the issue.
- **End-to-End Testing**: Run end-to-end tests where the UI interacts with the actual API. Compare the results with what you see in isolated API and UI tests to determine the source of the issue.

Conclusion:

- **API Errors**: These are usually indicated by failed API calls, incorrect status codes, or malformed responses. They can be identified by directly testing the API or by observing the network calls during UI interaction.
- **UI Errors**: These occur when the UI fails to handle API responses correctly or displays incorrect information despite a successful API call. JavaScript errors and incorrect rendering are typical indicators of UI issues.

By following this approach, you can systematically identify whether an issue is due to an API error or a UI error.

3. How to Validate Error Responses in API Testing?

Validating error responses is crucial in API testing to ensure that the API behaves correctly under invalid or unexpected conditions. You should verify that the API returns appropriate HTTP status codes and error messages.

Steps to Validate Error Responses:

- Send a request with invalid data or parameters.
- Assert the HTTP status code matches the expected error code (e.g., 400 Bad Request, 404 Not Found, etc.).
- Verify the error message content.

Programming Example: Here's an example using RestAssured in Java to validate a 404 Not Found error response:

```
import static io.restassured.RestAssured.*;

import static org.hamcrest.Matchers.*;


public class ErrorResponseValidation {

   public static void main(String[] args) {

      given()

         .baseUri("https://api.example.com")

         .basePath("/invalidEndpoint")

      .when()

         .get()

      .then()
```

```
        .statusCode(404)
        .body("error", equalTo("Not Found"))
        .body("message", containsString("Endpoint does not exist"));
    }
}
```

In this example, we check that the status code is 404 and the response body contains the expected error message.

4. What is Latency in API Testing?

Latency refers to the time delay between the client sending a request and receiving a response from the server. It is a crucial performance metric in API testing, indicating how responsive the API is.

- **Latency** is measured in milliseconds (ms).
- Lower latency is better as it means the API responds faster.

Example to Measure Latency:

You can measure the latency using RestAssured by extracting the response time:

```
import io.restassured.response.Response;
import static io.restassured.RestAssured.*;

public class LatencyCheck {
    public static void main(String[] args) {
        Response response =
            given()
                .baseUri("https://api.example.com")
            .when()
                .get("/endpoint");

        long latency = response.getTime();
        System.out.println("Latency: " + latency + " ms");
```

```
    }
}
```

5. What are Different Types of Protocols Used in API Testing?

APIs can communicate over various protocols, depending on the use case and environment.

Common Protocols:

- **HTTP/HTTPS:** The most commonly used protocols for web APIs.
- **SOAP:** A protocol used for exchanging structured information, often in enterprise environments.
- **REST:** Architectural style using HTTP/HTTPS. It is not a protocol itself but often relies on HTTP/HTTPS.
- **gRPC:** A high-performance RPC framework using HTTP/2.
- **MQTT:** A lightweight messaging protocol used in IoT.
- **AMQP:** Advanced Message Queuing Protocol, often used in messaging systems.

6. What Types of Testing Can We Perform in API Testing?

API testing involves several types of tests to ensure the API functions correctly under different conditions.

Types of Testing:

- **Functional Testing:** Verifying that the API functions as expected with correct input.
- **Integration Testing:** Testing the API's interaction with other APIs or services.
- **Performance Testing:** Checking the API's responsiveness, throughput, and stability under load.
- **Security Testing:** Ensuring that the API is secure and handles authentication, authorization, and data encryption correctly.
- **Negative Testing:** Validating how the API handles invalid inputs or scenarios.
- **Boundary Testing:** Testing the API with boundary values (e.g., maximum, minimum).

7. What is API Testing?

API Testing involves directly testing APIs as part of integration testing to determine if they meet expectations for functionality, reliability, performance, and security.

Key Aspects of API Testing:

- **Validation of Endpoints:** Ensuring that the endpoints return the expected results for various inputs.
- **Data Validation:** Verifying that the API processes and returns the correct data.
- **Response Validation:** Checking the structure, content, and status codes of the API responses.
- **Performance Checks:** Assessing the speed, stability, and scalability of the API.
- **Security Verification:** Testing for vulnerabilities like SQL injection, cross-site scripting, and data exposure.

Simple Example of API Testing with RestAssured:

```
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;


public class SimpleAPITest {
    public static void main(String[] args) {
        given()
            .baseUri("https://api.example.com")
            .basePath("/users")
        .when()
            .get("/123")
        .then()
            .statusCode(200)
            .body("id", equalTo(123))
            .body("name", equalTo("John Doe"))
            .body("email", equalTo("john.doe@example.com"));
    }
}
```

In this example, we perform a basic API test to validate the response structure and content for a user endpoint.

