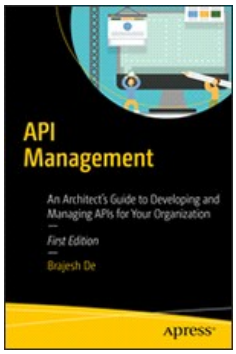

API Management: An Architect's Guide to Developing and Managing APIs for Your Organization

by Brajesh De

Chapter 9: API Testing Strategy

B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_9

Overview

API testing is different from other GUI - based application testing. It tests the programmatic interface that allows access to the data or business logic. Instead of testing the look and feel of the application, API testing concentrates on testing the business logic of the remote software component and its communication mechanism. Hence, API testing is performed using special -purpose software that sends requests to the API and reads the response received. This chapter looks at the importance of API testing, the challenges therein, various testing considerations, and approaches for testing an API.

The Importance of API Testing

An API exposes business data and services through a defined standard - based interface. Developers use the interface to build applications that are dependent on the API. These applications use the API per the defined contract. The application invokes the API with a wide range and combination of data input. The API response depends on the input data and parameter combination. The API traffic pattern also varies by app usage. The API should be able to gracefully handle the traffic at different loads. With all of these permutations and combinations, the importance of API testing greatly increases. The API should be tested against all expected data inputs to validate that it is behaving as defined in the contract. There should also be testing for different load scenarios. API security testing is yet another important aspect.

Challenges in API Testing

API testing is like white -box testing. Testing an API involves special software that sends messages to an API endpoint as defined in the interface, get the output, and log and analyze the response. The test software should programmatically generate messages with different combinations of input parameters to test the API interface and the underlying business logic. The real challenge with testing APIs is automating the test cases. APIs developed for a business may require the execution of multiple APIs in a particular sequence. It could be a fixed sequence or a dynamic sequence based on the result of the previous API. The test execution may also require passing in dynamic values for different input parameters for the API. Some of these parameters' values may need to be derived from an earlier API execution in the test scenario. All of these requirements necessitate the need for an automated API testing approach. It requires tools and framework that help with automating the test scenarios and reduces the execution time for running API tests with different permutations and input parameter combinations.

Often, APIs are exposed via an API management platform configured to implement different policies on top of the API business logic. These policies may implement security, traffic throttling, data transformation, routing, or orchestration. API testing strategy must consider testing these policies in the API gateway.

Testing API traffic management is tricky. APIs might be protected by policies that throttle traffic based on the nature of the consuming app, the time of day, or other parameters, such as location, originating IP, and so forth. The traffic management policies allow a certain number of requests within a given time interval. The interval can be calendar time or a rotating time. Accurately testing API traffic management rules may require the precise coordination of test executions, which can be a challenge. Traffic management policies may also control the number of simultaneous connections to backend services. Ensuring that excess connections are not made to the back-end services and that calls are rejected with proper error messages when the threshold limit is reached are also challenges in API testing.

Test data management is another challenge in API testing. Testing a set of APIs may involve managing a wide range of data sets to cover different permutations and combinations of API input parameters. Managing these data sets in multiple test environments could easily become a challenge. It needs a proper test data management approach to effectively manage the test data. If not managed properly, the test results are erroneous and misleading. For example, if an API key or OAuth access token generated in a SIT environment is used to test APIs in a UAT environment, would result in errors.

The other big challenge with API testing is cultural. Since automation is necessary for API testing, testers need to code to test the APIs. Some of the traditional UI testing is manual in nature. And getting testers with experience in coding may not be always easy. Additionally, testers may need to have knowledge of latest API testing frameworks, such as Chai and Mocha, which adds to the challenges.

