ZENQ

EFFECTIVE
API TESTING STRATEGIES

WHITE PAPER

Contents

Overview

Service Oriented Architecture (SOA) makes use of loosely coupled services as a means of supporting the requirements of users and corporate processes. Resources in the SOA environment are made available as independent services that one can access without having knowledge of the underlying platform implementation. SOA is essentially a collection of services that communicate with each other involving either simple data passing or it could involve two or more services coordinating some activity.

The services involved in SOA are **Web services** and the interface used as a communication channel among web services is an **API** (Application Programming Interface).

The most popular ways of developing web service API are - REST (Representational State Transfer protocol) and SOAP (Simple Object Access protocol).

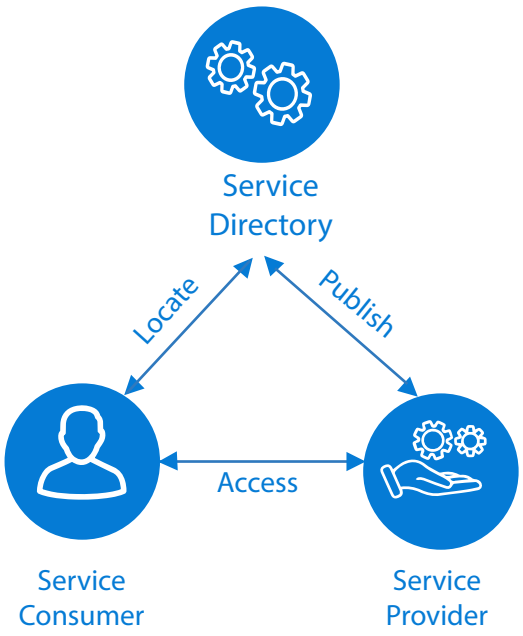


Figure 1: SOA Architecture

Advantages of Web service APIs

- **Loose coupling** – SOA follows loosely coupled approach, which means the application and architecture is split into various services, hence, software can be built with minimal dependencies
- **Flexibility** – Web service API provides the privilege to write various components in any language or platform. Hence, one could write the user interface in any dynamic and productive language like Python/Ruby/JavaScript while critical components can be written in lower level languages like Java or C
- **Easier testing and debugging** – Small and independent components are easy to test and debug that helps bring in efficiency and higher quality
- **Scalability** – Since components are independent, it is much simpler to scale up the components and eventually the architecture. One could easily scale up a particular component and test it in isolation, without affecting other components at all. This makes it easy to add in servers without facing any downtime
- **Reusability** – Since various components are built by assembling small, self-contained and loosely coupled pieces of functionality, it becomes much easier to reuse them as needed
- **Adaptability** – SOA applications are flexible enough to adapt to changing technologies

Why Web service API testing is important

Just like traditional application testing through the UI, applications build on SOA such as web service APIs could also have significant bugs that could undermine the performance and functionality.

Firms build web service APIs to support other internal products or for external third party vendors. Each consumer of the web service API may use it in different ways which could expand the reach among the growing consumers.

Just like any software system, even a single bug in web service API production would cost a lot in terms of maintenance and price required to fix it. Hence, it is imperative that the Web Services API be tested for quality.

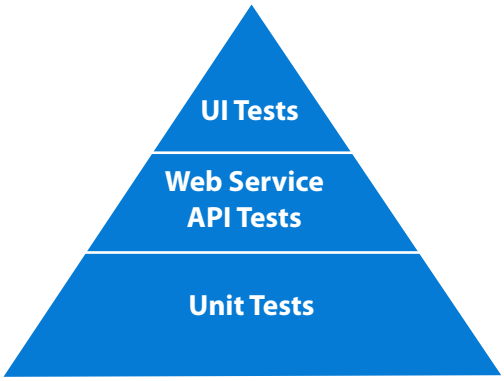


Figure 2: Test Automation Pyramid

If the application is designed in SOA, as per the test automation pyramid, a good testing strategy would invest significant efforts in creating the web service API tests, as compared to the UI tests. These tests are fast in execution, less brittle and can cover as many user scenarios/end-end scenarios as possible.