---

8. How to Validate Error Responses in API Testing?

Validating error responses is a critical part of API testing, as it ensures the API behaves correctly under erroneous or invalid conditions. Error responses can occur due to invalid inputs, incorrect requests, unauthorized access, or system errors.

Steps to Validate Error Responses:

1. **Identify the Error Scenarios:**
    - Identify possible scenarios that can lead to an error, such as sending invalid data, accessing unauthorized endpoints, or using unsupported HTTP methods.
2. **Trigger the Error Condition:**
    - Craft a request that is expected to fail. For example, sending a request with a missing required field or using an invalid endpoint.
3. **Check the HTTP Status Code:**
    - Verify that the API returns the appropriate HTTP status code (e.g., 400 Bad Request, 401 Unauthorized, 404 Not Found, 500 Internal Server Error).
4. **Validate the Error Message:**
    - Ensure the error message in the response is clear, descriptive, and informative. It should provide enough details to understand the issue.
5. **Validate the Structure of the Error Response:**
    - Confirm that the error response follows the expected schema (e.g., it contains fields like error, message, timestamp, etc.).

Programming Example:

Let's consider a scenario where you are testing an API that returns user information. You want to validate that an invalid user ID returns a 404 Not Found error.

```
import static io.restassured.RestAssured.*;

import static org.hamcrest.Matchers.*;


public class ErrorResponseValidation {

   public static void main(String[] args) {

      given()
```

```
        .baseUri("https://api.example.com")
        .basePath("/users")
        .pathParam("userId", 9999)  // Assuming 9999 is an invalid user ID
    .when()
        .get("/{userId}")
    .then()
        .assertThat()
        .statusCode(404)  // Validate that the status code is 404
        .body("error", equalTo("Not Found"))  // Check the error field
        .body("message", containsString("User not found"));  // Check the error message
  }
}
```

Explanation:

- **statusCode(404)**: Validates that the API returns a 404 Not Found status.
- **body("error", equalTo("Not Found"))**: Asserts that the error field in the response contains "Not Found".
- **body("message", containsString("User not found"))**: Ensures the message field in the response includes the phrase "User not found".

9. What is Latency in API Testing?

Latency refers to the time it takes for a request to travel from the client to the server and for the server to send a response back to the client. In API testing, measuring latency is important for evaluating the performance and responsiveness of the API.

Technical Details:

- **Latency Measurement**: Latency is typically measured in milliseconds (ms). It includes the time taken for the request to be processed by the server and for the response to be transmitted back to the client.
- **Impact of Latency**: High latency can lead to slow response times, which can negatively affect user experience, especially in real-time applications.

Example to Measure Latency:

You can measure the latency of an API endpoint using RestAssured as follows:

```
import io.restassured.response.Response;
import static io.restassured.RestAssured.*;


public class LatencyCheck {
   public static void main(String[] args) {
      // Send a GET request to the API
      Response response =
         given()
            .baseUri("https://api.example.com")
            .basePath("/data")
         .when()
            .get("/info");


      // Extract and print the latency (response time)
      long latency = response.getTime();
      System.out.println("Latency: " + latency + " ms");
   }
}
```

Explanation:

- **response.getTime()**: This method returns the time taken (in milliseconds) for the entire round trip of the request and response.
- **Output**: The program prints the latency to the console, helping you assess the performance of the API.

10. What are Different Types of Protocols Used in API Testing?

APIs can be built using different communication protocols, each with its own use cases, advantages, and limitations.

Common Protocols in API Testing:

1. **HTTP/HTTPS (Hypertext Transfer Protocol/Secure):**
    - The most widely used protocol for web APIs.
    - **HTTP** is an application layer protocol for transmitting hypertext.
    - **HTTPS** is HTTP with encryption, using SSL/TLS for secure communication.
    - Used for RESTful services.
2. **SOAP (Simple Object Access Protocol):**
    - A protocol for exchanging structured information in web services.
    - Relies on XML-based messaging.
    - Known for its robustness and security features.
    - Commonly used in enterprise applications.
3. **gRPC (gRPC Remote Procedure Calls):**
    - A high-performance, open-source RPC framework.
    - Uses HTTP/2 for transport, Protocol Buffers (Protobuf) for serialization.
    - Ideal for microservices communication due to its efficiency.
4. **MQTT (Message Queuing Telemetry Transport):**
    - A lightweight messaging protocol often used in IoT (Internet of Things) applications.
    - Works on a publish-subscribe model, efficient for resource-constrained devices.
5. **AMQP (Advanced Message Queuing Protocol):**
    - A protocol for message-oriented middleware.
    - Provides message queuing, publish/subscribe, and routing capabilities.
    - Commonly used in systems requiring high reliability and speed.