API Testing Considerations

APIs provide an interface for communicating with back -end business data and assets. The actual business logic is normally out of the scope of the API implementation. Only some exceptional cases—such as peripheral business logic like data validation—may be implemented in the API layer. Hence, API testing should focus on testing the following aspects of the API:

- API interface specifications
- API documentation
- API security

API Interface Specification Testing

An API interface defines the way to communicate with the API. It defines the input parameters required by the API and the expected response

from the API. The input parameters may be passed as query parameters, in the body as form parameters, or as payload in JSON or XML formats or even Http headers. API testing should validate the API response when the parameters are passed as documented in the specification. If the API specification describes the parameters to be passed in a query parameter, testing should verify the success and error scenarios for the right and wrong combination of parameters and parameter values. For example, consider testing an API that fetches the product details using an API interface (for example, such as at *https://api.foo.com/v1/products*). This API may accept multiple optional query parameters as inputs—including category, name, and SKU, which may be passed as follows:

https://api.foo.com/v1/products?<queryParamName>=<queryParamValue>

Or, for example, *https://api.foo.com/v1/products?category=Electronics*

The approach to test this API should include the following:

- What is the default API behavior when no query parameters are passed?
- What is the API behavior when the right query parameter with the right value is passed?
- What is the API behavior when the parameter name passed is incorrect?
- What is the API behavior when the parameter does not have any value?
- What is the API behavior when the parameter value is incorrect?
- What is the API behavior when multiple query parameters are passed in the right combination?
- What is the API behavior when multiple query parameters are passed in incorrect combinations?
- What is the default data format for the API response when no information about the requested data format is passed?
- What is the data format for the API response for both success and error conditions?
- What is the HTTP response status code for different success and error conditions?
- What is the API response for unexpected HTTP methods, headers, and URLs?

API Documentation Testing

The API test team needs to validate that the API interface documentation is correct and up -to -date. When new versions of APIs are released, the API documentation should be updated to reflect the changes; otherwise, this can cause frustration among the developer community consuming the APIs due to hindering effective adoption. Hence, with every release of a new version of an API, its corresponding documentation should be tested to ensure that it reflect the latest updates. The API rest team should also ensure that the documentation provides enough information to interact with the API. If the API documentation is interactive, it should also be tested to validate proper responses for every API operation.

API Security Testing

APIs provide an access point for business services and data to consumers—internal and external. Depending on the criticality of the data, the API is a point of attack. Hackers may want unauthorized access to system resources for undue benefits. Hackers are always in search of security holes through exposed APIs. There could also be DoS attacks that put an API in an unavailable or unstable state. Hacked APIs can damage the brand value of an enterprise. Hence, testing API security is very important. This section looks at the various aspects of API security testing.

Authentication and Authorization

Testing access mechanism and access control policies of an API is of paramount importace. If an API is exposing a protected resource, the security testing must ensure that only authenticated clients are able to access the APIs. Testing the security policies can include testing API access protected via API Key or Mutual Authentication using PKI or OAuth/OpenID token. Testing OAuth scope validation to ensure that APIs can be accessed using tokens having the right OAuth scope, should form part of the API testing strategy. It is important to validate that right http error codes are returned in case of authentication or authorization failure while accessing an API.

API Fuzzing

API fuzzing is an attack in which the attacker tries to get information about the API and the system resources by sending random input parameters. The attacker sends all possible permutations and combinations of input parameters and analyzes the response in an effort to gain insight into the system resources. The attacker may try to analyze the error messages for various data combinations to understand system behavior. Hence, APIs should be tested with all permutations and combinations of input parameters, and the responses should be analyzed to ensure that the information provided in the responses is appropriate. Error responses provided under different combinations of invalid input data should only provide optimum information as required for the API. It should not reveal information about the internal data structure or database query, file system information, or any other information that can potentially be used to get unauthorized access to the system. For example, for an invalid input to fetch data for an entity, the response should not have any information about the SQL queries that failed to execute in the back end. The error response should indicate only the parameters that are invalid.

Malformed Payload Injection

APIs need to be protected against malformed or unexpected message injection attacks. Very large JSON or XML payloads, JSON payloads with long attribute names or values, and payloads with highly nested structures are used as means for attacking the underlying systems. Processing complex payload structures can take up lot of system resources and CPU cycles. A high volume of such requests may potentially bring down the underlying systems, thus impacting the overall system availability. Hence, API testing should test the API's ability to withstand such vulnerabilities. An API testing strategy should include test cases that test API behavior when a request payload is an unexpectedly large size or has an unexpectedly complex and heavily nested structure.

