Rest Assured

Rest Assured is one of the most used library for REST API Automation Testing.

Rest-Assured is a Java-based library that is used to test RESTful Web Services.

Rest-Assured library also provides the ability to validate the HTTP Responses received from the server. For e.g. we can verify the Status code, Status message, Headers and even the Body of the response.

# What is API Testing?

An API ([Application Programming Interface](http://www.guru99.com/api-testing.html)) is a collection of software functions and procedures which can be executed by other software applications.

So API testing is:

* Testing without GUI
* Programmatically simulate data or control follow scenarios.
* Focus on functionality, not on behaviour or customer experience.

# Why API Testing?

1. **API testing is the trend**
2. **Time efficient. – Faster Execution**
3. **Language independent -** In API testing, data is exchanged via [XML](http://en.wikipedia.org/wiki/XML) or [JSON](http://json.org/) so that any language can be used to test the response. For example, if you have service whose response is in JSON format, you can easily parse data with Java, C# or any language.

# Why to Automate API?

1. Does the service/application respond quickly enough for the intended users?
2. Will the application's server respond with the correct values?
3. How will the service/application handle exceptions and illegal values?
4. Is the service/application stable under expected and unexpected user loads?

# What is Request and Response?

Client sends the request to the server to fetch some information or data. Additionally, the client's request is an HTTP Request, which communicates between the client and the server, or you may say, two different computer systems. Moreover, it is a simple text file formatted in either *XML* or *JSON*, which sends the client's binary data to the server.

Sample JSON:

<https://reqres.in/>

## Different Request Methods

* **GET** - Get method fetches the information from the server. Moreover, it is the most commonly used method which does not have a request body.
* **POST** - The Post method works to send data to the server.
* **PUT** - The Put method is similar to the Post method since it updates the data. The only difference is that we use it when we have to replace an existing entity completely.
* **DELETE** - the Delete method deletes the server's representations of resources through the specific URL. Additionally, just like the Get method, they do not have a request body.
* **PATCH** - This method is again similar to Post and Put methods, but we use it when we partially update some data. Moreover, unlike the Post and Put methods, you may only send the Patch method to the entity that needs updating in the request body.

## HTTP Respose

Every HTTP Response received as a result of an HTTP request sent by the client to the server has a status code. This status code value tells us if HTTP Response was successful or not.

An HTTP Response contains:

1. A status.
2. Collection of Headers.
3. A Body.

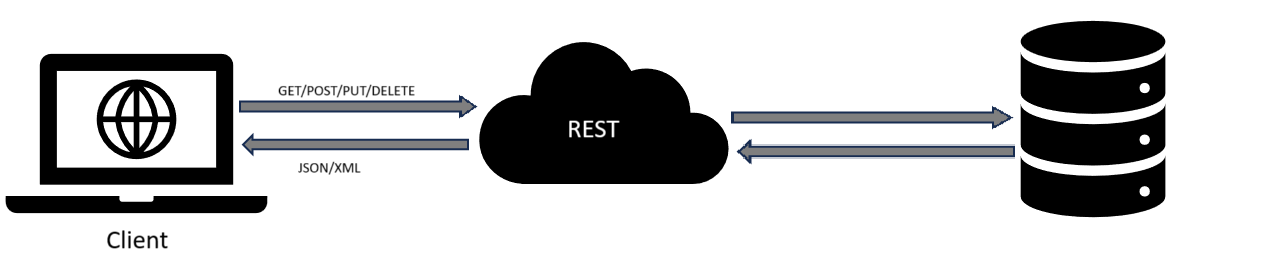
When the client requests a piece of particular information from the server, the server sends a response with a status code back to the client. The status code that the server returns tells us whether the request was successful or not. If the request was successful, the server sends the status code in the range of 200-299.

Status code definitions

<https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

# What is REST?

REST, or **RE**presentational **S**tate **T**ransfer, is an architectural style and provides standards between the systems on the web.



# RestAPI Testing using RestAssured

## GET

Steps to verify GET request

1. Use the RestAssured class to generate a RequestSpecification for the URL.
2. Specify the HTTP Method type (GET method).
3. Send the Request to the server.
4. Get the Response back from the server.
5. Verify the Response code
6. Print the returned Response’s body.

### Simple code to perform GET request

@Test

**public** **void** getUserDetails() {

RestAssured.*baseURI* = "https://reqres.in/api/";

Response res = RestAssured.*given*().request(Method.***GET***,"users/2");

System.***out***.println(res.getStatusCode());

System.***out***.println(res.body().asPrettyString());

System.***out***.println(res.statusLine());

}

### Validating Response Header

The response received from the server consists of zero or more headers along with response status and response body. Each header is a key-value pair. The header part of the response is used by the server to send extra information which is also referred to as "***Metadata***" of the response.

