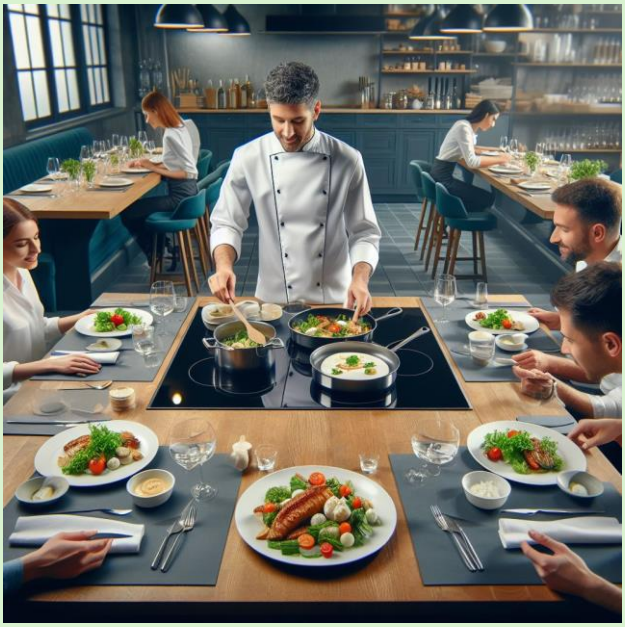
API Testing for SDET, QA and Manual Tester roles

explained by Inder P Singh



Download for reference ⬇️ Like 👍 Share 🔗

YouTube Channel: Software and Testing Training (81,300 Subscribers)

Blog: Software Testing Space (1.77 Million Views)

Chapter 1: Introduction to API Testing 3
Chapter 2: Fundamentals of API Testing 5
Chapter 3: What to Test in API Testing 7
Chapter 4: Writing Effective API Test Cases 10
Chapter 5: Common API Testing Types 13
Chapter 6: API Testing Tools Overview 17
Chapter 7: Examples of API Test Cases 21
Chapter 8: API Testing with Postman 23
Chapter 9: API Testing with SoapUI 25
Chapter 10: API Testing with REST Assured 28
Chapter 11: Common API Testing Challenges and Solutions 33
Chapter 12: API Testing Common Error Codes 36
Chapter 13: Advanced API Testing Techniques 39
Chapter 14: Interview Preparation Tips and Questions 42

Author	Version	Date
Inder P Singh	1.0	31 December 2024

Chapter 1: Introduction to API Testing

Q: What's API testing, and why is it important?

A: API testing means testing Application Programming Interfaces (APIs) to test if they work as expected, meet performance standards, and handle errors. APIs handle the communication between software systems, enabling them to exchange data and functionality. API testing is important for the following reasons:

- **Logic Validation:** APIs can encapsulate the core business logic of an application. API testing finds out if that logic works as intended.
- **Cascading Effect Prevention:** Since APIs often connect multiple systems, a failure in one API can disrupt the entire system. For example, in an e-commerce system, if the API managing payment processing fails, it can prevent order confirmations and impact inventory updates, customer notifications, and financial records.
- **Integration Validation:** APIs handle the interactions between different systems. Testing these interactions for correctness, reliability, performance and security is critical.
- **Early Bug Detection:** By testing APIs before the UI is complete, defects can be identified earlier, reducing downstream issues.

Q: What's the primary focus of API testing?

A: The primary focus areas include:

- **Functionality:** Testing if the API executes intended operations and returns accurate responses. Example: A "getUserDetails" API should return the correct user details based on the provided user ID.
- **Performance:** Validating the API's speed and responsiveness under varying loads. Example: Testing if the API responds within 300 ms when handling 100 simultaneous requests.
- **Security:** Checking data protection, authentication, and authorization mechanisms. Example: Ensuring unauthorized users cannot access restricted endpoints.
- **Reliability:** Confirming if the API delivers consistent results across multiple calls and scenarios. Example: A weather API should always return the correct temperature for a given city.

Q: Is API testing considered functional or non-functional testing type?

A: API testing is generally categorized as **functional testing** because its main objective is to validate if the API performs its expected functions accurately. However, API testing also involves non-functional testing types, depending on the test scope:

- **Performance Testing:** To measure the API's responsiveness and stability under different conditions. Example: Load testing an API that handles ticket booking during a flash sale.
- **Security Testing:** To validate data confidentiality and access control mechanisms. Example: Testing an API for vulnerabilities like SQL injection or unauthorized access.

Q: How does API testing differ from UI testing?

A: API testing focuses on the backend logic, while UI testing validates the user interface. Their differences include:

Aspect	API Testing	UI Testing
Scope	Validates backend systems and business logic.	Tests user interface interactions.
Speed	Faster since it bypasses the graphical interface.	Slower due to rendering processes.
Reliability	API tests are more stable; less prone to flaky results caused by UI changes.	Prone to instability if UI elements change.
Example	Verifying a "createOrder" API works correctly.	Testing if the "Place Order" button functions properly.

Q: Does API testing come under integration testing or system testing test levels?

A: API testing is considered a part of **integration testing** because it validates how different components or systems interact with each other. For example, Testing an API that bridges a payment gateway with an e-commerce platform: The focus would be on testing the correct and complete communication, accurate data exchange, and correct handling of alternate workflows like declined payments.

Q: Can API testing also fall under system testing test level?

A: Yes, API testing can be a part of **system testing** when it is used to validate end-to-end workflows that involve APIs. For example, an order management system involves several APIs for inventory, payment, and customer notification. System testing would involve validating the entire order placement process, including all the APIs in the workflow.

Q: Why is classifying API testing important?

A: Classifying API testing determines the test scope and test approach for testing. For example:

- For **integration testing**, focus on inter-component communication.
- For **system testing**, test the APIs as part of larger workflows to ensure end-to-end functionality.

Chapter 2: Fundamentals of API Testing

Q: What are the key concepts in API testing that you know as an SDET, QA or manual tester?

A: API testing has the following key concepts:

- **Endpoints:** Endpoints are the URLs where APIs are accessed. **Example:** A weather API endpoint might look like `https://api.weather.com/v1/city/temperature`. **Tip:** You should always document endpoints clearly, including required parameters and response formats.
- **Requests and Methods:** APIs use HTTP methods to perform operations. The common ones are:
  1. **GET:** Retrieve data. *Example:* Fetching user details with `GET /user/{id}`.
  2. **POST:** Create new data. *Example:* Creating a new user with `POST /user`.
  3. **PUT:** Update existing data. *Example:* Updating user details with `PUT /user/{id}`.
  4. **DELETE:** Remove data. *Example:* Deleting a user with `DELETE /user/{id}`.

  Tip: Verify that the API strictly adheres to the HTTP method conventions.

- **Request Payloads and Parameters:** APIs often require input parameters or payloads to function correctly:
  1. **Query Parameters:** Added to the URL (e.g., `?userId=123`).
  2. **Body Parameters:** Sent in the request body (e.g., JSON payload for POST requests).
     Tip: Validate edge cases for parameters, such as missing, invalid, or boundary values.
- **Responses and Status Codes:** API responses include data and status codes. Design tests for all possible response scenarios, including success, error, and unexpected responses. Common status codes are:
  1. **200 OK:** Successful request.
  2. **201 Created:** Resource successfully created.
  3. **400 Bad Request:** Client-side error.
  4. **401 Unauthorized:** Missing or invalid authentication.
  5. **500 Internal Server Error:** API failure.
- **Headers:** Headers carry metadata such as authentication tokens, content type, and caching information. *Example:* `Authorization: Bearer <token>` for authenticated APIs. **Tip:** Always validate headers for correctness and completeness.
- **Assertions:** Assertions validate the API's behavior by checking:
  1. **Response Status Codes:** Validate if the expected codes are returned.
  2. **Response Body:** Validate if the response data matches the expected format and content.
  3. **Performance:** Measure if the API responds within acceptable time limits.
     Tip: Use libraries like `REST Assured` or Postman to implement assertions quickly.