Malicious Content Injection

Injecting SQL scripts, JavaScript, shell script through input parameters or payloads is also a common form of attack. These scripts, if executed on the server or by a third party, may provide vital information to unauthorized users. The scripts might also be damaging enough to modify or delete data—impacting the business severely. Hence, API testing should test the API behavior when such scripts are injected in the API requests. The API should reject such messages with appropriate error messages. Testing the presence of malicious script should include testing the API behaviour when different types of script like SQL, javascript, shell script, regular expression, XPath, XQuery, python or groovy script are injected throuogh the API payload.

Testing API Gateway Configuration

In many scenarios, a business service is exposed through an API gateway. The API gateway enforces policies in the request and response flow of the API, which may perform one or more of these depending on the requirements: security, throttling, data validation, transformation, routing, error handling, caching, and mashup. Most of these are implemented either using policy blocks or filters within the flow. Unit testing of the API proxy must test the execution of these policy blocks and the conditions applied, if any. Verification methods look at the input and output of each of these policy blocks in the debug trace. If a policy block is to set a local variable or an HTTP header, you can validate whether it is being done properly by looking at the trace output. Similarly, if a policy is supposed to transform the message payload, the output of the policy execution should be the successfully transformed payload. It is also important to look at the average execution time of each of these policy blocks, either in debug logs or in the debug trace of the message flow execution. This can help identify potential performance bottlenecks at an early stage of the testing and reduce efforts to troubleshoot API latency issues at a later stage of performance testing.

API Performance Testing

APIs are no longer seen only as mechanisms for integration but have become mainstream for the delivery of data and services to end users through various digital channels. This increases the demand on APIs to perform well under loads. The overall performance of a client app is dependent on the performance of the underlying APIs powering the app. Hence, the importance of performance and load testing for APIs increases greatly. This section looks at the strategy for load testing an API.

Preparing for the Load Test

It is important to plan well and have a well-defined load testing strategy. Planning for API load testing starts with identifying the list of APIs to be tested. Load testing is most effective when the workload for the API is as close to the real expected traffic. It is not useful to know that an API can handle say 500 transactions per second without knowing whether the real traffic is higher than or lower than that. The first step in preparation for a load test is to gather information on the performance requirements that the APIs are expected to handle. This includes the following information:

- The average throughput in terms of the number of requests per second for each API deployed on the platform
- The peak throughput that projects the maximum number of requests that each API is expected to handle at any given point in time (normally during peak loads)
- Throughput distribution across all the APIs deployed on the platform.
- The traffic distribution patterns of client apps using the API helps predict accurate API usage
- The number of concurrent users expected for each client app using the API, which predicts the total number of concurrent connections that the API platform is expected to handle under load.

Having decided on the performance requirements for API testing, there may be different approaches to actual test execution. Actual test execution can start with the generation of repetitive loads for each of the API endpoints. This establishes the upper bounds of the performance that may be achieved in the test platform. If it is low, you should look at options to tune and optimize the API and platform parameters for a better throughput. Adding hardware or instances in the cluster configuration can be the second option to look at if the results from the repetitive load test are not satisfactory.

Once the platform has stabilized and the upper bounds of load testing have been determined from repetitive load tests, it is time to simulate a realistic traffic pattern. Using a real traffic scenario might be ideal, but not practical for various reasons. The simulated traffic should consider the following:

- Traffic distribution across various deployed APIs. For example, 45% of calls are to product catalog APIs, 35% of calls are to customer information APIs, and 20% of calls are o payment APIs.
- Traffic growth pattern during the day. For example, gradual increase or sudden spikes or a constant load through out.

- API traffic for both success and failure responses.
- Geographically distributed API traffic to test for any network traffic congestion at high loads.