11. What Types of Testing Can We Perform in API Testing?

API testing encompasses several types of tests, each focusing on different aspects of the API's behavior and performance.

Types of Testing in API Testing:

1. **Functional Testing:**
    - Verifies that the API functions as expected.

- Checks if the API correctly processes inputs and returns the correct outputs.
- Example: Sending a valid request to an API and ensuring the correct response is received.

2. **Integration Testing:**
    - Ensures that different components of an application work together as expected.
    - Involves testing APIs in conjunction with other services or APIs.
    - Example: Testing a payment API integrated with a third-party service.
3. **Performance Testing:**
    - Assesses the API's responsiveness, stability, and throughput under load.
    - Includes **Load Testing** (normal load) and **Stress Testing** (beyond normal load).
    - Example: Sending a large number of requests to an API to see how it handles the load.
4. **Security Testing:**
    - Ensures the API is secure against threats such as unauthorized access, data leaks, and SQL injection.
    - Involves testing authentication, authorization, data encryption, and other security features.
    - Example: Attempting to access protected resources without proper authentication.
5. **Negative Testing:**
    - Validates how the API handles invalid inputs or unexpected conditions.
    - Example: Sending a request with missing required parameters and ensuring the API returns an appropriate error.
6. **Boundary Testing:**
    - Tests the limits or boundaries of the API's input values.
    - Example: Testing the maximum and minimum values for numeric inputs.

12. What is API Testing?

API Testing is the process of testing Application Programming Interfaces (APIs) directly to ensure they meet functionality, reliability, performance, and security expectations. Unlike UI testing,

which focuses on the graphical user interface, API testing involves testing the backend logic and business rules implemented in the API.

Key Aspects of API Testing:

1. **Validation of Endpoints:**
    - Testing whether the API endpoints return the expected results for various inputs.
    - Ensures the correct HTTP status codes, headers, and body content are returned.
2. **Data Validation:**
    - Verifying that the API processes and returns the correct data.
    - This includes validating response data types, structure, and content.
3. **Response Validation:**
    - Checking the structure, content, and status codes of the API responses.
    - Ensures that the response conforms to the expected schema.
4. **Performance Checks:**
    - Assessing the API's speed, stability, and scalability under various conditions.
    - Involves measuring metrics like response time, throughput, and error rates.
5. **Security Verification:**
    - Testing for vulnerabilities like SQL injection, cross-site scripting, and data exposure.
    - Ensuring that the API enforces proper authentication and authorization mechanisms.

Example of API Testing Using RestAssured:

```
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;


public class SimpleAPITest {
   public static void main(String[] args) {
      given()
         .baseUri("https://api.example.com")
```

```
        .basePath("/users")
    .when()
        .get("/123")
    .then()
        .statusCode(200)  // Validate that the status code is 200 (OK)
        .body("id", equalTo(123))  // Validate that the "id" field is 123
        .body("name", equalTo("John Doe"))  // Validate that the "name" field is "John Doe"
        .body("email", equalTo("john.doe@example.com"));  // Validate the email field
  }
}
```

Explanation:

- **statusCode(200)**: Verifies that the API returns a 200 OK status, indicating a successful request.
- **body("id", equalTo(123))**: Ensures that the id field in the response matches the expected value 123.
- **body("name", equalTo("John Doe"))**: Checks that the name field in the response is "John Doe".
- **body("email", equalTo("john.doe@example.com"))**: Confirms that the email field contains the correct email address.

This example demonstrates a basic API test where the expected output is validated against the actual API response.

13.Why Do We Go for API Automation?

API automation is crucial in modern software development for several reasons:

1. **Speed and Efficiency**:
    - **Rapid Feedback**: API automation provides quick feedback to developers about the functionality of their code. Manual testing can be time-consuming, especially with complex APIs.
    - **Continuous Integration/Continuous Deployment (CI/CD)**: Automated API tests can be integrated into the CI/CD pipeline, allowing tests to run automatically

whenever code is pushed or deployed. This ensures that any issues are detected early in the development process.

2. **Consistency and Reliability**:
    - **Elimination of Human Error**: Automated tests reduce the risk of human errors that can occur during manual testing.
    - **Reusability**: Once written, automated tests can be reused across different builds, saving time and ensuring consistent results.
3. **Coverage**:
    - **Broader Test Coverage**: Automation allows for more extensive test coverage, including edge cases and negative scenarios that might be missed during manual testing.
    - **Parallel Testing**: API tests can be run in parallel, covering multiple scenarios simultaneously, which is not feasible with manual testing.
4. **Cost-Effectiveness**:
    - **Reduced Manual Effort**: Once automated, tests can be executed multiple times at no additional cost, reducing the manual effort required to test the application.
    - **Long-Term Savings**: Though there is an initial investment in creating automated tests, the long-term savings in time and resources are significant.
5. **Scalability**:
    - **Handling Large Volume of Requests**: Automated tests can simulate a large number of API requests, allowing for performance and load testing that is difficult to achieve manually.
    - **Integration with Other Systems**: APIs often interact with multiple systems. Automated tests can verify these interactions efficiently.