Q: Why is API testing important in modern software development?

A: Modern software relies heavily on APIs for communication, making their reliability paramount:

- **APIs Drive Application Functionality:** APIs implement the key features of applications, like user authentication, data retrieval, and payment processing. **Example:** A banking app's core functionalities, such as checking account balances, transferring funds, and viewing transaction history, are implemented with APIs.

- **Integration Testing:** APIs connect multiple systems. Ensuring their proper integration prevents cascading failures. **Example:** In a ride-sharing app, APIs for user location, driver availability, and payment must work together correctly.
- **Early Testing Opportunity:** APIs can be tested as soon as they are developed, even before the UI is ready. This enables early bug detection. **Example:** Testing an e-commerce app's `POST /addToCart` API before the cart UI is finalized.
- **Microservices Architecture:** Applications are composed of multiple independent services connected via APIs. Testing APIs finds out if these services communicate effectively. **Example:** A video streaming platform might use separate APIs for authentication, video delivery, and recommendation engines.
- **Scalability and Performance Assurance:** APIs must be able to handle high traffic and large datasets efficiently. **Example:** During a Black Friday sale, an e-commerce platform's APIs must manage thousands of concurrent users adding items to their carts.
- **Cost Efficiency:** API issues identified early are cheaper to fix than UI-related defects discovered later.

Tips and Tricks for Testers:

- **Use Mock Servers:** Mock APIs allow you to test scenarios without using the ready APIs.
- **Validate Negative Scenarios:** Don't just test happy paths; additionally test invalid inputs, unauthorized access, and server downtime.
- **Automate Tests:** Automating repetitive API tests saves time for test coverage. Tools like REST Assured and Postman can help you automate validations for different test scenarios.

Note: You can follow me in LinkedIn for more practical information in Test Automation and Software Testing at the link, https://www.linkedin.com/in/inderpsingh

Chapter 3: What to Test in API Testing

Q: How do you conduct functional testing of APIs?

A: Functional testing tests if the API performs its intended operations accurately and consistently. It includes the following tests:

- **Endpoint Validation:** Validate if the API endpoints respond to requests as expected. **Example:** Testing if the `GET /user/{id}` endpoint retrieves the correct user details for a given ID.
- **Input Validation:** Test how the API handles various input scenarios:
    - Valid inputs.
    - Invalid inputs (e.g., incorrect data types or missing required fields).
    - Boundary values (e.g., maximum or minimum allowable input sizes).

  Example: Testing an API that accepts a date range to ensure it rejects malformed dates like `32-13-2025`.
- **Business Logic Testing:** Validate that the API implements the defined business rules correctly and completely. **Example:** For an e-commerce API, ensure the `POST /applyCoupon` endpoint allows discounts only on eligible products.
- **Dependency Validation:** Test how APIs interact with other services. **Example:** If an API triggers a payment gateway, test if the API handles responses like success, failure, and timeout correctly.

Tip: Use tools like Postman to design and execute functional test cases effectively. Automate repetitive tests with libraries like REST Assured for scalability.

Q: What do you validate in API responses?

A: Validating API responses involves validating the accuracy, structure, and completeness of the data returned by the API.

- **Status Codes:** Confirm that the correct HTTP status codes are returned for each scenario.
    - **200 OK:** For successful requests.
    - **404 Not Found:** When the requested resource does not exist.
    - **500 Internal Server Error:** For server-side failures.
- **Response Body:** Validate the structure and data types. **Example:** If the API returns user details, validate if the response contains fields like `name`, `email`, and `age` with the correct types (e.g., string, string, and integer).
- **Schema Validation:** Check if the API response matches the expected schema **Tip:** Use schema validation tools like JSON Schema Validator to automate this process.

- **Data Accuracy:** Test if the API returns correct and expected data. **Example:** Testing the `GET /product/{id}` endpoint to verify that the `price` field matches the database record for the product.
- **Error Messages:** Validate that error responses are descriptive, consistent, and secure. **Example:** If a required parameter is missing, the API should return a clear error like `"Error: Missing parameter 'email'"`.

Tip: Include assertions for all fields in the response to avoid missed validations during regression testing.

Q: How do you perform security testing for APIs?

A: Security testing focuses on protecting APIs from unauthorized access, data breaches, and malicious attacks. Key test scenarios include:

- **Authentication and Authorization:** Test if the API enforces authentication (e.g., OAuth, API keys). Verify role-based access control (RBAC). **Example:** A `DELETE /user/{id}` endpoint should only be accessible to administrators.
- **Input Sanitization:** Check for vulnerabilities like SQL injection or cross-site scripting (XSS). **Example:** Test input fields by submitting malicious payloads like `'; DROP TABLE users;--` to confirm if they are sanitized.
- **Data Encryption:** Test that sensitive data is encrypted during transmission (e.g., via HTTPS). **Example:** Check if login credentials sent in a `POST /login` request are transmitted securely over HTTPS.
- **Rate Limiting:** Validate that the API enforces rate limits to prevent abuse. **Example:** A public API should reject excessive requests from the same IP with a `429 Too Many Requests` response.
- **Token Expiry and Revocation:** Test how the API handles expired or revoked authentication tokens. **Example:** Test that a revoked token results in a `401 Unauthorized` response.

Tip: Use tools like OWASP ZAP and Burp Suite to perform comprehensive API security testing.

Q: What aspects of API performance do you test?

A: The API performance testing evaluates the speed, scalability, and reliability of APIs under various conditions.

- **Response Time:** Measure how quickly the API responds to requests. **Example:** For a weather API, test if the response time for `GET /currentWeather` is under 200ms.
- **Load Testing:** Test the API's behavior under normal and peak load conditions. **Example:** Simulate 100 concurrent users hitting the `POST /login` endpoint to verify stability.
- **Stress Testing:** Determine the API's breaking point by testing it under extreme conditions. **Example:** Gradually increase the number of requests to an API until it fails to identify its maximum capacity.
- **Spike Testing:** Validate the API's ability to handle sudden traffic surges. **Example:** Simulate a flash sale scenario for an e-commerce API.
- **Resource Usage:** Monitor server resource usage (CPU, memory) during API tests. **Example:** Confirm that the API doesn't consume excessive memory during a batch operation like `POST /uploadBulkData`.
- **Caching Mechanisms:** Test if the API effectively uses caching to improve response times. **Example:** Validate if frequently requested resources like product images are served from the cache.

Tip: Use tools like JMeter and Gatling for automated performance testing. Monitor metrics like latency, throughput, and error rates to identify bottlenecks.

Note: In order to expand your professional network and share opportunities with each other, you're welcome to connect with me (Inder P Singh) in LinkedIn at https://www.linkedin.com/in/inderpsingh

Chapter 4: Writing Effective API Test Cases

Q: What best practices for writing API test cases do you follow?

A: Writing effective API test cases needs a methodical approach. Here are some best practices:

- **Understand the API Specification:** Study the API documentation, including endpoint definitions, request/response formats, and authentication mechanisms. **Example:** For a `GET /user/{id}` API, understand its parameters (`id`), response structure, and expected error codes.
- **Identify Test Scenarios:** Convert the API's functionality into testable scenarios:
    - Positive test cases: Validate the expected behavior for valid inputs.
    - Negative test cases: Test if the API handles invalid inputs gracefully.
    - Edge cases: Test boundary values to identify vulnerabilities.

