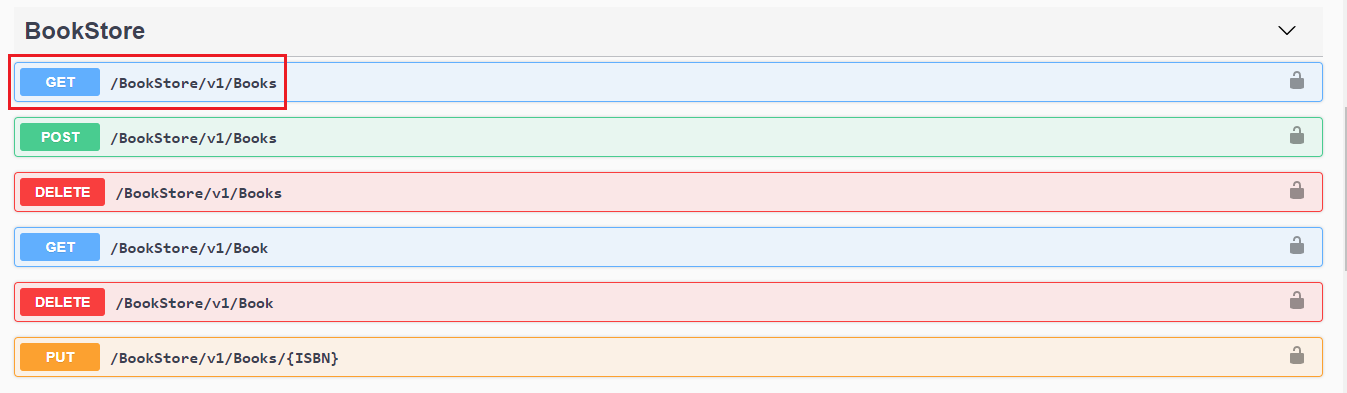
API Automation

REST (***RE****presentational****S****tate****T****ransfer*) was first presented in the year 2000 by [***Roy Fielding***](https://en.wikipedia.org/wiki/Roy_Fielding) as an architectural style for ***distributed hypermedia systems.*** REST-compliant or RESTful systems, are ***"stateless"*** *(discussed later in this article)* and separate a client and a server. A web application developed using REST *(RESTful web application)* exposes the data or information about its resources which can be anything that the developer wants. This allows the client using this application to take action on the resources. For example, using information exposed to users, clients can create a new user.

As another example, suppose we have a pet store and the information related to all pets in the store is stored on the server. If we want information related to a pet *(let's say pet with id=0)* then we will access the appropriate URL of the pet store in the browser *(Here the browser will be the client).* An example URL can be [***"https://demoqa.com/BookStore/v1/Books".***](https://demoqa.com/swagger/#/BookStore/BookStoreV1BooksGet)

The above interface looks as shown below:



All the methods *(coloured buttons)* shown in the above screenshot correspond to REST APIs that are executed with the click of a button. Please note that this is a demo website and therefore the APIs are so explicit on the screen. In general, all this is done in the back-end and the client is not shown any of this information. Coming back, when we click the ***"GET"*** button *(/Books),* we will get the response in JSON format that will show us the details of the particular book. This response is shown below:

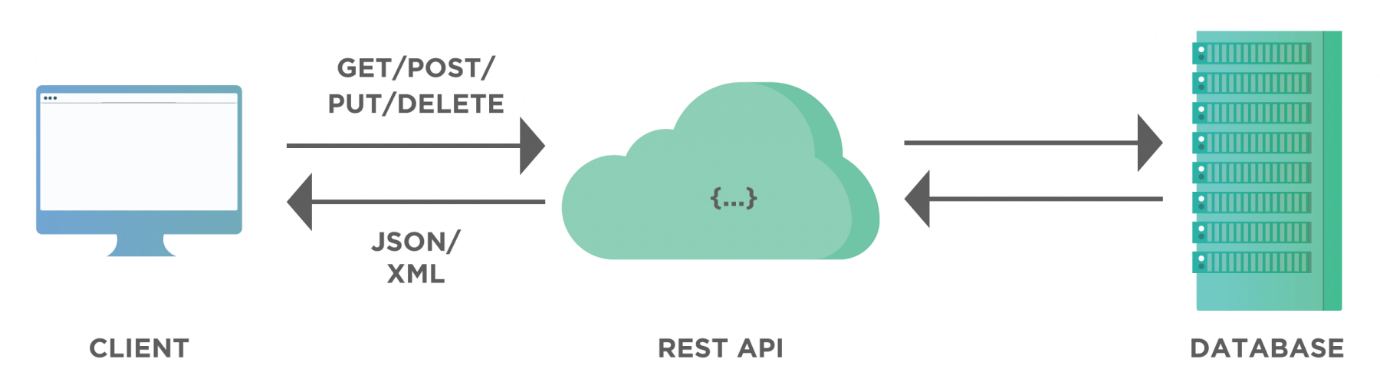


*Note: When we click on the****"GET"****button, in the browser we see the link changes to*[***"https://demoqa.com/BookStore/v1/Books".***](https://demoqa.com/swagger/#/BookStore/BookStoreV1BooksGet)

We will discuss all the methods shown above *(in form of buttons)* in our subsequent articles.

***Representation of REST flow***

Now let us depict the actual REST data transfer in the above example in the diagram below.



As the above diagram shows, the API works as a medium of communication between a database *(that is a part of a server)* and a client. When a client sends data through APIs, it goes to the database, do the appropriate operation *(such as add, delete, modify etc.)* and return the response data that contains response code, header files, cookie info etc.

