**API testing** is a type of software testing that verifies the functionality, reliability, performance, and security of Application Programming Interfaces (APIs). APIs allow different software systems to communicate and exchange data with each other. API testing ensures that data flows correctly between systems and that services respond as expected under various conditions.

* **Common HTTP status codes**:

Success (2xx):

* 200 OK: The request was successful, and the server has returned the requested data.
* 201 Created: The request has been fulfilled, and a new resource has been created (e.g., a new user account).
* 204 No Content: The request was successful, but there's no content to return.

Redirection (3xx):

* 301 Moved Permanently: The requested resource has been permanently moved to a new location.
* 302 Found (Temporary Redirect): The requested resource has been temporarily moved to a new location.
* 304 Not Modified: The client's cached version of the resource is still valid, so the server doesn't need to send a new copy.

Client Error (4xx):

* 400 Bad Request: The server cannot process the request due to a client error, like malformed syntax or invalid data.
* 401 Unauthorized: The request requires authentication, and the client is not authorized.
* 403 Forbidden: The client is authenticated, but does not have permission to access the requested resource.
* 404 Not Found: The requested resource could not be found on the server.

Server Error (5xx):

* 500 Internal Server Error: A generic error indicating a problem on the server side.
* 503 Service Unavailable: The server is temporarily unavailable, perhaps due to maintenance or overload.

**What are the common API testing types?**

API testing encompasses various types, including functional, performance, security, integration, and unit testing, each focusing on different aspects of the API's functionality and behavior. These tests help ensure the API operates correctly, efficiently, and securely.

Here's a more detailed look at different API testing types:

1. Functional Testing:

* This type verifies that the API functions as expected according to its specifications.
* It checks if the API returns the correct data, handles different input scenarios, and produces the desired output.
* Examples include testing different input values, error handling, and boundary conditions.

1. Performance Testing:

* This type evaluates how the API performs under various load conditions, including load, stress, and soak testing.
* It measures response time, throughput, and resource utilization to identify potential bottlenecks and performance issues.
* Load testing simulates normal user traffic, while stress testing pushes the API to its limits.

1. Security Testing:

* This type aims to identify vulnerabilities and weaknesses in the API's security mechanisms.
* It includes techniques like penetration testing, fuzzing, and security audits to ensure the API is protected against attacks.
* Examples include testing authentication, authorization, and data protection mechanisms.

1. Integration Testing:

* This type verifies that the API works correctly with other systems and services.
* It ensures that data can be exchanged seamlessly between different components and that the overall system functionality is maintained.
* Examples include testing integrations with databases, payment gateways, or other external APIs.

1. Unit Testing:

* This type focuses on testing individual API methods or functions in isolation.
* It verifies that each component works correctly in isolation, making it easier to debug and maintain the code.
* Unit tests are typically automated and run frequently as part of the development process.

1. Other API Testing Types:

* Load Testing: Evaluates the API's performance under different loads, simulating realistic user scenarios.
* Fuzz Testing: Injects random or invalid data into the API to uncover unexpected errors or vulnerabilities.
* Documentation Testing: Verifies the accuracy and completeness of the API documentation.
* Interoperability Testing: Ensures that the API can communicate and exchange data with other systems or protocols.
* Regression Testing: Ensures that new features or changes don't break existing functionality.

**What are the Limits of API Usage?**

* The limits of API usage (also called API rate limits or usage constraints) are the rules and restrictions set by an API provider to control how often and how much an API
* can be used. These limits help protect the API from misuse, prevent server overload, and ensure fair access for all users.

**🔑** Common API Usage Limits

1. Rate Limiting
   * Restriction on the number of API calls in a given time.
   * Example: Max 1000 requests per hour.
   * Enforced using status code 429 Too Many Requests.
2. Quota Limits
   * Restriction on the total number of API calls allowed over a longer time frame (daily/monthly).
   * Example: Free tier users can make up to 10,000 requests per month.
3. Concurrent Request Limits
   * Limits the number of simultaneous API calls a client can make.
   * Helps prevent performance issues on the server.
4. Payload Size Limit
   * Limits the size of data sent in a request or returned in a response.
   * Example: Max

**✅ What is an API?**

* + API stands for Application Programming Interface.  
    It’s a set of rules that allows different software systems to communicate with each other.
  + Think of it as a waiter in a restaurant:  
    You (the client) ask for food (data), and the waiter (API) delivers it from the kitchen (server).
  + Example:
  + A weather app uses a Weather API to get data like temperature and humidity from a remote server.