Challenges in testing Web service API

Like any other application testing, there are challenges which testers face while testing web service APIs -

- Services are atomic, which means building an end-to-end flow involves several API calls in a chain
- Each service could be poorly designed to cover multiple functionalities, leading to slow and error-prone responses
- Services return information that requires processing by the UI thereby defeating the purpose of separation and increasing complexity
- Using middleware/ESB technologies add an additional layer of complexity
- Hashing/salting using MD5, etc., important from the security perspective further adds to the processing needs

Testing strategies

While formulating a Test strategy for API web services, the above challenges should be kept in mind, so as to develop a robust and efficient test strategy.

A detailed web service API documentation is the primary requirement for a tester to be able to outline the testing strategy. The web service API documentation should be able to provide the following information:

- HOST and different END POINTS
- Verbs supported for each END POINT
- Allowed input parameters and field rules
- Expected output and Required fields to be verified in response

The graphic below illustrates the approach used for Web Service API testing -

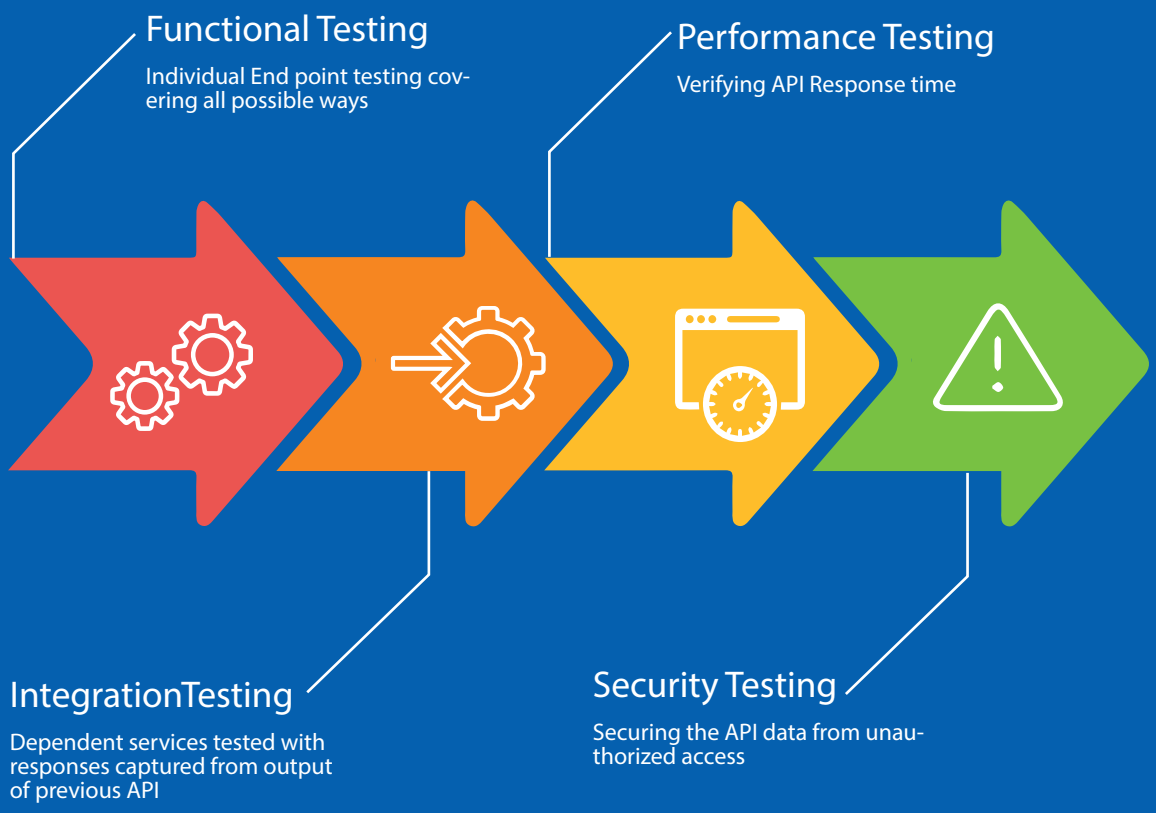


Figure 3: Web service API Testing Approach

1. Functional testing (Individual end point)

Each End point of the web service API could be tested with all possible types of Test Design Techniques.