  Example: For a pagination API, test scenarios include valid page numbers, invalid page numbers (negative values), and boundary values (e.g., maximum allowed page).

- **Use a Modular Approach:** Create reusable test scripts for common actions like authentication or header validation. **Example:** Write a reusable function to generate a valid authorization token for secure APIs.
- **Use Assertions:** Verify key aspects like status codes, response time, response structure, and data accuracy. **Example:** Assert that the response time for `GET /products` is under 200ms.
- **Automate Wherever Possible:** Use tools like REST Assured or Postman to automate test case execution for scalability and efficiency. **Example:** Automate regression tests for frequently changing APIs to minimize manual effort.
- Prioritize test cases based on business impact and API complexity. High-priority features should have extensive test coverage.

Q: How do you define inputs and expected outputs for API test cases?

A: Inputs:

- Define the parameters required by the API.
    - **Mandatory Parameters:** Verify that all required fields are provided.
    - **Optional Parameters:** Test the API behavior when optional parameters are included or excluded.
- Test with various input types:

    - Valid inputs: Proper data types and formats.
    - Invalid inputs: Incorrect data types, missing fields, and null values.
- **Example:** For a `POST /createUser` API, inputs may include:

```
{
 "name": "John Doe",
 "email": "john.doe@example.com",
 "age": 30
}
```

Expected Outputs:

- Define the expected API responses for various scenarios:
    - **Status Codes:** Verify that the API returns correct HTTP status codes for each scenario (e.g., 200 for success, 400 for bad request).
    - **Response Data:** Specify the structure and values of the response body.
    - **Headers:** Verify essential headers like `Content-Type` and `Authorization`.
- **Example:** For the `POST /createUser` API, the expected output for valid inputs might be:

```
{
 "id": 101,
 "message": "User created successfully."
}
```

Q: What's a well-structured API test case template?

A: A structured template enables writing test cases that are complete, reusable, and easy to understand. Below is a suggested template:

Test Case Template:

Test Case Template:

Field	Description
Test Case ID	Unique identifier for the test case.
API Endpoint	Specify the API endpoint being tested (e.g., `GET /user/{id}`).
Test Scenario	Brief description of the test objective.
Preconditions	Any setup required before executing the test (e.g., authentication tokens).
Test Data	Inputs required for the test case (e.g., query parameters or request body).
Test Steps	Detailed steps to execute the test.
Expected Result	Describe the expected API response (status codes, response body, etc.).
Actual Result	Record the actual response received during testing.
Status	Pass/Fail result of the test case.
Remarks	Any additional observations or notes.

You can use tools like Excel, Jira, or test management software to document and track test cases systematically.

Chapter 5: Common API Testing Types

Q: What's Functional Testing in API testing?

A: Functional Testing validates if the API meets its specified functionality and produces the correct output for given inputs. It tests if the API behaves as expected under normal and edge-case scenarios.

Key Aspects to Test:

- **Validation of Endpoints:** Test that each endpoint performs its intended functionality. **Example:** A `GET /user/{id}` API should fetch user details corresponding to the provided ID.

- **Input Parameters:** Test required, optional, and invalid parameters. **Example:** For a POST `/login` API, validate behavior when required parameters like `username` or `password` are missing.
- **Response Validations:** Verify the response codes, headers, and body. **Example:** Assert that `Content-Type` is `application/json` for API responses.

Tips for API Functional Testing:

- Use data-driven testing to validate multiple input combinations.
- Automate functional tests with tools like REST Assured or Postman for efficiency.

Q: What's API Load Testing?

A: Load Testing assesses the API's performance under normal and high traffic to test if it handles expected user loads without degradation.

Steps to Perform Load Testing:

- **Set the Benchmark:** According to the API performance requirements, define the expected number of concurrent users or requests per second. **Example:** An e-commerce API might need to handle 500 concurrent product searches.
- **Simulate the Load:** Use tools like JMeter or Locust to generate virtual users. **Example:** Simulate 200 users simultaneously accessing the `GET /products` endpoint.
- **Monitor Performance Metrics:** Track response time, throughput, and server resource utilization. **Example:** Verify that response time stays below 1 second and CPU usage remains under 80%.

Common Issues Identified:

- Slow response times due to inefficient database queries.
- Server crashes under high load.

Tips for Load Testing:

- Test with both expected and peak traffic to prepare for usage spikes.
- Use realistic data to simulate production-like scenarios.

Q: Why is Security Testing important for APIs?

A: APIs can be targets for malicious attacks, so Security Testing tests if they are protected against vulnerabilities and unauthorized access.

Important Security Tests:

- **Authentication and Authorization:** Verify secure implementation of mechanisms like OAuth2 or API keys. **Example:** Ensure a user with `user` role cannot access `admin`-level resources.
- **Input Validation:** Check for injection vulnerabilities like SQL injection or XML External Entity (XXE) attacks. **Example:** Test the API with malicious payloads such as `"' OR 1=1--"`.
- **Encryption and Data Privacy:** Validate that sensitive data is encrypted during transit using HTTPS. **Example:** Ensure `Authorization` headers are not logged or exposed.
- **Rate Limiting and Throttling:** Test whether APIs restrict the number of requests to prevent abuse. **Example:** A `GET /data` endpoint should return a `429 Too Many Requests` error after exceeding the request limit.

Tips for Security Testing:

- Use tools like OWASP ZAP and Burp Suite for vulnerability scanning.
- Include both manual and automated security tests for more test coverage.

Q: What's Interoperability Testing in API testing?

A: Interoperability Testing tests if the API work correctly with other systems, platforms, and applications.

Steps to Perform Interoperability Testing:

- **Validate Protocol Compatibility:** Check API compatibility across HTTP/HTTPS, SOAP, or gRPC protocols. **Example:** Test that a REST API supports both JSON and XML response formats, if required.

- **Integration Scenarios:** Test interactions between APIs and third-party services. **Example:** Verify that a payment API integrates correctly with a third-party gateway like Stripe.
- **Cross-Platform Testing:** Test API accessibility across different operating systems, browsers, or devices. **Example:** Verify that the API has consistent behavior when accessed via Windows, Linux, or macOS.

Common Issues:

- Inconsistent response formats between systems.
- Compatibility issues due to different versions of an API.

Tips for Interoperability Testing:

- Use mock servers to simulate third-party APIs during testing.
- Validate response handling for various supported data formats (e.g., JSON, XML).

Q: What's Contract Testing in API testing?

A: Contract Testing tests if the API adheres to agreed-upon specifications between providers (backend developers) and consumers (frontend developers or external systems).

Steps to Perform Contract Testing:

- **Define the Contract:** Use specifications like OpenAPI (Swagger) to document expected request/response structures. **Example:** A `GET /users` API contract may specify that `id` is an integer and `name` is a string.
- **Validate Provider Implementation:** Verify the API provider adheres to the defined contract. **Example:** Verify that all fields in the contract are present in the actual API response.
- **Test Consumer Compatibility: Verify** that consumers can successfully interact with the API as per the contract. **Example:** Check that a frontend application can parse and display data from the API correctly.

Common Tools for Contract Testing:

- **PACT:** A widely-used framework for consumer-driven contract testing.
- **Postman:** For validating API responses against schema definitions.

Tips for Contract Testing:

- Treat contracts as living documents and update them for every API change.
- Automate contract testing in CI/CD pipelines to detect issues early.

