What is Rest Assured?

REST Assured is a Java DSL for simplifying testing of REST based services built on top of HTTP Builder. It supports POST, GET, PUT, DELETE, OPTIONS, PATCH and HEAD requests and can be used to validate and verify the response of these requests.

Rest Assured Automation Setup :

1. Download Java from below link

http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html

2. Setup Environmental variables Path

3. Download Eclipse from below link

https://eclipse.org/downloads/

4. Download Rest Assured Jars from below link

https://github.com/rest-assured/rest-assured/wiki/Downloads

5. Setup Java Project in Eclipse and configure jars in build path.

**What is end point ?**

End point: Address where API is hosted on the Server.

Ex:Base URI+Resource

**HTTP methods which are commonly used to communicate with Rest API’s are**

**GET, POST, PUT, and DELETE**

GET- The GET method is used to extract information from the given server using a given URI. While using GET request, it should only extract data and should have no other effect on the data. No Payload/Body required

**How to send input data in GET?**  
Ans: Using Query Parameters

POST- A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.

**How to send input data in POST?**  
Ans: Using Form Parameters /Body Payload

PUT- Replaces all current representations of the target resource with the uploaded content.

DELETE- Removes all current representations of the target resource given by a URI.

**Resources:  
Resources represent API/Collection which can be accessed from the Server**

Google.com/maps  
google.com/search  
google.com/images  
Here maps, search, images are resources

**Path Parameters:**  
***Path parameters*** are variable parts of a URL path. They are typically used to point to a specific resource within a collection, such as a user identified by ID

[https://www.google.com/Images/**1123343**](https://www.google.com/Images/1123343)  
[https://www.google.com/docs/**1123343**](https://www.google.com/docs/1123343)  
[https://amazon.com/orders/**112**](https://amazon.com/orders/112)

Below one is query parameter:

[https://www.google.com/search**?q=newyork&oq=newyork&aqs=chrome..69i57j0l7.2501j0j7&sourceid=chrome&ie=UTF-8**](https://www.google.com/search?q=newyork&oq=newyork&aqs=chrome..69i57j0l7.2501j0j7&sourceid=chrome&ie=UTF-8)

**Query Parameters:**  
Query Parameter is used to sort/filter the resources.

Query Parameters are identified with**?””**

https://amazon.com/orders**?sort\_by=2/20/2020**

**Headers/Cookies**:

Headers represent the meta-data associated with the API request and response. In layman terms, we were sending Additional details to API to process our request.  
Example : Authorization details

**End Point Request URL can be constructed as below**  
Base URL/resource/(Query/Path)Parameters

**Postman Tool:**

Step1: Download postman tool from google based on your operating system and processor (32/64bit)

Step2: Post man mainly used for Rest API manual testing.

Step3: For Rest API automation Testing we use **Rest Assured API(Written in java and it’s a jar file)**

Step4: Once postman got installed then we have to create a collection(Project)

Click on Create Collection 🡪provide some collection name🡪 give some description 🡪click on Create buttion

Once collection got created 🡪 Click on …(3 dots) 🡪 click on Add Request🡪 give the request name🡪Click on Save to <Collection Name> button

Double click on created Request 🡪 It will open the window where we have do api testing.

Select the Request Type based on your API(GET/POST/PUT/DELETE) the provide the endpoint url(Base URI+Resource+Path/Query Parameters)

Note: Parameters we can directly place in the address bar or we can give below parameter selection.

Now place your API request in body in 🡪Click on Body🡪check raw radio button🡪select text type as JSON🡪 Now place your request json body

Note: Now we can observe that an header got added automatically i.e. **application/json**

**Now click on Send Button🡪 then observe response and validate it that response whether it is as expected or not.**

**Note:Check status code , 200 is for success response.**

Google Maps Add API (POST):

This API Will add new place into Server

**Complete URL :**[https://rahulshettyacademy.com](https://rahulshettyacademy.com/)**/maps/api/place/add/json?key=qaclick123**

**Base URL**:[https://rahulshettyacademy.com](https://rahulshettyacademy.com/)

**Resource**:/maps/api/place/add/json

**Query Parameters**: key =qaclick123

**Http Method**:POST

**Sample Body** :

{

"location": {

"lat": -38.383494,

"lng": 33.427362

},

"accuracy": 50,

"name": "Frontline house",

"phone\_number": "(+91) 983 893 3937",

"address": "29, side layout, cohen 09",

"types": [

"shoe park",

"shop"

],

"website": "http://google.com",

"language": "French-IN"

}

**Sample Response**

{

"status": "OK",

"place\_id": "928b51f64aed18713b0d164d9be8d67f",

"scope": "APP",

"reference": "736f3c9bec384af62a184a1936d42bb0736f3c9bec384af62a184a1936d42bb0",

"id": "736f3c9bec384af62a184a1936d42bb0"

}

Google Maps Delete API (DELETE):

This API Will delete existing place from Server

Note: In some API’s , post request also will be used for add/update/delete…so POST is a super request.