Test design techniques

- Access and Authentication – Accessing the web service API from different regions and authentication (valid and invalid)

- Varied input set of test data – Checking an endpoint with possible test data in its Body/ Header parameters. The conventional test design techniques such as BVA (Boundary Val ue Analysis) and EQP(Equivalence class parti tion) can be used here

Test verifications

The following could be verified from API testing -

- Response content
- Response code
- Response time
- Response message

Example:

To demonstrate this, let's take an example of 'Gmail' API. This API uses OAuth 2.0 for authorization and REST API to read & send emails and manage labels.

The following Http "Get" request is to get the specific message from user's mailbox with its id.
URI - https://www.googleapis.com/gmail/v1/users
Parameter – API key from Gmail API
Endpoint - /userId/messages/id
GET: https://www.googleapis.com/gmail/v1/users/userId/messages/id?key={YOUR_API_KEY}

Test Data setup

For testing this particular End Point, possible combinations of test data need to be drawn. Here, varied test data can be assigned for 'userId' and message 'id'.

userId – user email address

Test combinations:

- Existing email Id which has been in use for several months
- Existing email Id which is just created
- Existing email Id with no emails
- Non-existing email Id
- Email Id which is expired
- Email id which is deleted
- Email Id which has not been accessed for at least one year
- Invalid email IDs

id – message id (Example: alphanumeric characters of length 16 - 15746ab45e04275a)

- Existing Message Id
- Non-existing Message Id
- Invalid message Id
- Message id in 'Inbox' label
- Message id in 'Spam' label
- Message id in 'Trash' label
- Message id in 'Customized label'
- Message id permanently deleted

Endpoint	User Id	Message id
/userId/messages/id	which has been in use for several months	Valid and Existing
	which is just created	Valid and Existing
	with no emails	Valid and Existing
	Non-existing	Valid and Existing
	which is expired	Valid and Existing
	which has not been accessed for at least one year	Valid and Existing
	Invalid	Valid and Existing
	Valid and existing	Non-existing
	Valid and existing	Invalid
	Valid and existing	Message id in 'Inbox' label
	Valid and existing	Message id in 'Spam' label
	Valid and existing	Message id in 'Trash' label
	Valid and existing	Message id in 'Customized label'
	Valid and existing	Message id permanently deleted

With the above test data, several test cases can be built and testing can be automated using Data-driven testing approach, which not only saves a lot of effort but also makes test maintenance a lot easier.

Functional techniques can be executed once using tools like Postman and further these can also be automated using the HTTPClient library, SoapUI, etc. For the list of more tools please refer to the section
API Testing Tools

Sample code snippet in Java to automate a basic HttpGet method.

- GET method to retrieve the specific email content using a message ID

```
//Create HttpClient instance with credentials
CredentialsProvider provider = new BasicCredentialsProvider();
UsernamePasswordCredentials credentials = new UsernamePasswordCredentials("abc@
gmail.com", "*********");
provider.setCredentials(AuthScope.ANY, credentials);
HttpClient client = HttpClientBuilder.create().
setDefaultCredentialsProvider(provider).build();

//Create method instance
HttpGet get = new HttpGet("https://www.googleapis.com/gmail/v1/users/userId/
messages/{MESSAGE_ID}?key={YOUR_API_KEY}");
```

```
//Adding headers
get.addHeader("Content-Type", "application/json");
//get the response
HttpReponse response;
try {
     response = client.execute(get);
} catch (IOException e) {
     e.printStackTrace();
}
//Retrieving the response body
String responseBody;
try {
    InputStreamReader in = new InputStreamReader(response.getEntity().
getContent());
    BufferedReader br = new BufferedReader(in);String responseLine;
    while ((responseLine = br.readLine()) != null) {
         responseBody = responseBody + responseLine;
      }
    } catch (Exception e) {
         System.out.println(e);
    }
String JsonString = response.toString();
JSONParser parser = new JSONParser();
JSONObject object = null;
try {
    object = (JSONObject) parser.parse(JsonString);
} catch (ParseException e) {
     e.printStackTrace();
}
//Get the JSON object key value and verify
String jsonBodyValue = (String) (object.get(jsonBodyKey.toString()));
Assert.assertEquals(jsonBodyValue, "This is to test GMail API");
```