In order to stay updated and view the latest tutorials, subscribe to my Software and Testing Training channel (340 tutorials) at https://youtube.com/@QA1

Chapter 6: API Testing Tools Overview

Q: What's Postman, and why is it popular for API testing?

A: Postman is a powerful API testing tool that has a user-friendly interface for designing, executing, and automating API test cases. It's widely used because it supports various API types (REST, SOAP, GraphQL) and enables both manual and automated testing.

Features of Postman:

- **Collections and Requests:** Organize test cases into collections for reusability. **Example:** Group all CRUD (meaning Create, Read, Update and Delete) operations (`POST`, `GET`, `PUT`, `DELETE`) for a `user` API in a collection.
- **Environment Management:** Use variables to switch between different environments like `development`, `staging`, and `production`. **Example:** Define `{{base_url}}` for different environments to avoid hardcoding endpoints.
- **Built-in Scripting:** Use JavaScript for pre-request and test scripts to validate API responses. **Example:** Use assertions like `pm.expect(response.status).to.eql(200);`.
- **Automated Testing with Newman:** Run collections programmatically in CI/CD pipelines using Newman, Postman's CLI tool.

Few Best Practices for Using Postman:

- **Use Version Control:** Export and version collections in Git to track changes.
- **Data-Driven Testing:** Use CSV/JSON files for parameterizing tests to cover multiple scenarios. **Example:** Test the `POST /register` API with various user data combinations.
- **Documentation:** Generate API documentation directly from Postman collections for seamless collaboration.

Q: What's SoapUI, and how does it differ from Postman?

A: SoapUI is a comprehensive API testing tool designed for SOAP and REST APIs. Unlike Postman, which is more user-friendly, SoapUI provides advanced features for functional, security, and load testing, making it more suitable for complex enterprise-level APIs.

Steps to Get Started with SoapUI:

- **Install SoapUI:** Download and install the free version (SoapUI Open Source) or the licensed version (ReadyAPI) for advanced features.
- **Create a Project:** Import API specifications like WSDL (for SOAP) or OpenAPI (for REST) to create a test project. **Example:** Load a WSDL file to test a SOAP-based payment processing API.
- **Define Test Steps:** Create test cases with multiple steps such as sending requests, validating responses, and chaining steps **Example:** For a login API, test `POST /login` and validate that the token from the response is used in subsequent API calls.
- **Use Assertions:** Use built-in assertions for validating response status codes, time, and data. **Example:** Check if the `<balance>` field in a SOAP response equals `$1000`.

Advanced Features:

- **Data-Driven Testing:** Integrate external data sources like Excel or databases.
- **Security Testing:** Test for vulnerabilities like SQL injection.
- **Load Testing:** Simulate concurrent users to evaluate API performance.

Best Practices for SoapUI:

- Use Groovy scripting to create custom logic for complex scenarios.

- Automate test execution by integrating SoapUI with Jenkins or other Continuous Integration (CI) tools.
- Check that WSDL or API specifications are always up to date to avoid testing obsolete APIs.

Q: What's REST Assured, and why is it preferred by SDETs?

A: REST Assured is a Java library that simplifies writing automated tests for REST APIs. It integrates with popular testing frameworks like JUnit and TestNG, making it useful for SDETs familiar with Java.

How to Get Started with REST Assured:

- **Set Up REST Assured:** Add the REST Assured dependency in your Maven `pom.xml` or Gradle build file. **Example (Maven):**

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>5.3.0</version>
</dependency>
```

- **Write Basic Tests:** Create a test class and use REST Assured methods to send API requests and validate responses. **Example:**

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;
// connect with me in LinkedIn at https://www.linkedin.com/in/inderpsingh

public class ApiTest {
    @Test
    public void testGetUser() {
        given().
            baseUri("https://api.example.com").
        when().
            get("/user/1").
```

```
        then().
            assertThat().
            statusCode(200).
            body("name", equalTo("John Doe"));
    }
}
```

- **Parameterization:** Use dynamic query or path parameters for flexible testing. **Example:** Pass `userId` dynamically:

```
given().
    pathParam("id", 1).
when().
    get("/user/{id}").
then().
    statusCode(200);
```

- **Chaining Requests:** Chain API calls for end-to-end scenarios. **Example:** Use the token from a login response in subsequent calls.

Why Use REST Assured?

- Combines test case logic and execution in a single programming environment.
- Provides support for validations, including JSON and XML paths.
- Simplifies testing for authentication mechanisms like OAuth2, Basic Auth, etc.

Best Practices for REST Assured:

- **Follow Framework Design Principles:** Integrate REST Assured into a test automation framework for reusability and scalability. Use Page Object Model (POM) for API resources.
- **Log API Requests and Responses:** Enable logging to debug issues during test execution. **Example:**
  `RestAssured.enableLoggingOfRequestAndResponseIfValidationFails();`

Chapter 7: Examples of API Test Cases

Q: What are some examples of common API test cases?

A: Here are examples of API test cases for commonly encountered scenarios:

- **Validation of Response Status Code:** Test that the API returns the correct HTTP status code. Example: For a successful `GET /user/123` request, the status code should be `200`. **Tip:** Include negative test cases like checking for `404` for non-existent resources.
- **Response Time Verification:** Test that the API response time is within the acceptable limit. Example: For `GET /products`, the API should respond in less than `500ms`. **Tip:** Automate response time checks for frequent monitoring.
- **Header Validation:** Test if required headers are present in the API response. Example: Verify the `Content-Type` header is `application/json`. **Tip:** Include test cases where headers like `Authorization` are mandatory.
- **Pagination:** Test that the API returns correct paginated results. Example: For `GET /users?page=2&size=10`, ensure the response contains exactly 10 users from page 2. **Tip:** Validate `totalPages` or `totalItems` fields, if available.
- **Error Messages and Codes:** Test appropriate error codes and messages are returned for invalid inputs. Example: Sending an invalid `userId` should return `400` with the message, "Invalid user ID". **Tip:** Test for edge cases like sending `null` or special characters.

Q: Can you provide sample test cases for authentication and authorization APIs?

A: Authentication and authorization are important components of secure APIs. Below are a few test cases:

- **Positive Case: Valid Login Credentials:** Test that a valid username and password returns a `200` status with a token in the response. Example:
  Request:
  ```
  POST /login
  { "username": "testuser", "password": "password123" }
  ```

Response:
{ "token": "abc123xyz" }
Validate token structure (e.g., length, format, expiration).

- **Negative Case: Invalid Credentials:** Test that the invalid credentials return `401 Unauthorized`. Example:
  Request:
  `{ "username": "testuser", "password": "wrongpass" }`
  Response:
  `{ "error": "Invalid credentials" }`

- **Token Expiry Validation:** Test that expired tokens return `401 Unauthorized` or a similar error. **Tip:** Check token expiration logic by simulating delayed requests.
- **Role-Based Authorization:** Test that users with insufficient permissions are denied access. Example: `Admin` user can `POST /createUser`. `Guest` user attempting the same returns `403 Forbidden`.
- **Logout Validation:** Test that the `POST /logout` endpoint invalidates tokens, preventing further use. Example: After logout, `GET /user` should return `401 Unauthorized`.

Q: What are example test cases for CRUD operations?

A: CRUD operations (`Create`, `Read`, `Update`, `Delete`) are basic in API testing. Below are the examples:

- **Create (POST):** Test Case: Validate successful creation of a resource.
- **Read (GET):** Test Case: Verify fetching an existing resource returns correct details.
- **Update (PUT):** Test Case: Validate updating an existing resource works as expected.
- **Partial Update (PATCH):** Test Case: Confirm PATCH allows partial updates.
- **Delete (DELETE):** Test Case: Validate successful deletion of a resource.
- **Tips for CRUD Testing:**
  - Use mock data for test environments to avoid corrupting production systems.
  - Check database states post-operations for consistency.
  - Validate cascading deletes for related entities.