Advantages of API Automation

1. **Faster Execution**:
    - Automated tests execute much faster than manual testing. They can be run in minutes, whereas manual testing might take hours.
2. **Early Bug Detection**:
    - Bugs can be detected early in the development cycle, reducing the cost and effort required to fix them later.

3. **Improved Test Coverage**:
    - Automation allows for comprehensive testing, including edge cases, which improves the overall quality of the software.
4. **Efficiency in Regression Testing**:
    - Automated tests are perfect for regression testing, ensuring that new code changes do not break existing functionality.
5. **Repeatability**:
    - Automated tests can be executed multiple times with consistent results, ensuring the reliability of the application over time.

Programming Example in Java Rest Assured

Let's create a simple API automation example using Rest Assured in Java. This example will demonstrate how to automate a GET request to verify the response status code and a specific field in the JSON response.

```
import io.restassured.RestAssured;

import io.restassured.response.Response;


import static io.restassured.RestAssured.*;

import static org.hamcrest.Matchers.*;


public class ApiAutomationExample {


   public static void main(String[] args) {


      // Base URI for the API

      RestAssured.baseURI = "https://jsonplaceholder.typicode.com";


      // Automated GET request to verify status code and a specific field in the response

      Response response = given()

                        .header("Content-Type", "application/json")
```

```
                .when()
                    .get("/posts/1")
                .then()
                    .assertThat()
                    .statusCode(200)  // Verify status code
                    .body("userId", equalTo(1))  // Verify userId field in response body
                    .body("title", equalTo("sunt aut facere repellat provident occaecati excepturi optio reprehenderit")) // Verify title field
                    .extract()
                    .response();

        // Print the response body
        System.out.println("Response Body: " + response.getBody().asString());
    }
}
```

Explanation:

1. **Base URI**: The base URI (https://jsonplaceholder.typicode.com) is set for the API we are testing.
2. **GET Request**: A GET request is made to the endpoint /posts/1.
3. **Assertions**:
    - statusCode(200): Verifies that the response status code is 200.
    - body("userId", equalTo(1)): Verifies that the userId field in the response is 1.
    - body("title", equalTo("sunt aut facere repellat provident occaecati excepturi optio reprehenderit")): Verifies that the title field matches the expected value.
4. **Response Extraction**: The response is extracted and printed to the console for verification.

This example highlights the simplicity and power of API automation with Rest Assured, enabling rapid and reliable testing of APIs.

14.Programming Example in Java Rest Assured - POST Request ?

```
import io.restassured.RestAssured;
import io.restassured.response.Response;
import org.json.JSONObject;

import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class ApiAutomationPostExample {

    public static void main(String[] args) {

        // Base URI for the API
        RestAssured.baseURI = "https://jsonplaceholder.typicode.com";

        // Create a JSON object for the POST request payload
        JSONObject requestParams = new JSONObject();
        requestParams.put("title", "foo");
        requestParams.put("body", "bar");
        requestParams.put("userId", 1);

        // Automated POST request to verify status code and specific fields in the response
        Response response = given()
                            .header("Content-Type", "application/json")
                            .body(requestParams.toString())  // Attach the JSON payload
                        .when()
                            .post("/posts")
```

```
                .then()
                    .assertThat()
                    .statusCode(201)  // Verify status code for successful POST creation
                    .body("title", equalTo("foo"))  // Verify title field in response body
                    .body("body", equalTo("bar"))  // Verify body field in response body
                    .body("userId", equalTo(1))  // Verify userId field in response body
                    .extract()
                    .response();

        // Print the response body
        System.out.println("Response Body: " + response.getBody().asString());
    }
}
```

Explanation:

1. **Base URI**: The base URI (https://jsonplaceholder.typicode.com) is set for the API.
2. **JSON Object Creation**:
    - A JSONObject is created to represent the request payload.
    - The fields title, body, and userId are added to the JSON object.
3. **POST Request**:
    - The given() method is used to set up the request.
    - header("Content-Type", "application/json"): Specifies that the request body is in JSON format.
    - body(requestParams.toString()): Attaches the JSON payload to the request.
    - post("/posts"): Sends a POST request to the /posts endpoint.
4. **Assertions**:
    - statusCode(201): Verifies that the response status code is 201, indicating successful creation.

- body("title", equalTo("foo")): Verifies that the title field in the response matches the input.
- body("body", equalTo("bar")): Verifies that the body field in the response matches the input.
- body("userId", equalTo(1)): Verifies that the userId field in the response matches the input.