Data from production traffic logs of already deployed services can provide information to simulate realistic traffic scenarios for load testing.

API performance testing should consist of the following, with an aim to find the different performance parameters of the APIs and the API platform.

- **Baseline testing:** The objective of this testing is to find out how the system performs under normal expected load. The results from this test should be used to analyze the average and peak API response time and error rates. The CPU and memory utilization of the platform should also be looked into to eliminate any resource bottlenecks.
- **Load testing:** During load testing the load is increased to study the API performance under growing API traffic volumes. Performance metrics, such as response time, throughput of the APIs should be looked at to review the performance under load. The aim of this testing is not to find the breaking point, but to understand the expected system behavior and capability to handle expected peak loads. Server performance metrics, such as CPU utilization, heap memory utilization, network port utilization should be analyzed to understand the state of the platform and its ability to handle high load.
- **Stress testing:** The goal of stress testing is to find the breaking point of the platform. It is used to determine the maximum throughput that the system can handle. In this form of testing the API traffic load is gradually increased till a breaking point is reached when the performance starts to degrade or errors from API calls start to increase.
- **Soak testing:** Soak testing determines whether there are any system instabilities in long duration testing. The baseline test may be executed over several days or weeks to learn about any unwanted behaviors that may occur when the system is used for a long time. The aim is to discover any issues with releasing system resources and make them available for the next cycle of execution. If system resources are not getting released periodically, there is a high probability of the system crashing under sustained high loads. Normal baseline testing or load testing may not be able to unearth such problems, and then the importance of soak testing increases.

The load test strategy should consider the environments for doing the load test. A preproduction setup that is a replica of the production setup would be an ideal for load test. However, that may not be available all the time due to practical reasons. Hence, a dedicated load test environment that is a scaled down version of the production environment may be used for load and performance testing. Considerations should be made to scale down expected throughput by the same factor while performing the load test.

Setting up for the Load Test

Having identified the environment for the load test and approach, it is time to identify the right set of tools to execute the load test. There are many tools available to perform API load testing. Let's talk about the most commonly used tools for doing the execution testing.

- **JMeter** is an open source Java-based tools with a powerful GUI used to easily simulate non-trivial HTTP requests to test REST APIs. It allows you to model complex workflows using conditions. A test plan in JMeter allows you to define the thread group that is used to simulate end user behavior in terms of the number of concurrent users, the ramp-up time, and the REST API request sent by them. The HTTP request is parameterized. Parameterizing the test requests reuses it with different parameter values and dynamically passes the execution results from one test to another. Assertions are added to validate the test results automatically. Listeners provide widgets that are used to view the test results. JMeter is one of the best open source tools for functional testing used to model complex user flows using conditions. The availability of a large number of community plugins extends the built-in behavior. Its non-GUI-based option runs JMeter for test execution in an environment that does not support rich GUIs, such as Linux-based environments.
- **LoadUI** is a commercial API load-testing tool from SmartBear. LoadUI has an advanced feature that allows you to do distributed load testing by distributing the load tests to any desired number of LoadUI agents. It also allows running multiple test cases simultaneously and long running tests that may run days or weeks.
- **Wrk** provides a command-line interface to test REST APIs. Being multithreaded, it is able to take advantage of the underlying multicore processor; hence, it is used to simulate really high loads. The default reporting format for Wrk is limited to text only, which sometimes makes it difficult to interpret test results easily. However, its ease of use to simulate high loads makes it one of the best tools, when the goal is to find the load than an API can handle.
- **Vegeta** is an open source HTTP load testing tool for performance testing of REST APIs. It is useful when the aim of the testing is to learn how long the service can sustain a constant load of *x* requests per second. This is important when you have data about the peak load that is expected for an API and you want to find out how long the service can sustain that peak load before you start seeing a drop in performance.
- **BlazeMeter** and **Loader.io** are two tools that run the load test for APIs in a cloud platform. They provide load-testing infrastructures as a service in the cloud. The cloud-based approach reduces efforts to set up the environment for load testing. BlazeMeter provides the option to upload a JMeter test plan and run it from its cloud infrastructure.