Want to learn Test Automation, Software Testing and other topics? Take free courses for QA on my Software Testing Space blog at https://inderpsingh.blogspot.com/p/qa-course.html

Chapter 8: API Testing with Postman

Q: How do you set up Postman for API testing?

A: Setting up Postman for API testing is straightforward with the following steps:

- **Download and Install Postman:** Visit Postman's official website and download the application. It's available for Windows, macOS, and Linux.
- **Create an Account:** Sign up for a free account to sync your collections and settings across devices.
- **Set Up a Workspace** Create a new workspace to organize your API projects. Example: Use a dedicated workspace for a specific project like `E-Commerce APIs`.
- **Import API Specifications:** If your API provides a Swagger or OpenAPI specification, import it into Postman. **Tip:** Use the "Import" button to upload JSON/YAML files or provide the URL of the API specification.
- **Set Up Environment Variables:** Define environments (e.g., `Development`, `QA`, `Production`) with variables like `{{baseUrl}}` or `{{authToken}}`.**Example:**
  - Development: `baseUrl = https://dev-api.example.com`
  - Production: `baseUrl = https://api.example.com`

  Tip: Use Postman's built-in "Environment" feature to switch contexts easily.
- **Test API Connectivity:** Send a basic `GET` request to an endpoint using the configured base URL. Example: `GET {{baseUrl}}/health` should return a `200 OK` response.

Q: How can you write test scripts in Postman to automate validations?

A: Postman allows writing test scripts using JavaScript to automate checks on API responses. Below are examples:

- **Basic Assertions:** Example: Validate the response status code:
```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
```
- **Response Body Validation:** Example: Verify a specific field in the response body.
```
pm.test("Verify user name", function () {
    const responseData = pm.response.json();
    pm.expect(responseData.name).to.eql("John Doe");
});
```
- **Header Validation:** Example: Test that the `Content-Type` header is correct.
```
pm.test("Content-Type is JSON", function () {
    pm.response.to.have.header("Content-Type", "application/json");
});
```
- **Dynamic Data Handling:** Example: Extract values from one API response and use them in another request.
```
pm.environment.set("authToken", pm.response.json().token);
```
- **Chaining Requests:** Automate a workflow by chaining multiple API calls. Example: Use the extracted `authToken` in subsequent requests.
- **Error Handling and Debugging:** Use try-catch blocks for complex scripts to handle exceptions gracefully.
```
try {
    // Validation logic
} catch (e) {
    console.error("Error in script: ", e.message);
}
```

Q: How do you create and manage test collections in Postman?

A: Collections in Postman are a way to organize and automate API tests.

- **Creating a New Collection:** Use the "New Collection" button to group related API requests. Example: Create a collection named `User Management APIs` to store endpoints like `GET /users`, `POST /users`, etc.
- **Organizing Requests into Folders:** Use folders within a collection to categorize requests.Example:

    a. Authentication: `POST /login`, `POST /logout`
    b. User Operations: `GET /users`, `PUT /users/{id}`
- **Adding Test Cases to Collections:** For each request in a collection, write test scripts to automate validations.
- **Setting Up Pre-request Scripts:** Use pre-request scripts to handle prerequisites for API calls. Example: Automatically add the `authToken` to request headers.

```
pm.request.headers.add({ key: "Authorization", value: `Bearer ${pm.environment.get("authToken")}` });
```

- **Running Collections:** Use Postman's "Collection Runner" to execute all requests in a collection sequentially Example: Automate the execution of all CRUD operations for user management.
- **Generating Reports:** Use tools like Newman (Postman's command-line tool) to execute collections and generate reports. Command:
  `newman run UserManagementCollection.postman_collection.json -e DevEnvironment.postman_environment.json -r html`

Tips for Effective Test Collections:

- Regularly update collections to reflect API changes.
- Use descriptive names for requests and variables.
- Share collections with your team members for reuse.

Chapter 9: API Testing with SoapUI

Q: What's SoapUI? Why is it used for API testing?

A: SoapUI is a popular tool for functional, performance, and security testing of APIs. It supports both SOAP and RESTful web services. Its features include an user-friendly graphical interface, assertion mechanisms, and the ability to create reusable test cases and automated workflows.

Advantages of SoapUI:

- **Ease of Use:** Its user-friendly GUI simplifies API testing, even for manual testers.

- **Support for SOAP and REST:** Suitable for testing legacy SOAP-based services and modern REST APIs.
- **Advanced Assertions:** Includes out-of-the-box assertions for validating response content, structure, and status codes.
- **Data-Driven Testing:** Allows parameterizing tests using data from external sources like Excel or databases.
- **Integration Support:** Works well with CI/CD tools like Jenkins.

Q: How do you create and manage test cases in SoapUI?

A: Follow these steps to write effective API test cases in SoapUI:

- **Setting Up a New Project:**
  - Open SoapUI and create a new project by selecting **File > New SOAP Project** or **New REST Project**.
  - For REST APIs, enter the base URL of the API to auto-generate endpoints.
  - Example: Use `https://api.example.com` as the base URL for a REST project.
- **Adding Test Suites and Test Cases:**
  - Right-click on the project and select **New TestSuite** to organize your tests.
  - Within the TestSuite, create **TestCases** for individual API endpoints.
- **Configuring Requests:**
  - Add requests to each TestCase by selecting the specific HTTP method (`GET`, `POST`, `PUT`, `DELETE`). Example: Add a `POST /users` request for creating a new user with payload:

```
{
  "name": "Jane Doe",
  "email": "jane.doe@example.com"
}
```

- **Adding Assertions:** Use assertions to validate the API response. Example:
  - **Status Assertion:** Verify the response code is `200`.
  - **JSONPath Assertion:** Validate specific fields in the response:

```
$.email == "jane.doe@example.com"
```

- **Data-Driven Testing:** Parameterize requests with data from external sources.
- **Tips for Writing Effective Test Cases:**

- Group related requests in a single TestSuite for better organization.
- Use descriptive names for TestCases and TestSteps to improve readability.

Q: How do you automate API testing workflows in SoapUI?

A: SoapUI has several features to automate API testing, making it suitable for CI/CD pipelines.

- **Test Steps and Execution Flow:** Add **Test Steps** to a TestCase, such as:
  - HTTP Requests
  - Property Transfers
  - Groovy Scripts (for advanced logic)
  - Example: Chain a `POST /login` request to authenticate and pass the `authToken` to subsequent requests.
- **Groovy Scripting for Custom Logic:**
  - Use Groovy scripts for advanced scenarios like extracting dynamic tokens or performing calculations.
  - Example: Extract an `authToken` and add it to a header for the next request:
    ```
    def jsonResponse = new groovy.json.JsonSlurper().parseText(context.response)
    testRunner.testCase.setPropertyValue("authToken", jsonResponse.token)
    ```
- **Running TestSuites in Bulk:** Use the **Test Runner** feature to execute multiple TestSuites sequentially or in parallel. Example: Test an entire microservices architecture by running all its TestSuites.
- **Integration with CI/CD Pipelines:** Export SoapUI projects as `.xml` files and integrate them with Jenkins or other CI/CD tools. Use the `testrunner` command-line utility to execute tests:
  ```
  testrunner.sh -sTestSuite1 -cTestCase1 -r -f ./results MyProject.xml
  ```
- **Generating Reports:** SoapUI generates detailed test execution reports, including passed/failed test cases and response times. Export reports in formats like HTML or XML for further analysis.