5. **Response Extraction**:
   - The response is extracted and printed to the console for further verification.

This example demonstrates how to automate a POST request using Rest Assured, including setting up the request payload, sending the request, and verifying the response.

15.Programming Example in Java Rest Assured - POST Request with OAuth 2.0 ?

```
import io.restassured.RestAssured;

import io.restassured.response.Response;

import org.json.JSONObject;


import static io.restassured.RestAssured.*;

import static org.hamcrest.Matchers.*;


public class ApiAutomationPostWithOAuth {


   public static void main(String[] args) {


      // Base URI for the API

      RestAssured.baseURI = "https://api.example.com";


      // OAuth 2.0 Token (this should be obtained from your OAuth 2.0 provider)

      String accessToken = "your_oauth2_access_token_here";
```

```
    // Create a JSON object for the POST request payload
    JSONObject requestParams = new JSONObject();
    requestParams.put("name", "John Doe");
    requestParams.put("email", "john.doe@example.com");
    requestParams.put("age", 30);

    // Automated POST request with OAuth 2.0 authentication
    Response response = given()
                .auth()
                .oauth2(accessToken)  // Add OAuth 2.0 token to the request
                .header("Content-Type", "application/json")
                .body(requestParams.toString())  // Attach the JSON payload
            .when()
                .post("/users")
            .then()
                .assertThat()
                .statusCode(201)  // Verify status code for successful POST creation
                .body("name", equalTo("John Doe"))  // Verify name field in response body
                .body("email", equalTo("john.doe@example.com"))  // Verify email field
                .body("age", equalTo(30))  // Verify age field
                .extract()
                .response();

    // Print the response body
    System.out.println("Response Body: " + response.getBody().asString());
  }
}
```

Explanation:

1. **Base URI**: The base URI (https://api.example.com) is set for the API. Replace this with the actual base URL of your API.
2. **OAuth 2.0 Token**:
    - The accessToken variable holds the OAuth 2.0 access token required to authenticate the request. This token is typically obtained via an OAuth 2.0 flow, such as authorization code flow, client credentials flow, etc.
3. **JSON Object Creation**:
    - A JSONObject is created to represent the request payload with fields such as name, email, and age.
4. **POST Request with OAuth 2.0**:
    - The given() method is used to set up the request.
    - auth().oauth2(accessToken): Adds the OAuth 2.0 token to the Authorization header.
    - header("Content-Type", "application/json"): Specifies that the request body is in JSON format.
    - body(requestParams.toString()): Attaches the JSON payload to the request.
    - post("/users"): Sends a POST request to the /users endpoint.
5. **Assertions**:
    - statusCode(201): Verifies that the response status code is 201, indicating successful creation.
    - body("name", equalTo("John Doe")): Verifies that the name field in the response matches the input.
    - body("email", equalTo("john.doe@example.com")): Verifies that the email field matches the input.
    - body("age", equalTo(30)): Verifies that the age field matches the input.
6. **Response Extraction**:
    - The response is extracted and printed to the console for further verification.

Obtaining the OAuth 2.0 Token

Before running this code, ensure you have a valid OAuth 2.0 access token. Typically, you would obtain this token by performing an OAuth 2.0 flow using an authentication server. This flow might involve exchanging a client ID, client secret, and authorization code for an access token.

This example demonstrates how to automate a POST request with OAuth 2.0 authentication using Rest Assured, including setting up the request payload, sending the request, and verifying the response.

16.Handling Response Headers in Rest Assured ?

Question 1: How do you handle response headers in Rest Assured tests?

Answer: In Rest Assured, you can validate response headers using the header() or headers() methods in the then() block. This allows you to check the presence and value of specific headers.

Example:

```
import static io.restassured.RestAssured.*;

import static org.hamcrest.Matchers.*;


public class ResponseHeaderValidation {

   public static void main(String[] args) {

      given()

         .when()

         .get("/endpoint")

         .then()

         .header("Content-Type", "application/json")

         .header("Server", "nginx")

         .header("Content-Length", notNullValue());

   }

}
```

In this example, the test checks that the Content-Type header is application/json, the Server header is nginx, and the Content-Length header is present.

17. What is the purpose of the Matchers class in Rest Assured?

Answer: The Matchers class, part of the org.hamcrest library, provides a wide range of matchers (assertions) that you can use to validate responses in Rest Assured. It includes methods like equalTo(), containsString(), hasSize(), and more, which help in writing expressive and readable validation logic.