Data-driven Test

Same API test script can be data-driven by reading multiple sets of data (valid/invalid/existing/nonexisting, etc.) using TestNG 'DataProvider' feature.

```
@DataProvider(name = "GetContactsFromExcel")
public String[][] getMailId_MessageID() {
      String data[][] = {{abc@gmail.com,15f90243b18496b3},
                         {test@test@gmail.com,uuiiolmmvpl,mmk},
                         {yyyy@@test.com,3ref},
                         {$%^@test.com,$%@^&}};
      return data;
}
```

The data can also be retrieved from external files - Excel or CSV, etc.
A TestNG test is created using the above DataProvider that drives the test with iterations equaling set of values declared.

```
@Test(dataProvider="getMailId_MessageID")

public void test(String Email_Id,String message_Id){

        //Create HttpClient instance with credentials

       CredentialsProvider provider = new BasicCredentialsProvider();

       UsernamePasswordCredentials credentials = new
UsernamePasswordCredentials(Email_Id, "*********");
```

```
        provider.setCredentials(AuthScope.ANY, credentials);
        HttpClient client = HttpClientBuilder.create().
setDefaultCredentialsProvider(provider).build();
             //Create method instance
        HttpGet get = new HttpGet("https://www.googleapis.com/gmail/v1/users/userId/
messages/"+message_Id+"?key={YOUR_API_KEY}");
        //Adding headers
        get.addHeader("Content-Type", "application/json");
        //get the response
        try {
             HttpResponse response = client.execute(get);
        } catch (IOException e) {
               e.printStackTrace();
        }
//Retrieving the response body
        try {
            InputStreamReader in = new InputStreamReader(response.getEntity().
getContent());
            BufferedReader br = new BufferedReader(in);String responseLine;
          while ((responseLine = br.readLine()) != null) {
                 responseBody = responseBody + responseLine;
            }
            } catch (Exception e) {
                  System.out.println(e);
            }
            String JsonString = response.toString();
            JSONParser parser = new JSONParser();
            JSONObject object = null;
            try {
                object = (JSONObject) parser.parse(JsonString);
            } catch (ParseException e) {
                  e.printStackTrace();
          }
            //Get the JSON object key value and verify
            String jsonBodyValue = (String) (object.get(jsonBodyKey.toString()));
            Assert.assertEquals(jsonBodyValue,"This is to test GMail API");
        }
}
```

2. Integration testing (end-end)

This approach involves simulating real user behaviour by combining two or more dependent/related web service APIs. The objective of this testing is to verify the accuracy of the data exchange when two or more endpoints interact with each other, eventually resulting in an expected output.

Example

A sample user flow using same Gmail API is explained below.