Best Practices for Automation:

- Use reusable properties for dynamic data across TestSuites.

- Schedule automated test runs to catch regressions early.
- Regularly update TestCases to reflect changes in the API.

If you like my document, you're welcome to Follow me or better, Connect with me (Inder P Singh) in LinkedIn at https://www.linkedin.com/in/inderpsingh

Chapter 10: API Testing with REST Assured

Q: What's REST Assured, and why is it popular for API testing?

A: REST Assured is a Java library designed for testing RESTful APIs. It simplifies HTTP request handling and response validation, integrating with existing Java test frameworks like JUnit or TestNG. REST Assured is particularly useful for automating end-to-end API tests, including functional, integration, and regression testing.

Advantages of REST Assured:

- **Fluent API Design:** Uses an intuitive syntax for writing API tests.
- **Built-in Validation:** Includes powerful methods for validating HTTP status codes, response headers, and payloads.
- **Support for JSON and XML:** Effortlessly handles JSONPath and XMLPath expressions.
- **Integration Ready:** Works well with Maven and CI/CD pipelines.
- **Authentication Mechanisms:** Supports OAuth, Basic, and Token-based authentication.

Q: How do you set up REST Assured and write your first test script?

A: Follow these steps to start using REST Assured for API testing:

- **Setting Up REST Assured:**
    - Add REST Assured dependencies to your Maven project.

```xml
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>rest-assured</artifactId>
  <version>4.4.0</version>
  <scope>test</scope>
</dependency>
```

- For TestNG, include the TestNG dependency:

```xml
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>7.5</version>
  <scope>test</scope>
</dependency>
```

- **Writing Your REST Assured Test Script:** Example: Test a GET `/users` endpoint.

```java
import io.restassured.RestAssured;
import io.restassured.response.Response;
import org.testng.Assert;
import org.testng.annotations.Test;

public class APITest {
    @Test
    public void testGetUsers() {
        RestAssured.baseURI = "https://api.example.com";
        Response response = RestAssured
                .given()
                .header("Content-Type", "application/json")
                .get("/users");

        // Validate Status Code
        Assert.assertEquals(response.getStatusCode(), 200);

        // Validate JSON Response
        String email = response.jsonPath().getString("data[0].email");
        Assert.assertEquals(email, "user@example.com");
    }
}
```

- **Components of REST Assured Scripts:**
  - **Base URI:** The API's base URL (e.g., `https://api.example.com`).
  - **HTTP Methods:** Supports all methods like `GET`, `POST`, `PUT`, `DELETE`.
  - **Headers:** Specify content types like `application/json`.
  - **Payloads:** Use `.body()` for sending JSON/XML payloads in `POST` and `PUT` requests.

Q: How do you test a `POST /login` API that requires a JSON payload?

A: Example script:

```java
@Test
public void testLogin() {
    RestAssured.baseURI = "https://api.example.com";
    String requestBody = "{ \"username\": \"testuser\", \"password\": \"password123\" }";

    Response response = RestAssured
            .given()
            .header("Content-Type", "application/json")
            .body(requestBody)
            .post("/login");

    Assert.assertEquals(response.getStatusCode(), 200);

    // Validate Response
    String token = response.jsonPath().getString("token");
    Assert.assertNotNull(token);
}
```

Q: How does REST Assured validate JSON responses?

A: REST Assured supports JSONPath for validating JSON structures and values.

- **Basic Validation with JSONPath:** Example: Test a `GET /products` API.

```java
Response response = RestAssured
        .given()
        .get("/products");

// Validate Status Code
Assert.assertEquals(response.getStatusCode(), 200);

// Extract and Validate JSON Fields
String productName = response.jsonPath().getString("data[0].name");
Assert.assertEquals(productName, "Laptop");
```

- **JSON Assertions:** Validate arrays and complex structures:
  ```
  List<String> productNames =
  response.jsonPath().getList("data.name");
  Assert.assertTrue(productNames.contains("Laptop"));
  /* View the tutorials that you need in my Software and Testing Training
  channel (340 tutorials) */
  ```

Q: How does REST Assured validate XML responses?

A: Use XMLPath for validating XML structures.

- **Basic XML Validation:** Example: Test an API that returns XML.

```
Response response = RestAssured
        .given()
        .header("Accept", "application/xml")
        .get("/books");

// Validate XML Response
String bookTitle = response.xmlPath().getString("bookstore.book[0].title");
Assert.assertEquals(bookTitle, "Effective Java");
```

- **Handling Namespaces:** Include namespaces for XML:

```
Response response = RestAssured
        .given()
        .header("Accept", "application/xml")
        .get("/books");

XmlPath xmlPath = new XmlPath(response.asString()).setRoot("ns:bookstore");
String bookTitle = xmlPath.getString("ns:book[0].ns:title");
Assert.assertEquals(bookTitle, "Effective Java");
```

Best Practices for Validations:

- **Use Assertions:** Always validate status codes, headers, and response times.

- **Parameterize Inputs:** Avoid hardcoding input values to enhance test reusability.
- **Schema Validation:** Use REST Assured's `matchesXSD` or `matchesJSONSchema` methods to validate response structures against schemas.

Chapter 11: Common API Testing Challenges and Solutions

Q: What are some of the common issues encountered during API testing, and how can they be debugged?

A: API testing involves testing interactions between systems, and errors often arise from various layers of the application. Here are the common issues and strategies to debug them:

- **Invalid Endpoints or URLs:**
  - **Issue:** Incorrect API endpoints lead to `404 Not Found` or connection errors.
  - **Solution:** Double-check the API documentation for correct endpoints and verify configurations in the test scripts. Use tools like `Postman` to confirm the endpoint works before automating.
- **Incorrect Headers:**
  - **Issue:** Missing or invalid headers, such as `Content-Type` or `Authorization`, can cause `400 Bad Request` or authentication failures.
  - **Solution:** Add necessary headers explicitly and validate their formats. Debug the request using tools like Postman or network traffic analyzers like Fiddler.
- **Improper Authentication:**
  - **Issue:** Failure in authentication mechanisms like OAuth or API keys results in `401 Unauthorized` or `403 Forbidden`.
  - **Solution:** Ensure the authentication tokens or keys are valid and refreshed when expired. Automate token generation if possible to prevent manual errors.
- **Malformed Payloads:**
  - **Issue:** Invalid JSON/XML payload structures lead to `400 Bad Request` or parsing errors.
  - **Solution:** Validate payloads against schema definitions using tools like `JSON Schema Validator` or `XML Schema Validator`.

- **Intermittent Failures:**
  - **Issue:** Flaky tests due to server timeouts or race conditions.
  - **Solution:** Implement retry logic with exponential backoff in the test scripts. Use appropriate delays between requests to simulate realistic user interactions.
- **Third-party API Failures:**
  - **Issue:** External dependencies might be unavailable or return unexpected responses.
  - **Solution:** Use mock servers to simulate third-party responses during testing. Tools like `WireMock` or `Mockoon` can help isolate testing from external dependencies.

Q: Why is error handling important in API testing, and how should it be implemented?

A: Robust error handling allows your tests to remain reliable and provide meaningful feedback. Ignoring errors or failing to handle edge cases can lead to incomplete testing or missed defects.