Example:

```
import static io.restassured.RestAssured.*;

import static org.hamcrest.Matchers.*;


public class MatchersExample {

   public static void main(String[] args) {

      given()

         .when()

         .get("/endpoint")

         .then()

         .statusCode(200)

         .body("key", equalTo("value"))

         .body("data.size()", greaterThan(0))

         .body("message", containsString("success"));

   }

}
```

Here, the equalTo() matcher checks if the value of key is value, greaterThan(0) checks if the data array has elements, and containsString("success") verifies that the message contains the word "success."

18.How do you perform a POST request with a JSON payload in Rest Assured?

Answer: To send a POST request with a JSON payload, you use the body() method to include the JSON data. You can specify the payload directly as a string or by using a POJO (Plain Old Java Object) that is serialized to JSON.

Example:

```
import static io.restassured.RestAssured.*;


public class PostRequestExample {
    public static void main(String[] args) {
        given()
            .header("Content-Type", "application/json")
            .body("{\"key\": \"value\"}")
            .when()
            .post("/endpoint")
            .then()
            .statusCode(201);
    }
}
```

This example sends a POST request with the JSON payload {"key": "value"} and expects a 201 Created status code.

19.What is the purpose of the config(JsonConfig.jsonConfig()) method in Rest Assured?

Answer: The config(JsonConfig.jsonConfig()) method is used to configure how JSON data is handled during serialization and deserialization in Rest Assured. This is useful when you need to customize the way JSON data is processed, such as setting up a specific date format or controlling how null values are treated.

Example:

```
import static io.restassured.RestAssured.*;
import io.restassured.config.JsonConfig;
import io.restassured.path.json.config.JsonPathConfig;


public class JsonConfigExample {
    public static void main(String[] args) {
        JsonConfig config = JsonConfig.jsonConfig()
```

```
        .numberReturnType(JsonPathConfig.NumberReturnType.BIG_DECIMAL);


    given()
        .config(RestAssured.config().jsonConfig(config))
        .when()
        .get("/endpoint")
        .then()
        .statusCode(200)
        .body("price", equalTo(new BigDecimal("19.99")));
  }
}
```

In this example, the JSON configuration ensures that numbers are returned as BigDecimal, allowing precise handling of monetary values.

20.Explain the purpose of the auth().oauth2AuthorizationCodeFlow() method in Rest Assured ?

Answer: The auth().oauth2AuthorizationCodeFlow() method in Rest Assured is used to handle OAuth 2.0 authentication using the authorization code flow. This method helps automate the process of obtaining an access token from an authorization server and using it to authenticate API requests.

Example:

```
import static io.restassured.RestAssured.*;


public class OAuth2Example {
  public static void main(String[] args) {
    given()
        .auth()
        .oauth2AuthorizationCodeFlow("clientId", "clientSecret", "authorizationUri", "redirectUri", "tokenUri")
        .when()
```

```
        .get("/secured-endpoint")
        .then()
        .statusCode(200);
    }
}
```

In this example, the oauth2AuthorizationCodeFlow method is used to authenticate the request with an OAuth 2.0 secured endpoint.

---

21. How do you extract data using JSONPath with Rest Assured?

Answer: JSONPath is used to extract specific data from JSON responses. You can use the jsonPath() method to retrieve values from the response body.

Example:

```
import static io.restassured.RestAssured.*;
import io.restassured.response.Response;

public class JSONPathExample {
    public static void main(String[] args) {
        Response response = given()
            .when()
            .get("/endpoint");

        String value = response.jsonPath().getString("key");
        System.out.println("Extracted Value: " + value);
    }
}
```

This example extracts the value of key from the JSON response.

22. What are Request Specifications in Rest Assured?

Answer: Request Specifications allow you to define reusable settings (like base URL, headers, authentication) that apply to multiple requests.

Example:

```
import static io.restassured.RestAssured.*;

import io.restassured.specification.RequestSpecification;


public class RequestSpecExample {

   public static void main(String[] args) {

      RequestSpecification requestSpec = given()

         .baseUri("https://api.example.com")

         .header("Content-Type", "application/json");


      given()

         .spec(requestSpec)

         .when()

         .get("/endpoint")

         .then()

         .statusCode(200);

   }

}
```

23. How do you perform query parameter validation in Rest Assured?

Answer: Query parameters can be validated by specifying them in the request and then verifying their effect on the response.

Example:

```
import static io.restassured.RestAssured.*;


public class QueryParamExample {

   public static void main(String[] args) {
```

```
        given()
            .queryParam("name", "test")
            .when()
            .get("/endpoint")
            .then()
            .statusCode(200)
            .body("responseKey", equalTo("expectedValue"));
    }
}
```

24. How do you use Response Specification in Rest Assured?

Answer: Response Specifications are reusable expectations for responses. They can be used to enforce common validation across multiple tests.