We can summarise the REST characteristics and the working in the above diagram as follows:

* *A client access data from the server passed by REST. This can either be through authorized access or without following any strict guideline.*
* *As we can see, the application developed using REST is an interactive application and mostly it uses web services. In this case, the web service follows RESTful guidelines and fulfils the constraints of REST that are discussed later in this article.*
* *A web service using REST provides web resources in textual format and allows them to be read and edited using a predefined set of operations.*

REST is a way to access a web service and is often viewed as an alternative to [***SOAP***](https://www.toolsqa.com/soapui/what-is-soapui/) *(Simple Object Access Protocol).*

In a RESTFul application, we have entities namely client and resource which are used commonly. Let us discuss them briefly next.

**What are Clients and Resources?**

A client can be a software or a person or a system using the APIs to access data from another application server. For example, a developer might access Facebook APIs to embed a live post in their own website. The developer program will call the Facebook API through a browser. So in this case, the browser acts as a client that is calling the Facebook APIs.

If we visualize this system using the REST diagram above, the client or browser will connect to Facebook-Server over REST API and then get the information required to render it on the screen.

As another example, suppose I have an application ***"myHealthApp".*** I want data on the Covid-19 pandemic from a city, Pune, for example. To achieve this, I will develop APIs or methods, such as, ***"Corona API"*** using which I can request the data from Pune Municipal Council (*PMC*). This is done using myHealthApp. This means using myHealthApp which acts as a client, I will make a request for data using Corona API *(say getCoronaData method to be more specific)* through the browser *( for example,****https://myHealthApp/getCoronaData****).* This getCoronaData method will in turn connect to PMC servers and fetch the required data as a JSON response *(or any other format).* On the client side, I can take that data and perform various operations on it. This is the use of APIs.

***What Is A Resource in REST?***

Any smaller unit that can be transformed and addressed through a URL and HTTP method is considered a resource. This resource makes changes to the database. For one application, you may have a lot of resources with all of them assigned a particular task. For example, an online book store may have a resource as a table of the database. A resource in a REST architecture is anything that a client has access to and can modify or update.

In a way, you can say that a resource is any object for which we need information from API. So in the above example about Facebook, a resource can be a post, page, or user account. In the example of Corona API, resources are all the details on Corona like Corona data, pages on treatment, vaccination, etc.

**Guiding principles or constraints of REST**

For an API to be RESTful, it has to fulfil or adhere to the following guiding principles or constraints defined by REST.

***Uniform interface***

The uniform interface principle has the following parts that an API has to follow:

1. *The request to the server needs to have a resource identifier.*
2. *The server returns the response and includes information such that the client can modify or edit the resource.*
3. *The request sent through the API contains the information that the server needs to execute the request. Each response returned by the server also contains all the information so that client understands the response.*
4. *Hypermedia as the engine of application state. The application means the web application that is running on the server. Hypermedia is the hyperlinks or simple links included in the response. So the server basically informs the client about the ways to change the application web.*

***Client-server separation***

The interaction between the client and the server is independent and is only in the form of requests. The client initiates a request and the server sends the responses. The response is a reaction to the request. So all server does is wait to receive requests from the client. It never sends out information about the resources to the client on its own. For more details on client-server refer to our tutorial, [***Client-server Architecture.***](https://toolsqa.com/client-server/client-server-architecture-and-model)

***Stateless***

The word ***"stateless"*** means the server does not keep track of the user who uses the API. So when a new request comes in, the server will not know if the same user has sent a GET request in the past for the same resource. It doesn't remember the resources requested by the user earlier. For example, HTTP is a stateless protocol. HTTP server does not keep track of any state of information passed to and fro. Hence at any given time, a client can send a valid command, and the server will not relate or keep track of any previous similar commands.

Hence each request regardless of the other requests made by the user will contain all the data needed by the server to execute the request and send a response.

***Layered system***

A layered system provides a hierarchical structure between a client and a server. There can be a lot of intermediaries between the client and the server working along with REST API without the client's notice. Our clients think there is a direct connection to the server. We then take advantage of it to improve our architecture and bring down our distributed system complexity. These intermediate elements provide a security layer, load-balancing layer, and other functionality to the system. The only guideline is that the presence of these intermediate layers should not affect the request or response.

*Note: The abstraction of layers does not let one layer be aware of the presence of another layer.*

***Cacheable***

The server data received in the response contains information regarding whether the data is cacheable or not. If the data is cacheable, it will contain some kind of version number that makes caching possible. The client will know which version of the data it has got from the previous response. This way client can avoid requesting repeated data. Cacheable data *(and therefore version number)* also helps the client to know the expiration of version data and the requirement of a new request to fetch the latest data.

***Code-on-demand***

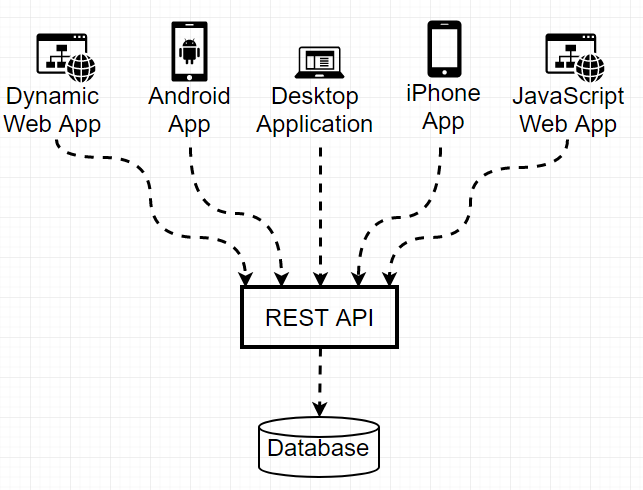
This particular constraint ***"Code on demand"*** is optional and without fulfilling it we can have a RESTful API.