**Complete URL**: https://rahulshettyacademy.com/maps/api/place/delete/json?key=qaclick123

**Base URL**: https://rahulshettyacademy.com

**Resource**: /maps/api/place/delete/json

**Query Parameters:**key

**Http request** : DELETE

**Sample Body** :

{

"place\_id":"928b51f64aed18713b0d164d9be8d67f"

}

**Sample Response**

{

"status": "OK"

}

Google Maps get Place API (GET):

**This API Will get existing place details from Server**

**Complete URL :**http://rahulshettyacademy.com/maps/api/place/get/json?place\_id=xxxx&key=qaclick123

Note: Take place\_id from 1st api response.

Ex: "place\_id": "102bc6a73585b1148582078c7f7c51a8"

**Base URL**:[https://rahulshettyacademy.com](https://rahulshettyacademy.com/)

**Resource**: /maps/api/place/get/json

**Query Parameters**: key, place\_id //( place\_id value comes from Add place response)

**Http request**: GET

Note: Key value is hardcoded and it is always qaclick123

Note: In Get request we no need to send the body, body we will send only for post request.

**Sample Response for the Provided Place\_Id**

{

"location":{

"lat" : -38.383494,

"lng" : 33.427362

},

"accuracy":50,

"name":"Frontline house",

"phone\_number":"(+91) 983 893 3937",

"address" : "29, side layout, cohen 09",

"types": ["shoe park","shop"],

"website" : "http://google.com",

"language" : "French-IN"

}

Google Maps Put Place API (PUT):

This API Will update existing place in Server with new values

**Complete URL :**http://rahulshettyacademy.com/maps/api/place/update/json?key=qaclick123

**Base URL**:[https://rahulshettyacademy.com](https://rahulshettyacademy.com/)

**Resource**: /maps/api/place/update/json

**Query Parameters**: key

**Http Method**:PUT -

Note: Key value is hardcoded and it is always qaclick123

**Sample Request:**

{

"place\_id":"8d2573bdf6ceec0e474c5f388fa917fb",

"address":"70 Summer walk, USA",

"key":"qaclick123"

}

**Sample Response for the Provided Place\_Id**

{

"location":{

"lat" : -38.383494,

"lng" : 33.427362

},

"accuracy":50,

"name":"Frontline house",

"phone\_number":"(+91) 983 893 3937",

"address" : "29, side layout, cohen 09",

"types": ["shoe park","shop"],

"website" : "http://google.com",

"language" : "French-IN"

}

**JIRA API Testing:**

In the Next lecture, I will show cookie based Authentication for Jira Server API.

**Use below official Jira link for reference to know more about this topic**

<https://developer.atlassian.com/server/jira/platform/cookie-based-authentication/>

<https://docs.atlassian.com/software/jira/docs/api/REST/7.6.1/>

**Download Jira from below link** <https://www.atlassian.com/software/jira/download>

**Authentication API**

<https://developer.atlassian.com/jiradev/jira-apis/jira-rest-apis/jira-rest-api-tutorials/jira-rest-api-example-cookie-based-authentication>

**Create Issue**

<https://docs.atlassian.com/jira/REST/cloud/#api/2/issue-createIssue>

**Please do below mandatory setting and make attachments ON before watching next video**

From Jira page, on top right, Select Settings Icon > System > Attachments (under Advanced): Set **Allow Attachments** to**ON**

**Steps to Automate in Jira Application:**

1. Login to Jira to create session using login API
2. Create an issue by using issue API
3. Add a comment to existing issue using Add comment API.
4. Add an attachment to existing issue using Add Attachment API.
5. Get issue details and verify if added comments and attachment exists using get issue API.

We are going to cover below concepts in Jira:

1. How to create session filter for authentication in Rest Assured Automation.
2. Introducing Path Parameter and Query Parameter together in a single Test.
3. Sending files as attachments using Rest Assured with MultiPart Method.
4. Parsing complex Json and limiting json response through query parameters.
5. Handling HTTPS Certification Validation through automated code.