Example:

```
import static io.restassured.RestAssured.*;
import io.restassured.specification.ResponseSpecification;
import static org.hamcrest.Matchers.*;

public class ResponseSpecExample {
    public static void main(String[] args) {
        ResponseSpecification responseSpec = expect()
            .statusCode(200)
            .contentType("application/json");

        given()
            .when()
            .get("/endpoint")
            .then()
```

```
        .spec(responseSpec)
        .body("key", equalTo("value"));
    }
}
```

25. How do you handle timeouts in Rest Assured?

Answer: Timeouts can be handled using timeout() methods or by configuring the HttpClient.

Example:

```
import static io.restassured.RestAssured.*;
import io.restassured.config.HttpClientConfig;
import org.apache.http.params.CoreConnectionPNames;

public class TimeoutExample {
    public static void main(String[] args) {
        given()
            .config(RestAssured.config()
                .httpClient(HttpClientConfig.httpClientConfig()
                .setParam(CoreConnectionPNames.CONNECTION_TIMEOUT, 5000)
                .setParam(CoreConnectionPNames.SO_TIMEOUT, 5000)))
            .when()
            .get("/endpoint")
            .then()
            .statusCode(200);
    }
}
```

26. What is the "RootPath" feature in Rest Assured?

Answer: The RootPath feature allows you to define a common path for JSON extraction, making it easier to retrieve values from deeply nested structures.

Example:

```
import static io.restassured.RestAssured.*;
import static io.restassured.path.json.JsonPath.*;

public class RootPathExample {
    public static void main(String[] args) {
        String json = "{ \"store\": { \"book\": [ { \"author\": \"John\" }, { \"author\": \"Doe\" } ] } }";
        with(json).rootPath("store.book");

        List<String> authors = getList("author");
        System.out.println("Authors: " + authors);
    }
}
```

27. How do you handle dynamic values in response validation?

Answer: Dynamic values, such as timestamps, can be validated using matchesPattern() or by ignoring certain fields.

Example:

```
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class DynamicValueExample {
    public static void main(String[] args) {
        given()
            .when()
            .get("/endpoint")
            .then()
            .body("timestamp", matchesPattern("\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}Z"));
```

```
    }
}
```

28. How do you handle nested JSON structures in response validation?

Answer: Nested JSON structures can be handled by specifying the full path to the desired value.

Example:

```
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class NestedJSONExample {
    public static void main(String[] args) {
        given()
            .when()
            .get("/endpoint")
            .then()
            .body("store.book[0].author", equalTo("John"))
            .body("store.book[1].author", equalTo("Doe"));
    }
}
```

29. How can you reuse specifications and expectations in Rest Assured?

Answer: You can reuse request and response specifications by creating them separately and applying them to multiple requests or responses.

Example:

```
import static io.restassured.RestAssured.*;
import io.restassured.specification.RequestSpecification;
import io.restassured.specification.ResponseSpecification;
import static org.hamcrest.Matchers.*;
```

```
public class ReuseSpecExample {
    public static void main(String[] args) {
        RequestSpecification requestSpec = given()
            .baseUri("https://api.example.com")
            .header("Content-Type", "application/json");

        ResponseSpecification responseSpec = expect()
            .statusCode(200)
            .contentType("application/json");

        given()
            .spec(requestSpec)
            .when()
            .get("/endpoint1")
            .then()
            .spec(responseSpec);

        given()
            .spec(requestSpec)
            .when()
            .get("/endpoint2")
            .then()
            .spec(responseSpec);
    }
}
```

30. How do you handle Unicode characters in Rest Assured?

Answer: Unicode characters can be handled directly in Rest Assured, as it supports UTF-8 encoding by default.

Example:

```
import static io.restassured.RestAssured.*;

import static org.hamcrest.Matchers.*;


public class UnicodeExample {

   public static void main(String[] args) {

      given()

         .when()

         .get("/endpoint")

         .then()

         .body("message", equalTo("こんにちは"));

   }

}
```

31. How can you extract response time information using Rest Assured?

Answer: Response time can be extracted using the time() method, which returns the time taken for the request to complete.

Example:

```
import static io.restassured.RestAssured.*;


public class ResponseTimeExample {

   public static void main(String[] args) {

      long responseTime = given()

         .when()

         .get("/endpoint")

         .time();
```

```
        System.out.println("Response Time: " + responseTime + " ms");
    }
}
```

32. What is the purpose of the relaxedHTTPSValidation() method in Rest Assured?

Answer: The relaxedHTTPSValidation() method is used to disable SSL certificate validation, which can be useful for testing APIs in non-production environments where the certificate might not be valid.

Example:

```
import static io.restassured.RestAssured.*;


public class RelaxedHTTPSValidationExample {
    public static void main(String[] args) {
        given()
            .relaxedHTTPSValidation()
            .when()
            .get("https://self-signed.badssl.com/")
            .then()
            .statusCode(200);
    }
}
```

This example allows you to test an endpoint with a self-signed SSL certificate without validation errors.