The client can send a request to the server asking for the code and then the server will respond with some code in the form of a script or some other entity. For example, servers can extend the client functionality by downloading and executing pre-compiled code like an applet or a client-side script like JavaScript. So when we click on any video on Facebook, Facebook will run a precompiled or any third-party software to run that video.

Once an API fulfils the above constraints we discussed, we can say it is a RESTful API.

**What is REST API?**

An API *(Application Programming Interface)* conforming to the REST architectural style or REST standards is known as a ***"REST API".*** A Rest API facilitates interaction between the client and RESTful web services\*( server)\* to get the required information. They can be used for a variety of purposes by interacting with the system. These may include specific actions such as retrieving the location of a particular city or data updation such as registering a new user. API developers use REST standards in a variety of ways to develop REST APIs. The following diagram shows a general REST API functionality.



As shown in the above diagram, REST API sits in the middle layer to the database and the presentation layer *i.e. the interactive systems.* The other applications *(shown as the top layer)* will call the REST API that has centralized core logic in one place. The applications call REST APIs to access the desired data. For example, if we try to hard code everything, we need to code for each action on the website. Retrieving book by book serial number is an action that may need to run through the middle layer and then the database. It can take a lot of time depending on the size of the database and website. With RESTful APIs, the process becomes much faster as they are lightweight.  So instead of writing separate code and logic for each application, we write REST APIs accessible by any application.

***What happens when a client makes a request through the RESTful API?***

When a client makes a specific request, RESTful API transfers a state representation of the resource to the requester or endpoint. The format of this representation is one of several formats like HTTP: JSON *(Javascript Object Notation),* HTML, XLT, Python, PHP, or plain text.  The most popular format generally used nowadays is JSON since it is easy to read and is very lightweight.

Headers and parameters also play an important role in the HTTP methods of a RESTful API [***HTTP request.***](https://www.toolsqa.com/client-server/http-request/) They contain important identifier information regarding the request's uniform resource identifier *(URI),* metadata, authorization, caching, cookies, etc. These request and response headers have their own HTTP connection and status code information.

Next, let us move on to why we use REST APIs.

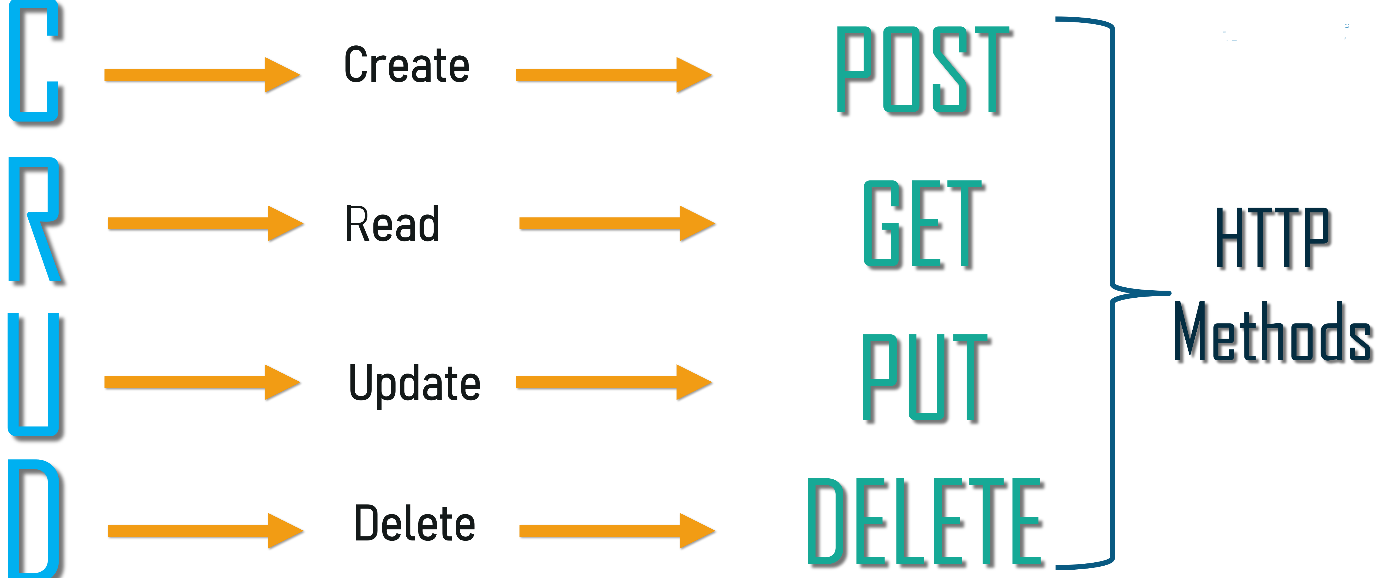
**Why use REST API?**

We mainly use REST API for the following reasons:

1. *As we have discussed earlier, the REST API creates an object and transmits the object values in response to the client's request. The REST API breaks down this transaction into smaller modules wherein each module addresses a specific part of the transaction. This breaking down of transactions into smaller modules requires lots of effort but also provides more flexibility for the user.*
2. *The REST API has strict criteria to conform to. We have already seen the guiding principles (constraints) for an application to be RESTAssured in our earlier tutorials of*[***What is REST***](https://toolsqa.com/rest-assured/what-is-rest/)*and*[***Rest Architectural Element.***](https://toolsqa.com/rest-assured/rest-architectural-elements/)*This strict adherence results in efficient REST APIs after their development. Also, we can implement this set of guidelines as needed.*
3. *The REST APIs are considered easier than other protocols like*[***SOAP***](https://en.wikipedia.org/wiki/SOAP)*(Simple Object Access Protocol) that have more specific requirements like built-in security, transaction compliance, XML messaging, etc. All these requirements make these protocols slower and heavier.*
4. *The REST applications (and APIs) are more lightweight, have increased scalability, and are more suited for IoT (Internet Of Things) and mobile app development.*