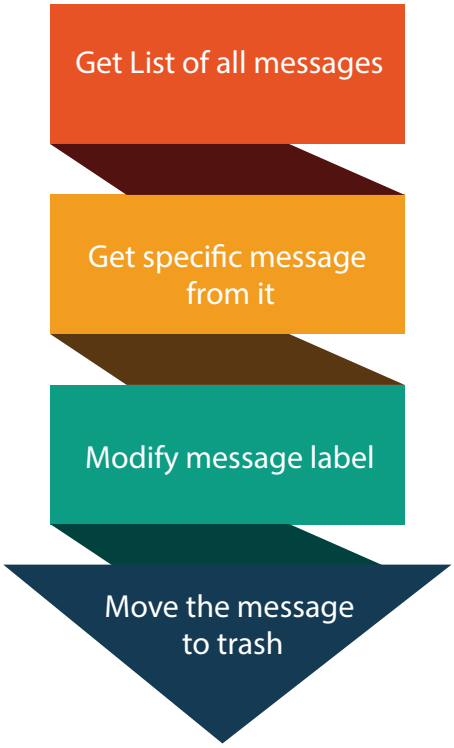


Figure 4: User flow

Following integration test scenarios can be drawn from the above user flow –

Figure 5: Integration flow 1

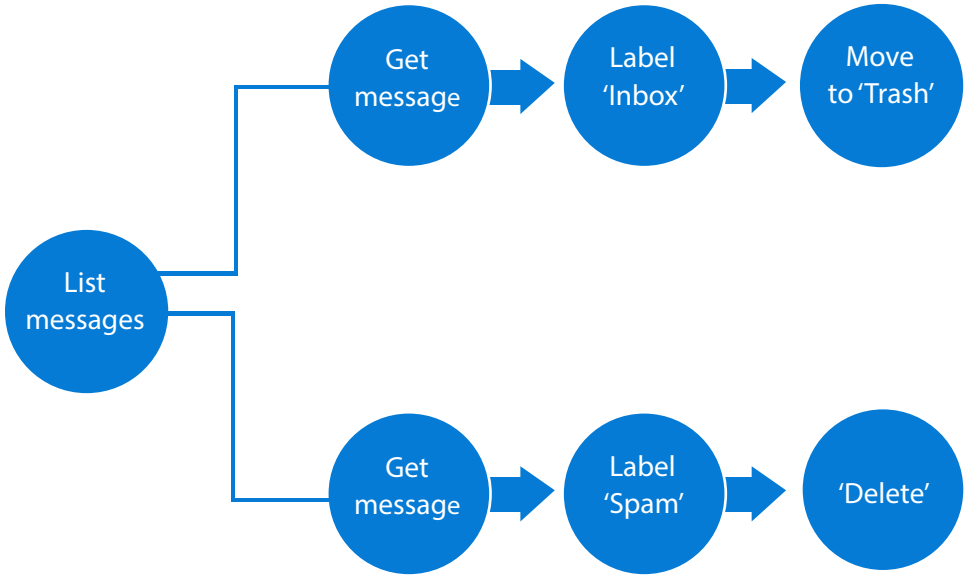


Figure 6: Integration flow 2

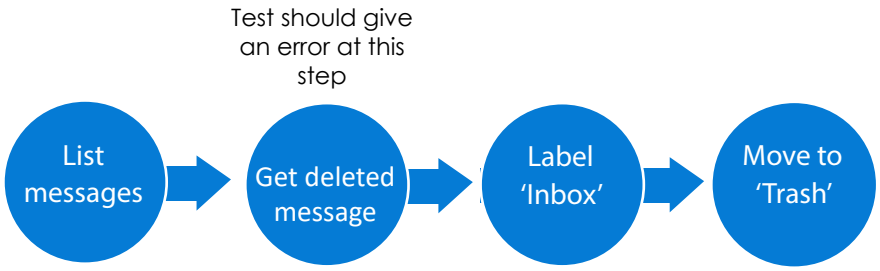


Figure 7: Integration flow 3 (Negative)

Following are the End point details needed to test the integration flows:

Method	Endpoint	Response	Action	Verifications
Get	/userId/messages	All messages ids	Capture one of the message Id	Response code Response body Response time
Get	/userId/messages/id	Message of passed id	Get the details of that particular message ID	
Post	/userId/messages/id/modify	Modified label id	Modify its label to "SPAM"	
Post	/userId/messages/id/trash	Trash label id	Move it to trash	

An integration flow is illustrated below with an example in Java –

- GET messages list, capture a message ID and pass it to next method
- Use same message ID and move it to Trash with POST method

```
//Create method instance

HttpGet get = new HttpGet("https://www.googleapis.com/gmail/v1/users/userId/
messages?key={YOUR_API_KEY}");


//Adding headers

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


//get the response

try {

     response = client.execute(get);

} catch (IOException e) {

     e.printStackTrace();

}


//Retrieving the response body

try {

    InputStreamReader in = new InputStreamReader(response.getEntity().getContent());

    BufferedReader br = new BufferedReader(in);String responseLine;

    while ((responseLine = br.readLine()) != null) {

          responseBody = responseBody + responseLine;

      }

    } catch (Exception e) {

          System.out.println(e);

    }

String JsonString = response.toString();

JSONParser parser = new JSONParser();

JSONObject object = null;

try { object = (JSONObject) parser.parse(JsonString);

} catch (ParseException e) {

     e.printStackTrace();

}
```

```
//Retrieving the first message ID from JSON response

String messageID;

JSONArray array_Nodes = (JSONArray) object.get("messages");

Iterator<JSONObject> i = array_Nodes.iterator();

JSONObject innerObject = i.next();

messageID = (String) innerObject.get("id");
```