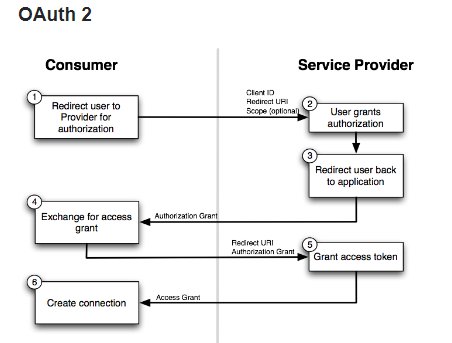
**Types of Authentications:**

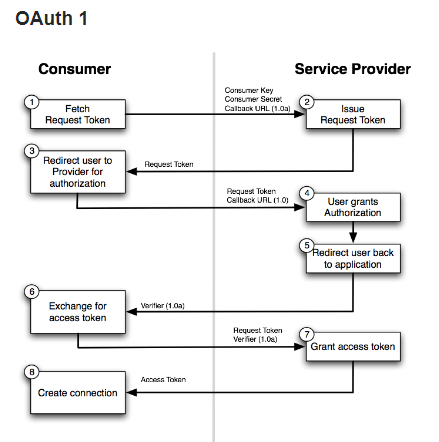
1. In Google Maps we used **key and value pair** authentication.
2. In Jira we used cookie based authentication.
3. OAuth 2.0 Authentication. Ex: BookMyshow Login, Udemy Login

It has different grant types

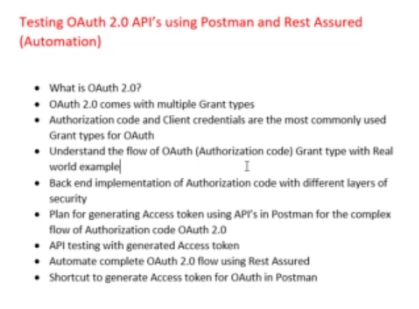
* Authorization code
* Client Credentials
* Refresh token
* Resource owner credentials

1. OAuth1.0





In Udemy we are going to cover below points.



**What is the difference between OAuth1 and OAuth2.0 ?**

The OAuth protocol specifies a process for resource owners to authorize third-party applications in accessing their server resources without sharing their credentials. This tutorial will take you through understanding OAuth protocol and introduce you to the offerings of OAuth 2.0 in a practical manner.

This article is an excerpt from a book written by Balachandar Bogunuva Mohanram, titled *[RESTful Java Web Services, Second Edition](https://www.packtpub.com/application-development/restful-java-web-services-second-edition)*.

Consider a scenario where Jane (the user of an application) wants to let an application access her private data, which is stored in a third-party service provider. Before OAuth 1.0 or other similar open source protocols, such as Google AuthSub and FlickrAuth, if Jane wanted to let a consumer service use her data stored on some third-party service provider, she would need to give her user credentials to the consumer service to access data from the third-party service via appropriate service calls.

Instead of Jane passing her login information to multiple consumer applications, OAuth 1.0 solves this problem by letting the consumer applications request authorization from the service provider on Jane’s behalf. Jane does not divulge her login information; authorization is granted by the service provider, where both her data and credentials are stored. The consumer application (or consumer service) only receives an authorization token that can be used to access data from the service provider.

Note that the user (Jane) has full control of the transaction and can invalidate the authorization token at any time during the signup process, or even after the two services have been used together.

The typical example used to explain OAuth 1.0 is that of a service provider that stores pictures on the web (let’s call the service StorageInc) and a fictional consumer service that is a picture printing service (let’s call the service PrintInc). On its own, PrintInc is a full-blown web service, but it does not offer picture storage; its business is only printing pictures. For convenience, PrintInc has created a web service that lets its users download their pictures from StorageInc for printing.

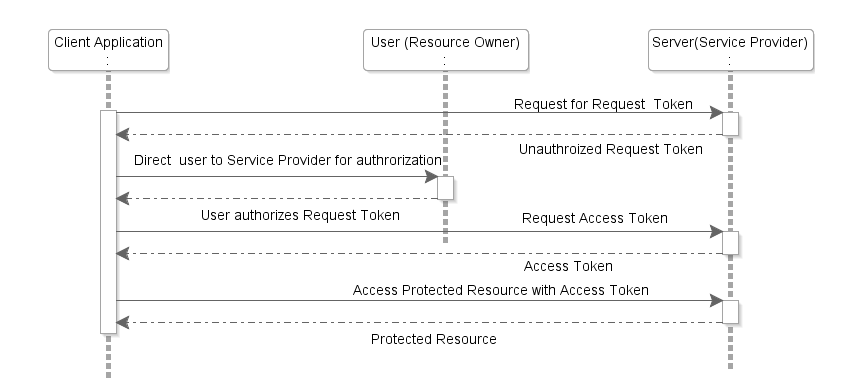
This is what happens when a user (the resource owner) decides to use PrintInc (the client application) to print his/her images stored in StorageInc (the service provider):