**Methods of RESTful API**

We perform CRUD Operations when we are working with web technologies and applications. As shown in the below figure, CRUD stands for ***C***reate, ***R***ead, ***U***pdate, ***D***elete. This means using CRUD operations, we can create, read, update and delete a resource. Generally, we make use of HTTP methods to perform CRUD operations. In the case of REST API methods, the REST provides these four methods in the form of APIs. Refer to the figure below:



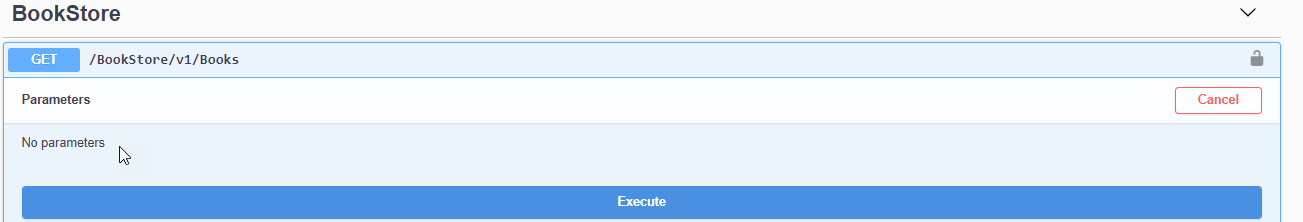
As shown above, POST, GET, PUT and DELETE are the HTTP methods used for CRUD operations. The following table shows the description of each of these methods as well as an example URL using the swagger tool [***https://demoqa.com/swagger/***](https://demoqa.com/swagger/)

| **HTTP Method** | **Operation** | **Operation Type** | **Example URL** |
| --- | --- | --- | --- |
| GET | Get the list of books | Read Only | ***curl -X GET "https://demoqa.com/BookStore/v1/Books" -H "accept: application/json"*** |
| POST | Add list of books | Non-Idempotent | ***curl -X POST "https://demoqa.com/BookStore/v1/Books" -H "accept: application/json" -H "Content-Type: application/json" -d "{ "userId": "toolsqa\_test", "collectionOfIsbns": [ { "isbn": "9781449325862" } ]}"*** |
| PUT | Replace ISBN object with given ISBN | N/A | ***curl -X PUT "https://demoqa.com/BookStore/v1/Books/9781449325889" -H "accept: application/json" -H "Content-Type: application/json" -d "{ "userId": "toolsqa\_test", "isbn": "9781449325862"}"*** |
| DELETE | Delete book with given ISBN | Idempotent: Same results irrespective of how many times the operation is invoked. | ***curl -X DELETE "https://demoqa.com/BookStore/v1/Book" -H "accept: application/json" -H "Content-Type: application/json" -d "{ "isbn": "9781449325862", "userId": "toolsqa\_test"}"*** |

***How to test GET operation using REST APIs?***

So how do we test these methods in a Swagger tool? Let us see an example of *GET* operation. Navigate to the following link: [***https://demoqa.com/swagger/#/BookStore/BookStoreV1BooksGet***](https://demoqa.com/swagger/#/BookStore/BookStoreV1BooksGet)

We can see the following REST API for *GET* book details.



Now let us ***"execute"*** the GET operation that ***"gets book details".*** When we click this button and execute the API, the below-given command or GET syntax gets executed:

curl -X GET "https://demoqa.com/BookStore/v1/Books" -H "accept: application/json"

On performing the above operation we get the following response.



In a similar manner, we can perform other operations as well for which we have crafted dedicated posts. In subsequent articles, we will learn to create a REST API ourselves.

So here are some of the points we should remember for REST API methods.

* *GET methods are read-only and safe.*
* *PUT and DELETE methods are idempotent: the response they return is always the same irrespective of the times the methods are invoked.*
* *PUT and POST operation are generally the same except for one difference that POST operation can return different results whereas PUT is idempotent.*

**Steps to write a Test Script Using Rest Assured**

1. Perform a maven build to import all dependencies, again you will find help on Maven set up on guru99.
2. Still, you see errors, then do a maven clean followed by a maven install, and it should build without any errors.
3. You can add the below lines in your java class and see no compile errors are present.

import io.restassured.RestAssured.\*;

import io.restassured.matcher.RestAssuredMatchers.\*;

import org.hamcrest.Matchers.\*;

**First simple Rest Assured script**

**Syntax:**

The syntax of Rest Assured.io is the most beautiful part, as it is very BDD like and understandable.

Given().

param("x", "y").

header("z", "w").

when().

Method().

Then().

statusCode(XXX).

body("x, ”y", equalTo("z"));

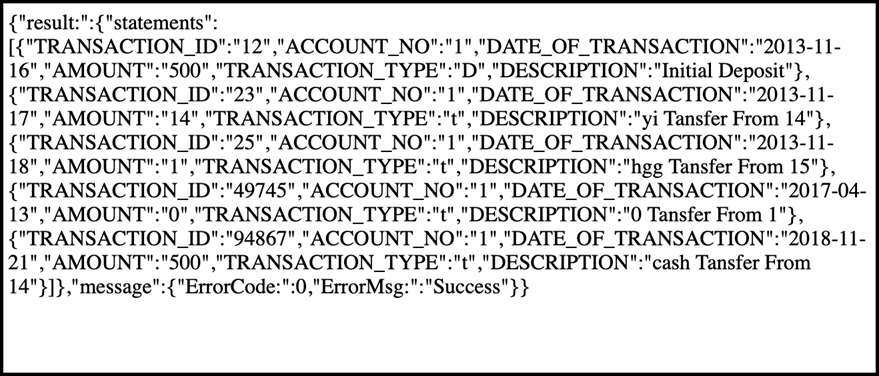
**Explanation:**

| **Code** | **Explanation** |
| --- | --- |
| Given() | ‘Given’ keyword, lets you set a background, here, you pass the request headers, query and path param, body, cookies. This is optional if these items are not needed in the request |
| When() | ‘when’ keyword marks the premise of your scenario. For example, ‘when’ you get/post/put something, do something else. |
| Method() | Substitute this with any of the CRUD operations(get/post/put/delete) |
| Then() | Your assert and matcher conditions go here |