MessageID captured in GET method above is stored and passed to next method – 'POST'.

```
//Create Post method instance

HttpPost post = new HttpPost(https://www.googleapis.com/gmail/v1/users/userId/
messages/"+<messageID>+"/trash?key={YOUR_API_KEY}");

//Adding headers

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


//get the response

try {

     response = client.execute(post);

} catch (IOException e) {

     e.printStackTrace();

}

//Retrieving the response body

try {

    InputStreamReader in = new InputStreamReader(response.getEntity().getContent());

    BufferedReader br = new BufferedReader(in);String responseLine;

    while ((responseLine = br.readLine()) != null) {

          responseBody = responseBody + responseLine;

      }

    } catch (Exception e) {

          System.out.println(e);

    }

String JsonString = response.toString();

JSONParser parser = new JSONParser();

JSONObject object = null;

try {

    object = (JSONObject) parser.parse(JsonString);

} catch (ParseException e) {

     e.printStackTrace();

}
```

```
String key = "id";

//Get the JSON object key value and verify the same message ID

String jsonBodyValue = (String) (object.get(key.toString()));

Assert.assertEquals(jsonBodyValue,messageID);


//Verify if the labelID as 'trash'

JSONArray array_Nodes = (JSONArray) object.get("labelIds");

String trashLabel = array_Nodes.get(0).toString();

Assert.assertEquals(trashLabel,"trash");
```

After executing POST method using MessageID, above snippet verifies if the same message is moved to trash by checking the JSON object.

3. Performance Testing

Web service API testing is not complete until it goes through a performance test. This is done to ascertain if the web service API responds in a stipulated time when requested from a user.

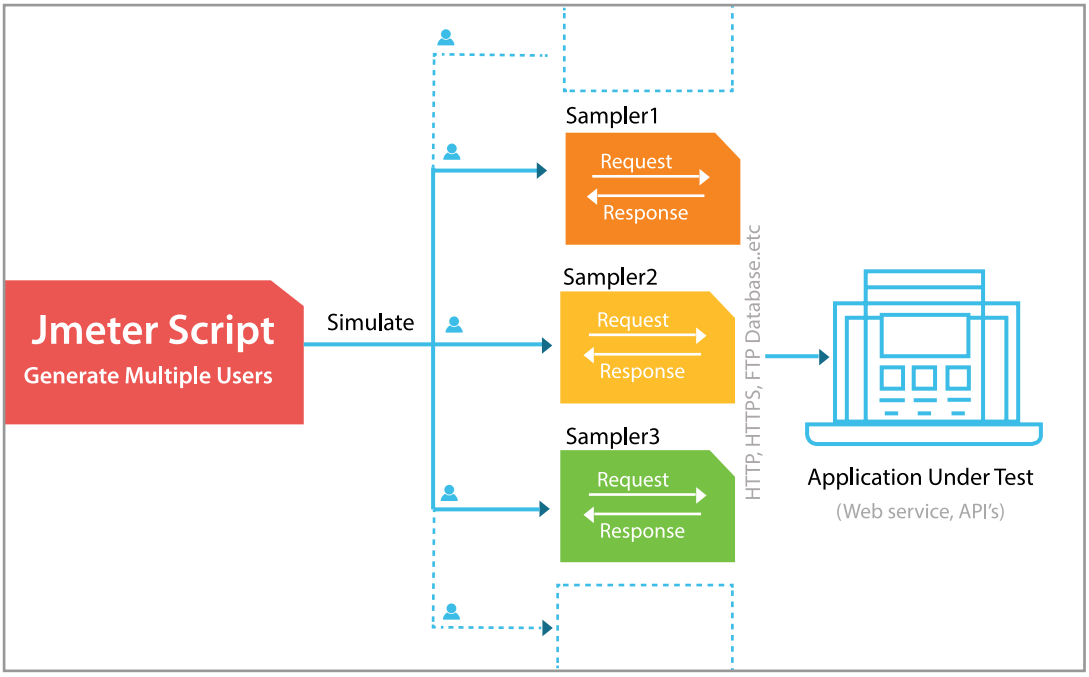
Web Services with high response time and low performance can lead to bad user experience. And hence, it is necessary to conduct Load Testing to identify any performance issues, even before the service is deployed. Web Services Load Testing helps to design and simulate usage traffic to test Web Service infrastructure for performance, reliability and scalability. This involves testing the performance and scalability of Web Services with varying simulated loads than can help determine the Web Services's behaviour in a multiple user scenario.