Steps for Effective Error Handling:

- **Categorize Errors:** Identify common error types, such as 4xx client errors, 5xx server errors, or connection timeouts. Example: Test scenarios for invalid data inputs, unauthorized access, and server unavailability.
- **Log Detailed Information:** Include request details (method, endpoint, headers, payload) and response information (status code, body) in the test logs. Example:

```
if (response.getStatusCode() != 200) {
    System.out.println("Request Failed: " + response.prettyPrint());
}
```

- **Validate Responses for Error Scenarios:** Write test cases to ensure that APIs return correct error codes and messages for invalid inputs or edge cases. Example: Testing a `POST /users` API with missing required fields:

```
Response response = RestAssured
    .given()
    .header("Content-Type", "application/json")
    .body("{ \"email\": \"\" }")
    .post("/users");

Assert.assertEquals(response.getStatusCode(), 400);
Assert.assertTrue(response.getBody().asString().contains("email is required"));
```

- **Retry Logic for Timeouts:** Implement retry mechanisms for APIs that may get transient failures. Example:

```
int retries = 3;
while (retries > 0) {
    Response response = RestAssured.get("/unstable-api");
    if (response.getStatusCode() == 200) {
        break;
    }
    retries--;
}
Assert.assertEquals(retries > 0, true, "API failed after retries");
```

- **Simulate Failure Scenarios:** Use mocking tools to simulate network errors, latency, or invalid responses.

Q: What are the best practices for maintaining API test suites over time?

A: Maintaining a test suite is needed for the longevity and reliability of your API testing framework. Follow these best practices to achieve a robust and scalable suite:

- **Organize Test Cases:** Group tests logically (e.g., by functionality, endpoints, or priority). Use descriptive names for test cases and methods. Example: Organize tests for authentication, CRUD operations, and edge cases in separate modules.
- **Parameterize Tests:** Avoid hardcoding values like URLs, credentials, or test data. Use configuration files or environment variables. Example: `RestAssured.baseURI = System.getenv("API_BASE_URL");`

- **Implement Reusable Components:** Use helper methods or classes for repetitive tasks like authentication, response parsing, or payload generation. Example:
```
public String getAuthToken() {
    // Logic to fetch token
}
```
- **Version Control and CI/CD Integration:** Store test scripts in version control systems like Git to track changes. Integrate the test suite into CI/CD pipelines to execute tests automatically on each build.
- **Update Tests Regularly:** Revise test cases as APIs evolve. Keep test data and assertions in sync with updated API contracts.
- **Monitor Test Metrics:** Track execution times, failure rates, and skipped tests to identify bottlenecks or flaky tests.
- **Use Mock APIs for Stability:** For unstable environments or third-party APIs, use mock APIs to avoid dependency issues.
- **Implement Assertions Effectively:** Include multiple assertions for validating status codes, headers, and response payloads in a single test.
- **Optimize Test Execution:** Parallelize tests to reduce execution time. Use tools like TestNG for test suite configuration.
- **Document the Test Suite:** Maintain meaningful documentation for each test case, including its purpose, inputs, and expected outcomes.

Chapter 12: API Testing Common Error Codes

Q: What are HTTP status codes, and why are they important in API testing?

A: HTTP status codes are three-digit numbers returned by a server in response to a client's request. They indicate whether the request was successful, encountered an error, or requires further action. Understanding and validating status codes is basic in API testing as they form the foundation of API response validation.

Categories of HTTP Status Codes:

- **1xx (Informational):** Indicate that the request has been received and the process is continuing. Example: `100 Continue`, `101 Switching Protocols`.
- **2xx (Success):** Indicate that the request was successfully received, understood, and accepted. Example:
    i. `200 OK`: Successful request.
    ii. `201 Created`: Resource successfully created.
- **3xx (Redirection):** Indicate further action is needed to complete the request. Example:
    i. `301 Moved Permanently`: Resource has been moved.
    ii. `302 Found`: Resource temporarily located elsewhere.
- **4xx (Client Errors):** Indicate issues caused by the client. Example:
    i. `400 Bad Request`: Invalid syntax or request.
    ii. `401 Unauthorized`: Authentication required.
    iii. `404 Not Found`: Resource not found.
- **5xx (Server Errors):** Indicate issues on the server side. Example:
    i. `500 Internal Server Error`: Generic server error.
    ii. `503 Service Unavailable`: Server temporarily overloaded or under maintenance.

Q: What are the common error codes in APIs, and how can they be handled effectively?

A: Common error codes provide insights into issues and help testers validate the behavior of the API under different scenarios.

- **400 Bad Request:**
    a. **Cause:** Invalid request format or missing required parameters.
    b. **Handling:** Validate input formats, enforce proper schema validation, and include error messages in API responses.
    c. **Example Test:** Send an incomplete payload and verify that the API returns `400` with an appropriate error message.
- **401 Unauthorized:**
    a. **Cause:** Missing or invalid authentication credentials.
    b. **Handling:** Ensure tokens are correctly generated and sent. Automate token refresh processes where applicable.
    c. **Example Test:** Omit the `Authorization` header and verify that the response includes `401 Unauthorized`.

- **403 Forbidden:**
    a. **Cause:** Lack of proper permissions to access the resource.
    b. **Handling:** Implement role-based access control testing to validate user permissions.
    c. **Example Test:** Log in as a user without admin rights and attempt to access restricted endpoints.
- **404 Not Found:**
    a. **Cause:** Invalid endpoint or non-existent resource.
    b. **Handling:** Validate API documentation for correct endpoint paths and provide detailed error responses for invalid resource requests.
    c. **Example Test:** Call a non-existent endpoint and verify that `404` is returned with a descriptive error message.
- **500 Internal Server Error:**
    a. **Cause:** Unhandled exceptions or server misconfigurations.
    b. **Handling:** Monitor logs and implement proper exception handling in the API code. Ensure graceful degradation under unexpected conditions.
    c. **Example Test:** Provide unexpected payload values to trigger server-side validation and confirm that errors are logged but not exposed in the response.
- **503 Service Unavailable:**
    a. **Cause:** Server overload or maintenance.
    b. **Handling:** Implement retry mechanisms in test cases and use mock servers during downtime.
    c. **Example Test:** Test API response when the server is deliberately brought down, ensuring `503` is returned.

Q: How do you design test scenarios to validate API error responses effectively?

A: Error scenarios should be created to test both expected and edge case behaviors. Below are common approaches to designing error validation test scenarios:

- **Invalid Inputs:** Test the API with invalid data types, missing required fields, or exceeding character limits. Example: Sending a string where an integer is expected should return `400 Bad Request`.

- **Unauthorized Access:** Test scenarios for missing, invalid, or expired authentication tokens. Example: Attempt API calls without an `Authorization` header to validate `401 Unauthorized`.
- **Resource Access Errors:** Test with IDs of resources that don't exist or ones the user has no permission to access. Example: Use a non-existent user ID in a `GET /users/{id}` API and validate the `404` response.
- **Rate Limiting:** Simulate multiple requests in a short duration to validate the API's rate-limiting mechanism. Example: Send 100 requests per second to verify the `429 Too Many Requests` response.
- **Boundary Conditions:** Test boundary values for input parameters. Example: For an API accepting age values between `0-100`, test with `-1`, `0`, `100`, and `101` to validate proper error handling.
- **Invalid Endpoints:** Test with incorrect API paths or endpoints. Example: Call `/getdata` instead of `/data` and verify the `404` response.
- **Service Downtime:** Simulate API unavailability by shutting down the service or using a mock. Example: Validate the `503` response when the API is unreachable.
- **Payload Tampering:** Modify payloads during transmission to check for integrity and validation. Example: Tamper with a JSON payload and verify the response is a `400 Bad Request`.

Read Test Automation and QA articles with code on my Software Testing Space blog.