For example, headers contain a "***Content-Type***" attribute that tells us how to interpret the data of the response body. So if the response body contains JSON data, then the corresponding content-type attribute in the header will be "***application/json***". *Similarly, if the data in the body is XML the****Content-Type****header will be "****application/xml****"*.

Below are the methods available to validate the headers

* ***headers()*** : returns ***Headers***
* ***getHeader()***: returns a ***Header***
* ***getHeaders()***: returns ***Headers***

**public** **void** getUserDetails() {

RestAssured.*baseURI* = "https://reqres.in/api/";

Response res = RestAssured.*given*().request(Method.***GET***,"users/2");

Headers getAllHeaders = res.headers();

**for**(Header h: getAllHeaders) {

System.***out***.println(h.getName()+"----"+h.getValue());

}

Assert.*assertEquals*("application/json; charset=utf-8", res.header("Content-Type"));

}

## Validating Response body

Below two methods will get us the response body

* ***Response.body() : returns ResponseBody***
* ***Response.getBody() : returns ResponseBody***

### Validating using string contains

String getResBody = res.body().asString();

Assert.*assertTrue*(getResBody.contains("janet.weaver@reqres.in"));

### By using Json Path

The above approach has a problem problem, what if the string " janet.weaver@reqres.in " is present in a wrong node or may be multiple instances of the same string are present.

Response interface gives you a mechanism to extract nodes based on a given JsonPath. There is a method called Response.JsonPath(), which returns a io.restassured.path.json.JsonPath Object. This object can be used to further query specific parts of the Response Json.

JsonPath jsonpathExtract = **new** JsonPath(res.asString());

String getO = jsonpathExtract.getString("data.email");

System.***out***.println(jsonpathExtract.getInt("data.id"));

### For multiple data

JsonPath jsonpathExtract = **new** JsonPath(res.asString());

String getO = jsonpathExtract.getString("data[1].email");

# Performing POST request

POST is a create operation in CRUD. when we are submitting any registration form on a particular webpage like Gmail. We provide the required data and click submit. So through this action of submitting data, we are actually POSTING or sending the data to the server.

## Steps to perform a POST request

1. Create a Request pointing to the service Endpoint.
2. Create a JSON Request which contains all the fields.
3. Add JSON body in the request and send the request.
4. Validate the Request.

To perform POST request, add the below dependency for JSON Simple

<dependency>

<groupId>com.googlecode.json-simple</groupId>

<artifactId>json-simple</artifactId>

<version>1.1.1</version>

</dependency>

RestAssured.*baseURI* = "https://reqres.in/api/users";

JSONObject requestParams = **new** JSONObject();

requestParams.put("name", "johnsmith");

requestParams.put("job", "Accountant");

Response res = RestAssured.*given*().body(requestParams.toJSONString()).when().post("");

System.***out***.println(res.getStatusCode());

System.***out***.println(res.getBody().asPrettyString());

JsonPath j = **new** JsonPath(res.asString());

System.***out***.println(j.getString("id"));

# Performing PUT Request

The PUT method creates a new resource or updates (substitutes) a representation of the target resource with the request payload. This means a Put request updates a resource at a specified URI. It is also used to create a new resource at the given URI or replace the entire product entity.

* A PUT method puts or places a file or resource precisely at a specific URI.
  + In case a file or a resource already exists at that URI, the PUT method replaces that file or resource.
  + If there is no file or resource, PUT creates a new one.

|  |  |
| --- | --- |
| PUT | POST |
| This method is idempotent. This means it will produce the same results if executed more than once. | This method is not idempotent. It produces different results every time it is executed. |
| When we need to modify a single resource that is already part of resource collection, we call the PUT method. | POST method is called when a child resource is to be added under resources collection. |
| PUT method syntax : PUT /questions/{question-id} | POST method syntax: POST /questions |
| PUT works as specific. | POST works as abstract. |
| Put method makes use of the *"UPDATE"* query. | POST method makes use of the *"CREATE"* query. |
| In the PUT method, the client decides which URI resource should have. | In the POST method, the server decides which URI resource should have. |
| \**PUT /vi/book/orders/1234* indicates updation of a resource identified by *"1234"*. | *POST /vi/book/orders*\* indicate that we are creating a new resource and return an identifier to describe the resource. |

header("Authorization", "Bearer " + token)

RestAssured.*baseURI* = "https://reqres.in/api";

JSONObject requestParams = **new** JSONObject();

requestParams.put("name", "johnsmith1");

requestParams.put("job", "Accountant");

Response res = RestAssured.*given*().header("Content-Type","application/json").body(requestParams.toJSONString()).when().put("/users/3");

System.***out***.println(res.getStatusCode());

System.***out***.println(res.getBody().asPrettyString());

JsonPath j = **new** JsonPath(res.asString());

System.***out***.println(j.getString("name"));