1. The user creates an account in PrintInc. Let’s call the user Jane, to keep things simple.
2. PrintInc asks whether Jane wants to use her pictures stored in StorageInc and presents a link to get the authorization to download her pictures (the protected resources). Jane is the resource owner here.
3. Jane decides to let PrintInc connect to StorageInc on her behalf and clicks on the authorization link.
4. Both PrintInc and StorageInc have implemented the OAuth protocol, so StorageInc asks Jane whether she wants to let PrintInc use her pictures. If she says yes, then StorageInc asks Jane to provide her username and password. Note, however, that her credentials are being used at StorageInc’s site and PrintInc has no knowledge of her credentials.
5. Once Jane provides her credentials, StorageInc passes PrintInc an authorization token, which is stored as a part of Jane’s account on PrintInc.
6. Now, we are back at PrintInc’s web application, and Jane can now print any of her pictures stored in StorageInc’s web service.
7. Finally, every time Jane wants to print more pictures, all she needs to do is come back to PrintInc’s website and download her pictures from StorageInc without providing the username and password again, as she has already authorized these two web services to exchange data on her behalf.

The preceding example clearly portrays the authorization flow in OAuth 1.0 protocol. Before getting deeper into OAuth 1.0, here is a brief overview of the common terminologies and roles that we saw in this example:

* **Client (consumer)**: This refers to an application (service) that tries to access a protected resource on behalf of the resource owner and with the resource owner’s consent. A client can be a business service, mobile, web, or desktop application. In the previous example, PrintInc is the client application.
* **Server (service provider)**: This refers to an HTTP server that understands the OAuth protocol. It accepts and responds to the requests authenticated with the OAuth protocol from various client applications (consumers). If you relate this with the previous example, StorageInc is the service provider.
* **Protected resource**: Protected resources are resources hosted on servers (the service providers) that are access-restricted. The server validates all incoming requests and grants access to the resource, as appropriate.
* **Resource owner**: This refers to an entity capable of granting access to a protected resource. Mostly, it refers to an end user who owns the protected resource. In the previous example, Jane is the resource owner.
* **Consumer key and secret (client credentials)**: These two strings are used to identify and authenticate the client application (the consumer) making the request.
* **Request token (temporary credentials)**: This is a temporary credential provided by the server when the resource owner authorizes the client application to use the resource. As the next step, the client will send this request token to the server to get authorized. On successful authorization, the server returns an access token. The access token is explained next.
* **Access token (token credentials)**: The server returns an access token to the client when the client submits the temporary credentials obtained from the server during the resource grant approval by the user. The access token is a string that identifies a client that requests for protected resources. Once the access token is obtained, the client passes it along with each resource request to the server. The server can then verify the identity of the client by checking this access token.

The following sequence diagram shows the interactions between the various parties involved in the OAuth 1.0 protocol:



You can get more information about the OAuth 1.0 protocol [here.](https://tools.ietf.org/html/rfc5849)

## What is OAuth 2.0?

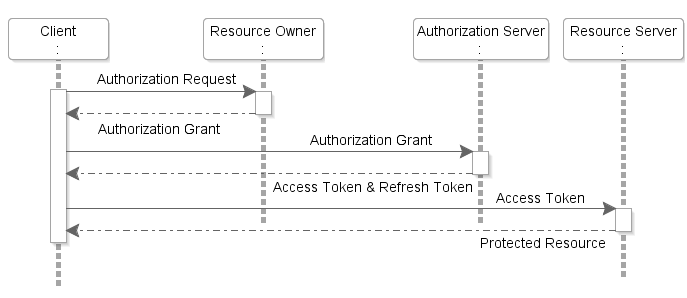
OAuth 2.0 is the latest release of the OAuth protocol, mainly focused on simplifying the client-side development. Note that OAuth 2.0 is a completely new protocol, and this release is not backwards-compatible with OAuth 1.0. It offers specific authorization flows for web applications, desktop applications, mobile phones, and living room devices. The following are some of the major improvements in OAuth 2.0, as compared to the previous release:

* **The complexity involved in signing each request**: OAuth 1.0 mandates that the client must generate a signature on every API call to the server resource using the token secret. On the receiving end, the server must regenerate the same signature, and the client will be given access only if both the signatures match. OAuth 2.0 requires neither the client nor the server to generate any signature for securing the messages. Security is enforced via the use of TLS/SSL (HTTPS) for all communication.
* **Addressing non-browser client applications**: Many features of OAuth 1.0 are designed by considering the way a web client application interacts with the inbound and outbound messages. This has proven to be inefficient while using it with non-browser clients such as on-device mobile applications. OAuth 2.0 addresses this issue by accommodating more authorization flows suitable for specific client needs that do not use any web UI, such as on-device (native) mobile applications or API services. This makes the protocol very flexible.
* **The separation of roles**: OAuth 2.0 clearly defines the roles for all parties involved in the communication, such as the client, resource owner, resource server, and authorization server. The specification is clear on which parts of the protocol are expected to be implemented by the resource owner, authorization server, and resource server.
* **The short-lived access token**: Unlike in the previous version, the access token in OAuth 2.0 can contain an expiration time, which improves the security and reduces the chances of illegal access.
* **The refresh token**: OAuth 2.0 offers a refresh token that can be used for getting a new access token on the expiry of the current one, without going through the entire authorization process again.

Before we get into the details of OAuth 2.0, let’s take a quick look at how OAuth 2.0 defines roles for each party involved in the authorization process. Though you might have seen similar roles while discussing OAuth 1.0 in last section, it does not clearly define which part of the protocol is expected to be implemented by each one:

* **The resource owner**: This refers to an entity capable of granting access to a protected resource. In a real-life scenario, this can be an end user who owns the resource.
* **The resource server**: This hosts the protected resources. The resource server validates and authorizes the incoming requests for the protected resource by contacting the authorization server.
* **The client (consumer)**: This refers to an application that tries to access protected resources on behalf of the resource owner. It can be a business service, mobile, web, or desktop application.
* **Authorization server**: This, as the name suggests, is responsible for authorizing the client that needs access to a resource. After successful authentication, the access token is issued to the client by the authorization server. In a real-life scenario, the authorization server may be either the same as the resource server or a separate entity altogether. The OAuth 2.0 specification does not really enforce anything on this part.

It would be interesting to learn how these entities talk with each other to complete the authorization flow. The following is a quick summary of the authorization flow in a typical OAuth 2.0 implementation:



Let’s understand the diagram in more detail:

1. The client application requests authorization to access the protected resources from the resource owner (user). The client can either directly make the authorization request to the resource owner or via the authorization server by redirecting the resource owner to the authorization server endpoint.
2. The resource owner authenticates and authorizes the resource access request from the client application and returns the authorization grant to the client. The authorization grant type returned by the resource owner depends on the type of client application that tries to access the OAuth protected resource. Note that the OAuth 2.0 protocol defines four types of grants in order to authorize access to protected resources.
3. The client application requests an access token from the authorization server by passing the authorization grant along with other details for authentication, such as the client ID, client secret, and grant type.
4. On successful authentication, the authorization server issues an access token (and, optionally, a refresh token) to the client application.
5. The client application requests the protected resource (RESTful web API) from the resource server by presenting the access token for authentication.
6. On successful authentication of the client request, the resource server returns the requested resource.

The sequence of interaction that we just discussed is of a very high level. Depending upon the grant type used by the client, the details of the interaction may change. The following section will help you understand the basics of grant types.

### Understanding grant types in OAuth 2.0

Grant types in the OAuth 2.0 protocol are, in essence, different ways to authorize access to protected resources using different security credentials (for each type). The OAuth 2.0 protocol defines four types of grants, as listed here; each can be used in different scenarios, as appropriate:

* **Authorization code**: This is obtained from the authentication server instead of directly requesting it from the resource owner. In this case, the client directs the resource owner to the authorization server, which returns the authorization code to the client. This is very similar to OAuth 1.0, except that the cryptographic signing of messages is not required in OAuth 2.0.
* **Implicit**: This grant is a simplified version of the authorization code grant type flow. In the implicit grant flow, the client is issued an access token directly as the result of the resource owner’s authorization. This is less secure, as the client is not authenticated. This is commonly used for client-side devices, such as mobile, where the client credentials cannot be stored securely.
* **Resource owner password credentials**: The resource owner’s credentials, such as username and password, are used by the client for directly obtaining the access token during the authorization flow. The access code is used thereafter for accessing resources. This grant type is only used with trusted client applications. This is suitable for legacy applications that use the HTTP basic authentication to incrementally transition to OAuth 2.0.
* **Client credentials**: These are used directly for getting access tokens. This grant type is used when the client is also the resource owner. This is commonly used for embedded services and backend applications, where the client has an account (direct access rights).