Chapter 13: Advanced API Testing Techniques

Q: What's parameterization in API testing? Why is it important to you?

A: Data parameterization means testing an API with different sets of input test data without hardcoding values in the test scripts. It allows test coverage and validation of the API's response under different conditions. This technique is useful in testing APIs that accept variable inputs, such as query parameters or JSON payloads.

Q: How do you implement parameterization in API testing?

A: Parameterization is implemented using tools and techniques like:

- **Data Files:** Use CSV, Excel, or JSON files to store input data.
- **Tool Support:** Tools like Postman, SoapUI, or REST Assured provide built-in capabilities for data-driven testing.
- **Scripting:** Write scripts in programming languages like Python or Java to loop through test data.

Example in Postman:

- Create a data file (e.g., `data.json`):

```
[
  { "username": "user1", "password": "pass1" },
  { "username": "user2", "password": "pass2" }
]
```

- Use Postman's collection runner to iterate over the data.

Tips for Parameterization:

- Use meaningful test datasets that cover edge test cases.
- Avoid repetitive values unless testing for load.
- Organize test data to separate valid, invalid, and boundary values.

Q: What are API mocking and stubbing? How do these help in API testing?

A: Mocking and stubbing means creating simulated versions of APIs to test client behavior when real APIs are unavailable or incomplete.

- **Mocking:** Simulates an API's functionality with predefined responses.
- **Stubbing:** Returns a fixed response for a particular request without actual processing.

Q: When do you use API mocking or stubbing?

A: Mocking (see SoapUI example here) can be used in the following situations:

- **Dependency Isolation:** If the dependent API is not ready.
- **Cost Constraints:** To avoid charges for invoking external APIs.
- **Error Scenarios:** Simulate specific errors, like `500 Internal Server Error`.

Q: How do you mock or stub APIs?

A:

- **Postman:** Create mock servers with predefined responses.
- **SoapUI:** Use virtual services to simulate APIs.
- **WireMock:** Simulate HTTP-based APIs programmatically.
- **Programmatic Approach:** Write stubs using frameworks like Node.js, Python's Flask, or Java's Spring Boot.

Example Mocking with Postman:

- Go to Postman > Mock Servers > Create Mock Server.
- Define an endpoint and a sample response.
- Replace the base URL in the client application with the mock server's URL.

Tips for Mocking and Stubbing:

- Document the mocked endpoints and responses clearly.
- Use realistic data to replicate production behavior.
- Regularly update mocks to reflect actual API changes.

Q: How does continuous integration (CI) enhance API testing?

A: CI enables the API tests to be automatically executed whenever new code is integrated. It provides immediate feedback on the stability of APIs, reducing the chances of defects escaping to production.

Q: How do you integrate API testing into a CI pipeline?

A: This process typically includes:

1. **Version Control Integration:** Store API test scripts in repositories like GitHub or GitLab.
2. **CI Tools:** Use CI/CD platforms such as Jenkins, GitLab CI, or CircleCI to automate the execution of tests.

3. **Test Execution:** Configure the CI pipeline to trigger API tests on code commits or merges.
4. **Report Generation:** Generate and store detailed test reports for analysis.

Example with Jenkins:

1. Create a Jenkins job.
2. Use tools like Newman (Postman's CLI) or REST Assured to execute API tests.
3. Configure post-build actions to publish results.

Q: What are the components of a robust CI pipeline for API testing?

A: The key components of an API testing pipeline are:

- **Environment Setup:** Use Docker or virtual machines to replicate test environments.
- **Data Management:** Automate the creation and cleanup of test data.
- **Parallel Testing:** Run multiple test suites simultaneously to save time.
- **Alerts and Notifications:** Notify stakeholders of test failures via Slack, email, or other channels.

Tips for CI Integration:

- Use environment variables to store dynamic configurations like URLs or tokens.
- Prioritize smoke tests in the initial stages of the pipeline.
- Execute reliable tests to avoid getting false positives or negatives.

Chapter 14: Interview Preparation Tips and Questions

Q: What are the tips for preparing for successful API testing interviews?

A:

- **Understand Core Concepts:** Be thorough with HTTP methods, status codes, headers, and payload structures. Familiarize yourself with the REST and SOAP architectures.

Learn about API security, including authentication methods like OAuth, JWT, and Basic Auth.

- **Practical Hands-On Practice:** Use tools like Postman, SoapUI, and REST Assured to create and execute API tests. Mock APIs to simulate real-world scenarios. Practice testing common API flows like CRUD operations and error handling.
- **Build Your Communication Skills:** Practice explaining technical concepts concisely and clearly. Be prepared to discuss past experiences, challenges faced, and how you resolved them.
- **Review Your Resume Again:** Ensure your resume highlights relevant skills such as API testing tools, scripting languages, and frameworks. Include specific examples, such as creating automated test scripts for REST APIs or optimizing API response validation.

Q: What are the specific strategies for answering technical questions?

A: Break down questions logically, even if you don't immediately know the answer. Use examples or analogies to explain concepts. Emphasize practical applications of your knowledge in real-world project scenarios.

Q: What are the most frequently asked questions during API testing interviews?

A:

- What's an API?
- Explain the differences between SOAP and REST APIs.
- What HTTP methods are most commonly used in API testing?
- How do you validate an API response?
- What are the common types of API testing?
- Explain how you would test an API for performance.
- How do you handle API errors like `500 Internal Server Error` or `429 Too Many Requests`?
- How do you perform data-driven testing in Postman or REST Assured?
- Explain the role of mocking and stubbing in API testing.
- How would you integrate API testing into a CI/CD pipeline?

Scenario-Based Questions:

- Given an API that fetches user data, write a test case to validate the response structure.
- How would you test an API that is secured with OAuth 2.0?
- What would you do if an API request returns a `401 Unauthorized` error during testing?

Tip: For scenario-based questions, structure your response by explaining:

1. **Test Objective:** What you aim to achieve.
2. **Test Design:** Inputs, execution steps, and expected results.
3. **Validation:** How to confirm if the test passed or failed.

Q: What should be included in a final checklist for API test automation?

A: Verify the following in the API test automation checklist:

- **Setup and Environment:**
  a. Ensure the test environment mirrors production as closely as possible.
  b. Validate API endpoints, request parameters, and authentication methods.
- **Functional Coverage:**
  a. Test all key functionalities, including positive, negative, and edge cases.
  b. Validate request and response structures for consistency with the API documentation.
- **Error Handling:**
  a. Test APIs for expected and unexpected error codes (e.g., `400 Bad Request`, `500 Internal Server Error`).
  b. Validate the error messages for clarity and relevance.
- **Performance and Load Testing:**
  a. Measure response times and identify potential bottlenecks.
  b. Test for concurrency to evaluate the API's scalability.
- **Security Testing:**
  a. Verify authentication and authorization mechanisms.
  b. Test for vulnerabilities like SQL injection or cross-site scripting.
- **Automation and Reporting:**
  a. Create reusable scripts for regression testing.
  b. Set up detailed reports that capture success rates, failed tests, and error logs.

Example Checklist Template:

Category	Checklist Item	Status
Functional Testing	Validated all CRUD operations.	✅
Error Handling	Simulated error responses for invalid requests.	✅
Security	Tested API with invalid tokens and credentials.	✅
Performance Testing	Measured response times under load.	✅

Tip: Maintain a version-controlled repository for test scripts and related documentation to track updates and ensure consistency.

Lastly, I want to explain the meaning of the image on the first page of this document. Like a chef serving multiple different diners, an API server serves multiple API clients 😀 If you want other new practical Test Automation and QA material, you can get it from my LinkedIn posts. Thank you 👍