# Patch Request

RestAssured.*baseURI* = "https://reqres.in/api";

JSONObject requestParams = **new** JSONObject();

requestParams.put("name", "johnsmith1");

//requestParams.put("job", "Accountant");

Response res = RestAssured.*given*().header("Content-Type","application/json").body(requestParams.toJSONString()).when().patch("/users/3");

System.***out***.println(res.getStatusCode());

System.***out***.println(res.getBody().asPrettyString());

JsonPath j = **new** JsonPath(res.asString());

# Delete Request

*An HTTP delete request is performed using the HTTP delete method (written as delete) which deletes a resource from the server identified by the URI we are sending through it.*

*Respose code for delete*

* ***202****(Accepted): The server accepts the request but does not enact.*
* ***204****(No Content)- A status code of 204 on the HTTP delete request method denotes successful enactment of the delete request without any content in the response.*
* ***200****(OK)- The action was successful and the response message includes representation with the status.*
* ***404****(Not Found) - When the server can't find the resource. The reason could either does not exist or previously deleted.*

Response res = RestAssured.*given*().header("Content-Type","application/json").when().delete("/users/2");

# SOAP, WSDL and UDDI

**SOAP (Simple Object Access Protocol)**

SOAP is an XML-based messaging protocol that allows applications to exchange structured information over the Internet. It provides a standard way for applications to communicate with each other, regardless of the programming language or platform they use.

**WSDL (Web Services Description Language)**

WSDL is an XML-based language used to describe web services and their operations, inputs, and outputs. It provides a way for developers to understand how to interact with a web service without having to access its source code.

**UDDI (Universal Description, Discovery, and Integration)**

UDDI is a registry standard for web services, allowing businesses to publish and discover services over the internet. It provides a centralized directory where developers can search for and obtain information about available web services.

# Serialization and Deserialization in Java

Serialization – Converting a Java object into a filesystem object is called serialization. We may sometimes want to send an object to the network. That is when we may want to save the object in a file and send it through the network.

We need below two classes from Java to do serialization:

* 1. **FileOutputSteam**
  2. **ObjectOutputSteam**

Deserialization - Converting a filesystem to a Java object is called deserialization.

We need below two classes from Java to do serialization:

* 1. **FileInputSteam**
  2. **ObjectInputSteam**

**class** Test1 **implements** Serializable{

**int** i=10, j=20;

}

**public** **class** Ser {

**public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {

Test1 t = **new** Test1();

FileOutputStream foo = **new** FileOutputStream("test.txt");

ObjectOutputStream oo = **new** ObjectOutputStream(foo);

oo.writeObject(t);

FileInputStream fii = **new** FileInputStream("test.txt");

ObjectInputStream oi = **new** ObjectInputStream(fii);

Test1 readtObj = (Test1) oi.readObject();

System.***out***.println(readtObj.i);

}

}

Let us take the below API

<https://github.com/vdespa/introduction-to-postman-course/blob/main/simple-books-api.md>

Create a Books Class

**public** **class** Books {

**public** **int** bookId;

**public** String customerName;

**public** **int** getbookID() {

**return** bookId;

}

**public** **void** setbook(**int** bookId) {

**this**.bookId = bookId;

}

**public** String getCustomerName() {

**return** customerName;

}

**public** **void** setCustomerName(String customerName) {

**this**.customerName = customerName;

}

}

**Add Jackson Dependency**

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.17.2</version>

</dependency>

---------------------Rest Assure Test-------------------

**public** **void** getUserDetails() {

RestAssured.*baseURI* = "https://simple-books-api.glitch.me";

String token = "3dd6847ac82835b8659f3018b589cdcd2c2f4b6418ad6840b67dd2485a40d75c";

Books bk = **new** Books();

bk.setbook(1);

bk.setCustomerName("John");

Response res =RestAssured.*given*().header("Content-Type","application/json").header("Authorization", "Bearer " + token).body(bk).when().post("/orders");

System.***out***.println(res.getStatusCode());

System.***out***.println(res.getBody().toString());

JsonPath j = **new** JsonPath(res.asString());

String OrderNum =j.getString("orderId");

System.***out***.println(OrderNum);

Response getRes = RestAssured.*given*().header("Content-Type","application/json").header("Authorization", "Bearer " + token).when().get("/orders/"+OrderNum);

JsonPath j1 = **new** JsonPath(getRes.asString());

System.***out***.println(j1.getString("bookId"));

System.***out***.println(j1.getString("customerName"));

---------------------------------------For Deserialization-----------------------------------------------

Add below line for Books class

@JsonIgnoreProperties(ignoreUnknown = **true**)

Books bb = RestAssured.*given*().header("Content-Type","application/json").header("Authorization", "Bearer " + token).when().get("/orders/"+OrderNum).as(Books.**class**);

System.***out***.println(bb.getbookID());