Few objectives and relevant test strategies to be consider for API performance tests –

Objective	Test Strategy	Methodology
API Performance under load	Load Test	Predefined number of Concurrent User Load Test simulating API calls as unit methods or as a flow.
Capacity of the infrastructure	Capacity Test	Gradually increasing concurrent user load test until performance degradation is observed or breaking point of the infrastructure
Stability of API & Instructure used for prolonged durations and detect if any memory leaks	Endurance Test	Extended duration of concurrent user load test with normal/average users count.

Key metrics to be captured during API performance tests-

- API Call Response time
- App/Web/DB Server system resources utilization (CPU, Memory, Network, etc.)
- Metrics from APM such as Transaction breakdown with App tier response time and DB query execution times, Errors and Traces, etc.



Open-source load test tools	Commercial load test tools
¬ JMeter ¬ Grinder	¬ Load Runner ¬ Neoload ¬ Load UI

4. Security Testing

The security of a Web service API is paramount to ensure that the data does not fall in the wrong hands. Since web service APIs are shared with multiple third party vendors for integration, it becomes mandatory to verify if the web service APIs are susceptible to any kind of security threat.

Following is the list of security testing types that can be performed:

Authentication

It is used to determine the identity of the user. Authentication is verified in multiple ways like - credentials, the access token (API key) or OAuth mechanism. The token is passed with each request to an API and is validated by the API before processing the request

Authorization

It is used to determine what resources the targeted user has access to

Encryption

It is used to hide information from unauthorized access. Secure Sockets Layer (SSL) is used for encrypting the data.

Signatures

Signatures are used to ensure the web service API requests and response have not been tampered during transfer of information.

The signature could be encrypted and only a few sources could be allowed to validate it or the encrypted data could be signed to further ensure that data is neither seen nor modified by unwanted parties.

Vulnerabilities

It is used to scan the system against known vulnerabilities as:

- Operating systems and network component issues such as buffer overruns, flooding with sockets, DOS attacks, etc.
- Hosting application server and related service issues such as message parsing, session hijacking or security misconfigurations
- Functional issues in the actual web service API such as injection attacks, sensitive data exposure, incomplete access control)

Major security threats can be identified by tools used for functional API testing. Some specific tools for API security testing are ZAP, IRONWASP and Burp Suite.

5. API Testing Tools

Among the plethora of tools available in the market for web service API testing, following are some of the most popular ones. These tools are classified under the following sections.

1. Manual –

These tools require testers to input manually, the required values such as Host URL, End point, Verb type, Parameters, etc., send the API request and validate the response. These tools would work in such situations, where the testing team doesn't have good programming skills or testing is required for a limited period.

2. Automation tools (Open source or Commercial)

These tools allow testers to build a customized framework or use off-the-shelf tools to test the web service APIs so that entire web service API testing process can be automated and integrated into CI systems such as Jenkins, TeamCity, etc.

Manual execution of web service API verbs	Open-sourceautoma-tion tools	Commercial automation tools
¬ Postman ¬ Http client provided ¬ Swagger ¬ Advanced Rest client ¬ JMeter ¬ Curl ¬ Fiddler	¬ HTTPClient ¬ REST Assured ¬ SoapUI	¬ HP UFT API ¬ SoapUI NG Pro (Ready API) ¬ vREST

Conclusion

With shift-left and microservices architecture coming into picture, Testing at API/Webserivces layer has become one of the primary testing strategies. This helps the tests to run much faster and ensures tests are not flaky, are inexpensive to develop and provide quick feedback for developers for any issues in the application under test.

ZenQ is a leading provider of independent software quality assurance and testing services to clients across the globe. We offer a comprehensive range of value-added testing solutions of the highest quality that our customers build quality products on.

Usage of the information and code from this Whitepaper is at the discretion of the reader. ZenQ is not responsible for any loss or damage caused by the usage of the information or code presented in this whitepaper.

For more information, please visit www.zenq.com