**✅ What is an Endpoint?**

* + An Endpoint is a specific URL or address where an API can access a resource or perform an action.
  + It defines what part of the API you are calling.
  + Example:
  + If the API is located at:  
    https://api.weather.com
  + Then an endpoint might be:  
    https://api.weather.com/v1/current?city=Delhi
  + Here:
  + https://api.weather.com = Base URL (the API)
  + /v1/current?city=Delhi = Endpoint (specific action: get current weather for Delhi)

**🔁 Relationship:**

* API = whole service (like a library of functions)
* Endpoint = one specific function/resource you call within that API

visual diagram

API (Base URL)

|

|-- Endpoint 1: /users → Get all users

|-- Endpoint 2: /users/1 → Get user with ID 1

|-- Endpoint 3: /products → Get all products

|-- Endpoint 4: /products/5/reviews → Get reviews for product ID 5

**How do you validate the Endpoints?**

Validating endpoints in API testing means checking whether the endpoint behaves as expected. Here's how to validate endpoints step-by-step:

**✅ Steps to Validate API Endpoints**

* + 1. Check the HTTP Status Code
  + Ensure the API returns the correct response code:
  + 200 OK – Success
  + 201 Created – Resource created
  + 400 Bad Request – Invalid input
  + 401 Unauthorized – Missing/invalid credentials
  + 404 Not Found – Endpoint or resource doesn't exist
  + 500 series – Server errors
  + 2. Validate the Response Body
  + Check if the data returned is correct and in the expected format.
  + Compare it with the API documentation or a sample schema.
  + Example:
  + {
  + "id": 1,
  + "name": "Alice",
  + "email": "alice@example.com"
  + }
  + Validate:
  + Keys (id, name, email) exist
  + Data types are correct (id is a number, email is a string)
  + 3. Verify Response Time
  + Ensure the API responds within acceptable time limits (e.g., under 2 seconds).
  + Tools like Postman, JMeter, or New Relic help measure this.
  + 4. Schema Validation
  + Match the response structure against a JSON Schema.
  + Tools like Postman, Chai (for JavaScript), or RestAssured (for Java) can be used.
  + 5. Test with Valid and Invalid Inputs
  + Send correct data to see if you get expected results.
  + Send incorrect data to ensure the API handles errors properly.
  + 6. Check Authentication and Authorization
  + Ensure that only authenticated users can access protected endpoints.
  + Verify role-based access control (e.g., admin vs user).

How do you validate the Request Methods?

Validating HTTP request methods—such as **GET**, **POST**, **PUT**, and **DELETE**—is crucial in API testing to ensure that each method performs its intended operation correctly and adheres to RESTful principles. Here's a comprehensive guide on how to validate these methods:

**🔍 1. Understand the Purpose of Each HTTP Method**

* **GET**: Retrieve data from the server without causing any side effects.
* **POST**: Submit data to the server to create a new resource.
* **PUT**: Update an existing resource or create it if it doesn't exist.
* **DELETE**: Remove a resource from the server.

Each method should be used appropriately to maintain the integrity of the API and its resources. ([Restful API](https://restfulapi.net/http-methods/?utm_source=chatgpt.com))

**✅ 2. Validation Steps for Each HTTP Method**

**A. GET Method**

* **Status Code**: Expect 200 OK for successful retrieval.
* **Response Body**: Ensure the data returned matches the expected resource.
* **Idempotency**: Multiple identical GET requests should return the same result without side effects.([Testsigma](https://testsigma.com/guides/api-testing/?utm_source=chatgpt.com" \o "API Testing | What It is, How to Test & Examples - Testsigma))

**B. POST Method**

* **Status Code**: Expect 201 Created upon successful creation.
* **Location Header**: Verify that the Location header points to the newly created resource.
* **Response Body**: Check that the response contains the details of the created resource.
* **Non-Idempotency**: Repeated POST requests with the same data may result in multiple resource creations.([Testsigma](https://testsigma.com/guides/api-testing/?utm_source=chatgpt.com" \o "API Testing | What It is, How to Test & Examples - Testsigma))

**C. PUT Method**

* **Status Code**: Expect 200 OK for successful updates or 201 Created if a new resource is created.
* **Idempotency**: Multiple identical PUT requests should produce the same result.
* **Response Body**: Confirm that the resource reflects the updated data.([LinkedIn](https://www.linkedin.com/pulse/validating-get-post-put-delete-responses-restassured-mohith-nayak?utm_source=chatgpt.com), [Testsigma](https://testsigma.com/guides/api-testing/?utm_source=chatgpt.com))

**D. DELETE Method**

* **Status Code**: Expect 200 OK or 204 No Content upon successful deletion.
* **Idempotency**: Multiple DELETE requests for the same resource should have the same effect as a single request.
* **Subsequent GET Request**: A GET request to the deleted resource should return 404 Not Found.

**🧪 3. Tools for Validating HTTP Methods**

* **Postman**: A user-friendly interface to send requests and validate responses.
* **Rest Assured (Java)**: A Java library for testing RESTful APIs.

**DELETE Request Test Case**

* **Objective**: Delete a specific user.
* **Request**: DELETE /users/1
* **Expected Outcome**:
  + Status Code: 204 No Content
  + Subsequent GET /users/1 returns 404 Not Found.

1. **How do you validate the Request Header?**

Validating request headers is a fundamental aspect of API testing, ensuring that clients communicate correctly with the server and that the server handles requests as expected.