Now that you have the setup and some background to the syntax, let’s create our first simple test. It is okay if so far the structure seems new to you, as you code further interpret each line, you will get the hang of it.

**What will you fetch?**

Open your browser and hit – http://demo.guru99.com/V4/sinkministatement.php?CUSTOMER\_ID=68195&PASSWORD=1234!&Account\_No=1. Ensure you see something as below.

[](https://www.guru99.com/images/2/041720_1114_RESTAssured2.png)

In case you get an error on the browser when you try to get a response for the request,

1. See if you have used Https or Http. Your browser might have settings to not open insecure websites.
2. See if you have any proxy or firewall blocks your browser from opening websites.

\***Note** – you did not use any headers here, no body, and no cookie. It was a URL and also you are getting content from the API and not posting or updating any existing content, so that makes it a GET call. Remember this to understand our first test better.

**The Objective of your test:**

The goal of the script is to print the same output on your IDE console as what you received on the browser through Rest Assured.

Let us code this with the below steps:

**Getting the response Body**

**Step 1)** Create a class named as “myFirstRestAssuredClass”

**Step 2)** Create a method called “getResponseBody”

**Step 3)** Similar to the structure learned earlier of given, when and then, type the below code

given(). -> No headers required, no query or path param.

when(). -> No specific condition setup

get(‘http://demo.guru99.com/V4/sinkministatement.php?CUSTOMER\_ID=68195&PASSWORD=1234!&Account\_No=1‘). ->only the url needs to be supplied

then(). -> No specific assertions required

log(). all() -> Once all the response is fetched, log response, headers, essentially everything that the request returns to you.

public static void getResponseBody(){

given().when().get("http://demo.guru99.com/V4/sinkministatement.php?CUSTOMER\_ID=68195&PASSWORD=1234!&Account\_No=1").then().log()

.all();

}

Now notice that the URL used is long and less readable, if you look closely, you will notice that 3 query parameters are being used which are

1. Customer\_ID
2. Password
3. Account\_No

Rest Assured, helps us pass every part(query, path, header param) separately, making the code more readable and easy to maintain. Also, we can parameterize the data from an external file as required.

For using query param, we go back to our definition of the syntax and see that all of them are passed as a part of given.

public static void getResponseBody(){

given().queryParam("CUSTOMER\_ID","68195")

.queryParam("PASSWORD","1234!")

.queryParam("Account\_No","1")

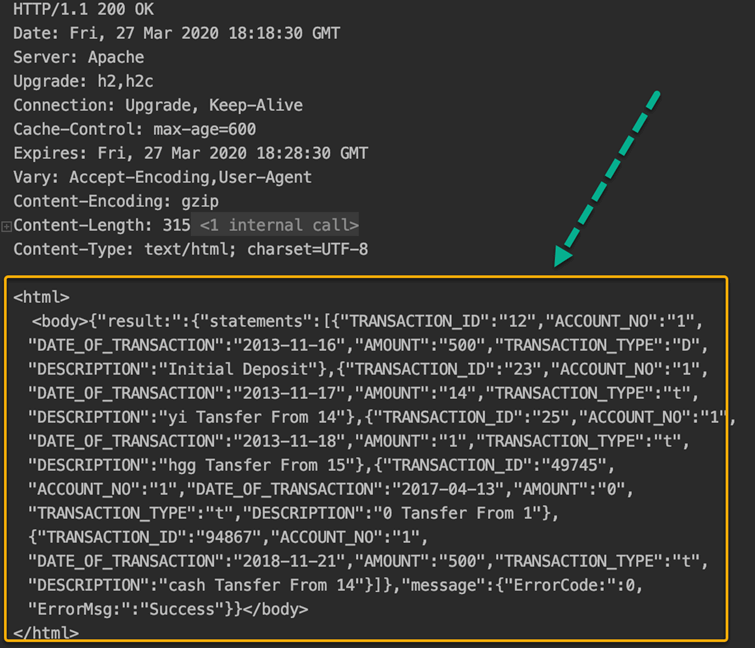
.when().get("http://demo.guru99.com/V4/sinkministatement.php").then().log()

.body();

}

\*\*Note that we used “body” instead of “all”; this helps us to extract only the body of the response.

**Output:**

[](https://www.guru99.com/images/2/041720_1114_RESTAssured3.png)Output for getResponseBody

**Getting the response status code**

The next method that we script will be to get the status code and also put an assertion to validate the same.

**Step 1)** Create a method called getResponseStatus()

**Step 2)** Use the same request structure used above. Copy and paste it.

**Step 3)** Instead of logging it, we use the ‘getStatusCode’ inbuilt method of Rest Assured to fetch the status code value

**Step 4)** In order to assert that your status code is 200, we use the keywords – assertThat().statusCode(expectedCode)

\*\*Note – URL is a variable used for simplicity. URL holds the entire API request URL

public static void getResponseStatus(){

int statusCode= given().queryParam("CUSTOMER\_ID","68195")

.queryParam("PASSWORD","1234!")