API Performance Test Metrics

Performance testing of API should look at the following metrics to measure the performance of the individual API and the platform.

- **API response time:** Measures the overall end-to-end response time of the API. Determines the time in which an end user is expected to

get a response from the API. Minimum, average, and peak API response times should be measured as part of API performance testing.

- **API target response time:** Determines the time it takes for the API back-end systems to respond. If an API is exposed through an API gateway, it measures the response time of the target back end for the gateway API proxy.
- **API latency:** Measures the latency introduced by any intermediary, such as an API gateway used in the API architecture.
- **Throughput:** Measures the number of requests processed in a second. Normally, this is measured in transactions per second (TPS).
- **Success and error rates:** These metrics are important for API performance testing. They measure the number of requests successfully processed under load.
- **CPU utilization:** Measures the capacity of the system under load. A low CPU utilization means that the system can handle a higher load. Higher CPU utilization is indicative of a system under stress.
- **Heap memory utilization:** Indicates how system memory is being utilized to process requests under load. The system RAM may need to be increased if heap memory utilization stays at its peak throughout the performance test. Low available memory may impact the overall performance of the APIs.

Selecting The Right API Testing Tool

Having looked at the various aspects of API testing, it becomes important to look at the feature that should be in an API testing tool to make it a success. The following lists can help with selecting the right API testing tools. They cover the features that an API testing tool should have and other nice-to-have features.

Must-Have Features

The following are must-have API testing tool features.

- API test tools should support automated API testing to cover a wide range of scenarios.
  - It should test success conditions with different data combinations
  - It should test error conditions and corner cases
- The automation of functional test cases should support the following and must be repeatable for multiple deployments and environments. The tool must have the following capabilities for API functional testing:
  - It should support the creation of HTTP requests with different combinations of verb, headers, query parameters, and payloads
  - It should support payload generation in multiple data formats (JSON, XML, SOAP) and even binary format
  - It should support automated creation of API request templates by importing WADL, RAML, Swagger, API Blueprint, and so forth
  - It should support automatic data validation for request/response messages based on a defined schema
  - It should support parameterized test creation
  - It should provide the ability to define test flow logic
  - It should support test visualization to understand the failure points of API executions
- Test asset management capabilities
  - It should group and tag test cases
  - It should search test cases and make changes to a group of test cases through find and replace
  - It should easily create new tests or update existing test assets based on changing API demands
  - It should manage test data for different environments
- Security testing capabilities
  - API authentication and authorization using protocols such as OAuth, OpenId, SAML, Basic Authentication, and SAML
  - Message encryption and decryption
  - Penetration attacks, such as SQL/script injection, malformed payload, virus attacks, parameter fuzzing, and so forth
- Performance testing capabilities
  - It should calculate API response times, throughput, and error rates
  - It should simulate regular performance loads

- It should simulate unpredictable and volatile performance load with valid payload
- It should simulate spikes and sudden bursts of traffic in consuming apps

Nice-to-Have Features

The following are some of the nice-to-have features in an API testing tool.

- Record and replay API traffic
- Integration with requirements management and issue tracking systems, such as JIRA and QC
- Integration with CI tools such as Jenkins, Cloud Bees, Cruise Control, and so forth
- Federated and cloud testing ability to execute test cases in a distributed scenario
- The ability to run in non-GUI mode with a command-line interface
- The ability to schedule test cases

Common API Testing Tools

The following are some common API testing tool products.

Unit Testing Tools	Integration Testing Tools	Performance Testing Tools
JUnit	JMeter	JMeter
Curl	SOAPUI	LoadUI
Postman	APICLI	Grinder
Advanced REST Client	Cucumber	Curl-Loader
Mocha	Jasmine	Wrk
Chai	Mocha	Vegeta
TestNq	jBehave	BlazeMeter
QUnit	NSpec	
PyUnit	SpecFlow	
Hurl.it		