---

33. Validating the Dynamic Structure of a Gmail API Response :

When dealing with a dynamic JSON response where the number of unread emails can vary, you can use JsonPath in RestAssured to extract and validate the relevant data dynamically.

Example:

```
import io.restassured.RestAssured;
```

```
import io.restassured.path.json.JsonPath;
import org.testng.Assert;

public class GmailAPITest {

    public static void main(String[] args) {
        // Sample API endpoint for unread emails
        String response = RestAssured.given()
                .auth().oauth2("YOUR_ACCESS_TOKEN")
                .get("https://gmail.googleapis.com/gmail/v1/users/me/messages?q=is:unread")
                .asString();

        // Extracting the count of unread emails using JsonPath
        JsonPath jsonPath = new JsonPath(response);
        int unreadEmailCount = jsonPath.getList("messages").size();

        // Validating that the number of unread emails is greater than 0
        Assert.assertTrue(unreadEmailCount > 0, "Unread email count is 0");

        System.out.println("Number of unread emails: " + unreadEmailCount);
    }
}
```

34. Automating OAuth2 Authentication for Gmail APIs

Automating OAuth2 authentication involves obtaining an access token and using it to authorize subsequent requests.

Example:

```
import io.restassured.RestAssured;
```

```
import io.restassured.response.Response;

public class GmailOAuthTest {

    public static void main(String[] args) {
        // Obtain OAuth2 token (this part usually involves a few steps, here it's simplified)
        String accessToken = "YOUR_ACCESS_TOKEN";

        // Use the token to authenticate API requests
        Response response = RestAssured.given()
                .auth().oauth2(accessToken)
                .get("https://gmail.googleapis.com/gmail/v1/users/me/messages");

        // Print response for verification
        System.out.println("Response: " + response.asString());
    }
}
```

35. Testing the Rate Limits for Gmail API

To test rate limits, you can simulate multiple requests in a loop and detect when the API starts returning a 429 Too Many Requests response.

Example:

```
import io.restassured.RestAssured;
import io.restassured.response.Response;

public class GmailRateLimitTest {

    public static void main(String[] args) {
```

```
        String accessToken = "YOUR_ACCESS_TOKEN";
        int requestCount = 0;

        while (true) {
            Response response = RestAssured.given()
                    .auth().oauth2(accessToken)
                    .get("https://gmail.googleapis.com/gmail/v1/users/me/messages");

            requestCount++;

            if (response.getStatusCode() == 429) {
                System.out.println("Rate limit reached after " + requestCount + " requests.");
                break;
            }
        }
    }
}
```

36. Validating the Content of an Email Message

You can use JsonPath to extract specific parts of the email, such as the body, and validate its content.

Example:

```
import io.restassured.RestAssured;
import io.restassured.path.json.JsonPath;
import org.testng.Assert;

public class GmailEmailContentTest {
```

```
    public static void main(String[] args) {
        String accessToken = "YOUR_ACCESS_TOKEN";

        String response = RestAssured.given()
                .auth().oauth2(accessToken)
                .get("https://gmail.googleapis.com/gmail/v1/users/me/messages/{messageId}")
                .asString();

        JsonPath jsonPath = new JsonPath(response);

        // Extracting the email body (assuming it's in plain text)
        String emailBody = jsonPath.getString("payload.body.data");

        // Decoding from Base64 (if necessary)
        byte[] decodedBytes = java.util.Base64.getDecoder().decode(emailBody);
        String decodedBody = new String(decodedBytes);

        // Validating content
        Assert.assertTrue(decodedBody.contains("expected content"), "Email body content validation failed.");
    }
}
```

37. Ensuring Test Resilience to API Changes

To ensure that your tests are resilient to changes like new fields being added, you can use JSON Schema validation.

Example:

```
import io.restassured.RestAssured;
import io.restassured.module.jsv.JsonSchemaValidator;
```

```
import org.testng.annotations.Test;

import java.io.File;

public class GmailSchemaValidationTest {

    @Test
    public void validateGmailApiResponseSchema() {
        String accessToken = "YOUR_ACCESS_TOKEN";

        RestAssured.given()
                .auth().oauth2(accessToken)
                .get("https://gmail.googleapis.com/gmail/v1/users/me/messages")
                .then()
                .assertThat()
                .body(JsonSchemaValidator.matchesJsonSchema(new
File("path/to/your/schema.json")));
    }
}
```

In the schema file (schema.json), you can define the expected structure of the JSON response, allowing your tests to pass even if new fields are added to the response, as long as the existing structure remains intact.

By Mahasweta S.