.queryParam("Account\_No","1") .when().get("http://demo.guru99.com/V4/sinkministatement.php").getStatusCode();

System.out.println("The response status is "+statusCode);

given().when().get(url).then().assertThat().statusCode(200);

}

**Output:**

[Output for getResponseStatus](https://www.guru99.com/images/2/041720_1114_RESTAssured4.png)Output for getResponseStatus

**Business Need**

One of the basic rules of automation is that we have to put checkpoints so that the test proceeds only if all the required conditions are met. In API testing, the most basic validation is to check if the status code of the request is in 2XX format.

The complete code, so far:

import java.util.ArrayList;

import static io.restassured.RestAssured.\*;

import static java.util.concurrent.TimeUnit.MILLISECONDS;

public class myFirstRestAssuredClass {

final static String url="http://demo.guru99.com/V4/sinkministatement.php?CUSTOMER\_ID=68195&PASSWORD=1234!&Account\_No=1";

public static void main(String args[]) {

getResponseBody();

getResponseStatus();

; }

//This will fetch the response body as is and log it. given and when are optional here

public static void getResponseBody(){

given().when().get(url).then().log()

.all();

given().queryParam("CUSTOMER\_ID","68195")

.queryParam("PASSWORD","1234!")

.queryParam("Account\_No","1") .when().get("http://demo.guru99.com/V4/sinkministatement.php").then().log().body();

}

public static void getResponseStatus(){

int statusCode= given().queryParam("CUSTOMER\_ID","68195")

.queryParam("PASSWORD","1234!")

.queryParam("Account\_No","1")

.when().get("http://demo.guru99.com/V4/sinkministatement.php").getStatusCode();

System.out.println("The response status is "+statusCode);

given().when().get(url).then().assertThat().statusCode(200);

}

}

**\*Note:**

1. 200 is a successful response for this scenario. At times, you need the request to fail as well, and then you might use 4XX or 5XX. Do try to change the status code by supplying invalid parameters and check.
2. When we assert a condition, there will be no printing on the console unless there is an error.

**Script to fetch different parts of a response**

Fetching response body and response status code is already covered in the above segment. It is worthy to note that to fetch different parts of the response, the keyword ‘extract’ is very important.

**Header**

Rest Assured is a very straightforward language, and fetching headers is just as simple. The method name is headers(). Like before, we will create a standalone method to do the same.

public static void getResponseHeaders(){

System.out.println("The headers in the response "+

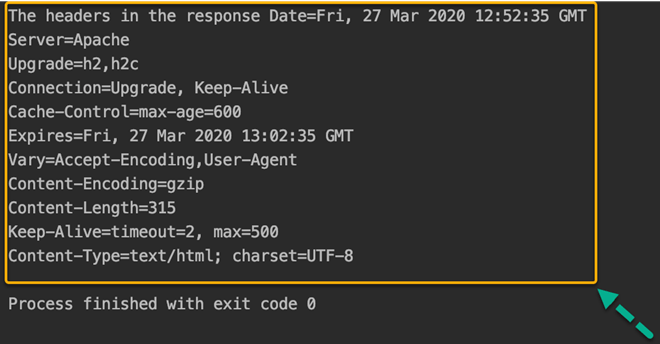
get(url).then().extract()

.headers());

}

Please note that ‘given().when()’ is skipped here, and the code line starts from get(), this is because there is no precondition or verification made here to hit the request and get a response. In such cases, it’s optional to use the same.

**Output :**

[](https://www.guru99.com/images/2/041720_1114_RESTAssured5.png)Output for getResponseHeader

**Business Need:**

Quite a few times, you would need to use the authorization token, or a session cookie for the subsequent request, and mostly, these details are returned as headers of the response.

**Response Time**

To get the time needed to fetch the response from the backend or other downstream systems, Rest Assured provides a method called ‘timeIn’ with a suitable timeUnit to get the time taken to return the response.

public static void getResponseTime(){

System.out.println("The time taken to fetch the response "+get(url)

.timeIn(TimeUnit.MILLISECONDS) + " milliseconds");

}

**Output:**

[Output for getResponseTime](https://www.guru99.com/images/2/041720_1114_RESTAssured6.png)Output for getResponseTime

**Business Need:**

A very important feature of testing APIs is their response time, to measure the performance of the application. Note that the time taken for your call may take more or less time depending on your internet speed, the performance of the API at that time, server load, and other factors impacting the time.

**Content-Type**

You can get the content-Type of the response returned using the method is “contentType ()”.

public static void getResponseContentType(){

System.out.println("The content type of response "+

get(url).then().extract()

.contentType());

}

Output

[Output for getContentType](https://www.guru99.com/images/2/041720_1114_RESTAssured7.png)Output for getContentType

**Business Need:**

At times getting the content-type is essential for ensuring there are no security gaps for any cross-origin threats or just to ensure the content passed is as per the standards of the API.

**Fetch Individual JSON Element**

From the given response, you are asked to calculate the total amount, you need to fetch every amount and sum it up.

**Steps:**

**Step 1)** The amount field is within an array with Key “statements” which is in turn in the list with key “result”

**Step 2)** Rest Assured, provides a mechanism to reach the values in the API using “path”

**Step 3)** The path to reach amounts is “result.statements.AMOUNT”. Think of it like Xpath in selenium.

**Step 4)** Fetch all amounts in a collection, and then loop for all values to calculate the sum

public static void getSpecificPartOfResponseBody(){

ArrayList<String> amounts = when().get(url).then().extract().path("result.statements.AMOUNT") ;

int sumOfAll=0;

for(String a:amounts){

System.out.println("The amount value fetched is "+a);

sumOfAll=sumOfAll+Integer.valueOf(a);

}

System.out.println("The total amount is "+sumOfAll);

}

Note: Since the amount value is in string data type, we convert to integer and use it for summation.