**✅ Positive Test Scenarios**

1. **Verify Required Headers:**
   * Ensure that all mandatory headers (e.g., Authorization, Content-Type) are present in the request.
   * Test each required header individually to confirm that the API accepts requests when the header is correctly provided.
2. **Test Optional Headers:**
   * Include optional headers one at a time to observe their impact on the API response.
   * Combine optional headers in various ways to test different scenarios.
3. **Boundary and Equivalence Partitioning:**
   * Use boundary value analysis to test the limits of header values (e.g., maximum length).
   * Apply equivalence partitioning to group header values into valid and invalid classes and test representative values from each class.
4. **Special Characters and Unicode:**
   * Include special characters and Unicode text in header values to ensure the API correctly handles diverse character sets.

**❌ Negative Test Scenarios**

1. **Missing Required Headers:**
   * Omit one required header at a time to verify that the API responds with appropriate error messages or status codes (e.g., 400 Bad Request or 401 Unauthorized).
2. **Invalid or Unsupported Header Values:**
   * Provide incorrect or unsupported values for headers to test the API's validation mechanisms.
   * Examples include invalid tokens in the Authorization header or unsupported media types in the Content-Type header.
3. **Empty Header Values:**
   * Send headers with empty values to check if the API treats them as missing or invalid and responds accordingly.
4. **Unsupported Headers:**
   * Include headers that the API does not recognize or support to observe if they are ignored or cause errors.([Final Round AI](https://www.finalroundai.com/blog/api-testing-interview-questions?utm_source=chatgpt.com))
5. **✅ Setting and Validating Request Headers in REST Assured**

In REST Assured, you can set request headers using the. header () or. headers () methods. While REST Assured doesn't provide built-in assertions for request headers after the request is sent, you can verify that the correct headers are being sent by logging the request details.

**Example:**

import static io.restassured.RestAssured.\*;

import static org.hamcrest.Matchers.\*;

given()

.header("Authorization", "Bearer your\_token\_here")

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

.log().all() // Logs all request details, including headers

.when()

.get("/api/endpoint")

.then()

.statusCode(200);

The .log().all() method will print all the request details to the console, allowing you to manually verify that the correct headers are being sent.

**✅ Using Filters to Inspect Request Headers**

For more advanced scenarios, you can use REST Assured's Filter interface to inspect or manipulate request headers programmatically.

**Example:**

import io.restassured.filter.Filter;

import io.restassured.filter.FilterContext;

import io.restassured.response.Response;

import io.restassured.specification.FilterableRequestSpecification;

import io.restassured.specification.FilterableResponseSpecification;

Filter requestHeaderFilter = new Filter() {

@Override

public Response filter(FilterableRequestSpecification requestSpec,

FilterableResponseSpecification responseSpec,

FilterContext ctx) {

// Access and print request headers

System.out.println("Request Headers: " + requestSpec.getHeaders());

return ctx.next(requestSpec, responseSpec);

}

};

given()

.filter(requestHeaderFilter)

.header("Authorization", "Bearer your\_token\_here")

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

.when()

.get("/api/endpoint")

.then()

.statusCode(200);

This approach allows you to programmatically access and verify the request headers before the request is sent.

**✅ Summary**

* **Setting Headers**: Use .header() for single headers or .headers() with a Map for multiple headers.
* **Verifying Headers**: Use .log().all() to print request details or implement a custom Filter to programmatically access headers.
* **Server Response Validation**: Test server responses to ensure they handle missing or incorrect headers appropriately

1. **✅ Validating Request Body in API Testing**

**🔍 Negative Test Scenarios for POST/PUT Requests**

1. **Invalid Field Values**
   * **Out-of-Bounds Values**: Provide values that exceed the allowed range (e.g., age = -1 or 200).
   * **Unsupported Values**: Use values not accepted by the API (e.g., status = "unknown").
2. **Incorrect Data Types**
   * Send a string where an integer is expected (e.g., age = "twenty").
   * Provide a boolean instead of a string (e.g., name = true).([apidog](https://apidog.com/blog/api-test-cases-for-post-requests/?utm_source=chatgpt.com" \o "API Test Cases for Post Requests - Apidog Blog))
3. **Empty Fields**
   * Assign empty strings to required fields (e.g., name = "").
   * Use empty objects for nested fields (e.g., address = {}).
4. **Null Values**
   * Explicitly set fields to null (e.g., email = null).
   * Omit fields entirely to test default handling.
5. **Missing Required Fields**
   * Exclude mandatory fields from the request body to ensure the API returns appropriate errors.
6. **Redundant Fields**
   * Include fields that are not defined in the API specification to check if they are ignored or cause errors.
7. **Empty Request Body**
   * Send an empty JSON object {} to test how the API handles lack of data.
8. **Malformed JSON/XML**
   * Send improperly formatted JSON or XML to verify that the API returns parsing errors.
9. **No Data Sent**
   * Send a request without any body to ensure the API responds with an error indicating missing data.
10. **Duplicate Resource Creation**
    * Attempt to create a resource that already exists to test how the API handles duplicates.

**🗑️ Negative Test Scenario for DELETE Requests**

* **Deleting Non-Existing Resources**
  + Attempt to delete a resource that doesn't exist to verify that the API returns a 404 Not Found or appropriate error message.