**Output:**

[](https://www.guru99.com/images/2/041720_1114_RESTAssured8.png)

**Authentication in Rest Assured**

**The tool provides support for several authentication schemes**:

* Basic Authentication
* Digest Authentication
* Form Authentication
* OAuth 1 and OAuth 2

**Using Basic Authentication**

[The basic authentication scheme](https://tools.ietf.org/html/rfc7617) requires the consumer to send user id and a password encoded in *Base64*.

REST Assured provides an easy way to configure the credentials that the request requires:

given().auth()

.basic("user1", "user1Pass")

.when()

.get("http://localhost:8080/spring-security-rest-basic-auth/api/foos/1")

.then()

.assertThat()

.statusCode(HttpStatus.OK.value());Copy

**2.1. Preemptive Authentication**

As we’ve seen on [a previous post on Spring Security authentication](https://www.baeldung.com/spring-security-basic-authentication#usage), a server might use [a challenge-response mechanism](https://tools.ietf.org/html/rfc2617#section-1.2) to indicate explicitly when the consumer needs authenticate to access the resource.

**By default, REST Assured waits for the server to challenge before sending the credentials.**

This can be troublesome in some cases, for example, where the server is configured to retrieve a login form instead of the challenge response.

For this reason, the library provides the *preemptive*directive that we can use:

given().auth()

.preemptive()

.basic("user1", "user1Pass")

.when()

// ...Copy

With this in place, REST Assured will send the credentials without waiting for an *Unauthorized* response.

We hardly ever are interested in testing the server’s ability to challenge.**Therefore, we can normally add this command to avoid complications and the overhead of making an additional request.**

**3. Using Digest Authentication**

Even though this is also considered a [“weak” authentication method](https://tools.ietf.org/html/rfc2617#section-4.4), using [Digest Authentication](https://tools.ietf.org/html/rfc7616) represents an advantage over the basic protocol.

This is due to the fact that this scheme avoids sending the password in cleartext.

**Despite this difference, implementing this form of authentication with REST Assured is very similar to the one we followed in the previous section:**

given().auth()

.digest("user1", "user1Pass")

.when()

// ...Copy

Note that, currently, the library supports only challenged authentication for this scheme,**so we can’t use *preemptive()*as we did earlier.**

**4. Using Form Authentication**

Many services provide an HTML form for the user to authenticate by filling in the fields with their credentials.

When the user submits the form, the browser executes a POST request with the information.

Normally, the form indicates the endpoint that it’ll call with its *action* attribute, and each *input*field corresponds with a form parameter sent in the request.

If the login form is simple enough and follows these rules, then we can rely on REST Assured to figure out these values for us:

given().auth()

.form("user1", "user1Pass")

.when()

// ...Copy

This is not an optimal approach, anyway, since REST Assured needs to perform an additional request and parse the HTML response to find the fields.

We also have to keep in mind that the process can still fail, for example, if the webpage is complex, or if the service is configured with a context path that is not included in the *action* attribute.

**Therefore, a better solution is to provide the configuration ourselves, indicating explicitly the three required fields:**

given().auth()

.form(

"user1",

"user1Pass",

**new** **FormAuthConfig**("/perform\_login", "username", "password"))

// ...

Copy

Apart from these basic configurations, REST Assured ships with functionality to:

* detect or indicate a CSRF token field in the webpage
* use additional form fields in the request
* log information about the authentication process

**5. OAuth Support**

OAuth is technically an *authorization* framework, and it doesn’t define any mechanism for authenticating a user.

Still, it can be used as the basis for building an authentication and identity protocol, as is the case of [OpenID Connect](https://www.baeldung.com/spring-security-openid-connect).

**OAuth2 Authentication:**

OAuth2 (Open Authorization 2.0) is a widely used authorization framework that allows applications to obtain limited access to user accounts on behalf of a third-party application. It’s commonly used to enable secure and controlled access to APIs and resources without exposing the user’s credentials.

OAuth2 involves various roles, including the resource owner (user), client application (third-party app), authorization server (handles authentication and issues access tokens), and resource server (holds the protected resources). The process revolves around obtaining an access token, which serves as a temporary authorization token that allows the client application to access specific resources on behalf of the user.

|  |
| --- |
| Response resp3 = given()     .auth()     .oauth2(“access\_token”)     .when().get(“https://api.example.com/resource”); |

With .oauth2(“access\_token”), Rest Assured automatically integrates the token into the request.

## What Is a Multipart Request?

Multipart requests are a type of HTTP POST request. **They allow sending various files or data in a single request.**

In a multipart request, the data is divided into multiple parts. Each part has a name and starts with its own set of headers that indicates the type of data it contains. The data and boundaries between each part are encoded.

**public** **class** UploadAFileinRestAssured {

**public** **static** **void** main(String[] args) {

String str="abcd";

RestAssured.*baseURI*="https://httpbin.org/post";

File f=**new** File("D:\\03rdOctoberFileNew\\test1567.txt");

File f1=**new** File("C:\\Users\\saura\\OneDrive\\Pictures\\Test1234.jpeg");

String path= f.getPath();

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

String Response= *given*().relaxedHTTPSValidation().

log().all().header("Content-Type","multipart/form-data")

.multiPart("file",f)

.multiPart("file",f1)

.multiPart("text",str)

.when().post().then().assertThat().statusCode(200).

log().all().extract().response().asString();

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

JsonPath js=**new** JsonPath(Response);

String FileData=js.getString("files.file");

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

Session Filter in Rest Assured

A **session filter** can be used record the **session** id returned from the server as well as automatically apply this **session** id in subsequent requests.



**Relaxed HTTPS Validation**

Use relaxed HTTP validation. This means that you'll trust all hosts regardless if the SSL certificate is invalid. By using this method you don't need to specify a keystore (see [keyStore(String, String)](https://www.javadoc.io/static/io.rest-assured/rest-assured/3.0.1/io/restassured/config/SSLConfig.html" \l "keyStore-java.lang.String-java.lang.String-) or trust store (see [trustStore(java.security.KeyStore)](https://www.javadoc.io/static/io.rest-assured/rest-assured/3.0.1/io/restassured/config/SSLConfig.html" \l "trustStore-java.security.KeyStore-). This method assumes that the protocol for the [SSLContext](http://docs.oracle.com/javase/1.5.0/docs/api/javax/net/ssl/SSLContext.html?is-external=true" \o "class or interface in javax.net.ssl) instance is . If this is not the case use [relaxedHTTPSValidation(String)](https://www.javadoc.io/static/io.rest-assured/rest-assured/3.0.1/io/restassured/config/SSLConfig.html#relaxedHTTPSValidation-java.lang.String-).

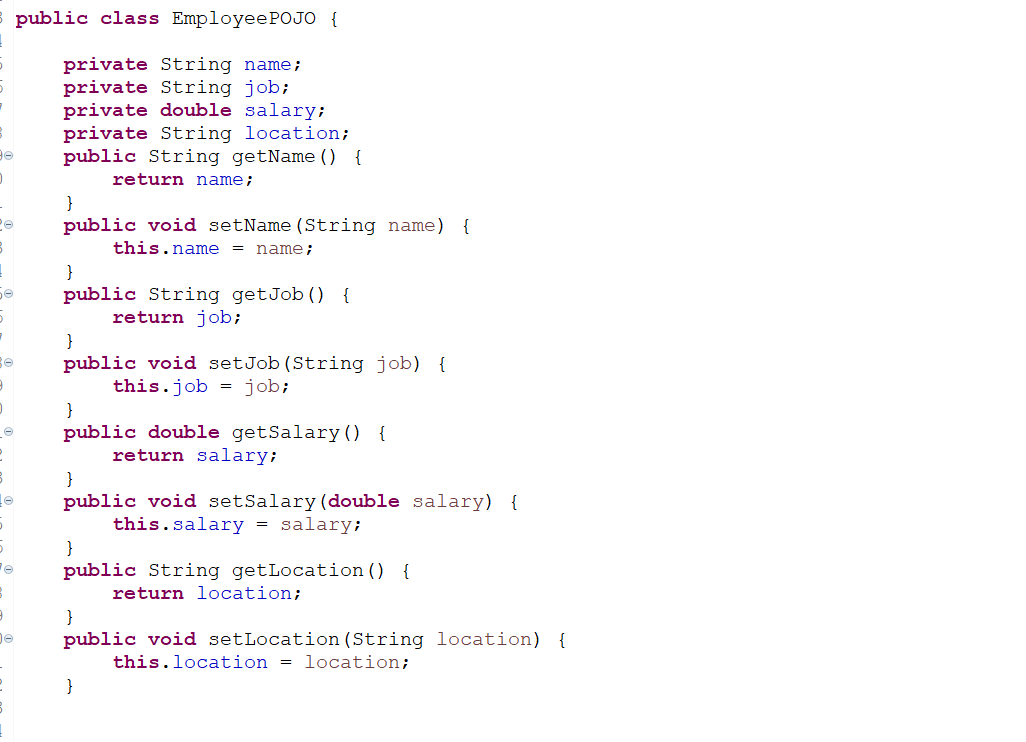
****

**Serialization and Deserialization in Rest Assured**

In Java, Serialization is a process of converting an object from its current state to a stream of bytes which can be written to a file or transported through a network or stored in a database.  
De-serialization is rebuilding the object from stream of bytes.

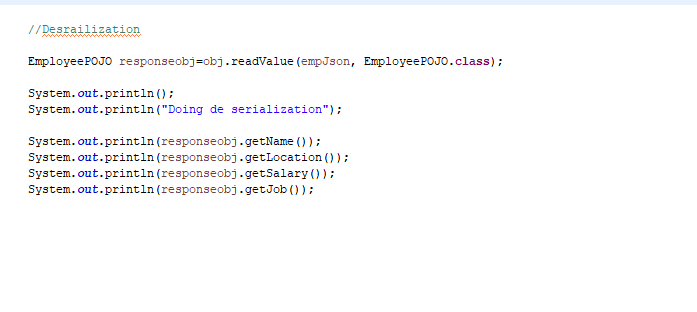
POJO Class

* An ordinary Java class that does not extends or implement properties from any technology or framework-related classes and interface is called a POJO class.
* POJO classes are used to represent data.
* POJO Class will contain only default constructor, private data members, and public setter and getter methods for every data member.
* Setter methods are used to set the value to the variable.
* Getter methods are used to get the value from the variable.
* Basically, POJO defines an entity.

****

****

**Using Deserialization**

****

**Request Specification**

‘RequestSpecification’ interface provided by Rest Assured is used to club and extract repetitive actions like setting up base URL, headers, HTTP verbs etc which may be common for multiple Rest calls. The extracted common code can be used in different requests thereby reducing number of lines of code and increasing maintainability.

**Illustration:**

Consider following JUnit test cases which are written to test an API endpoint.



As evident, most of the request configurations as highlighted by red block are common to both the test cases yet they are duplicated. In an exhaustive test suite this can easily become and overhead to write all these duplicate codes which is also hard to maintain in case some data needs to be changed across all test cases.

Request specifications come handy in such situation by extracting these common test parameters as a separate entity and used across all the test cases. Below example demonstrates the same



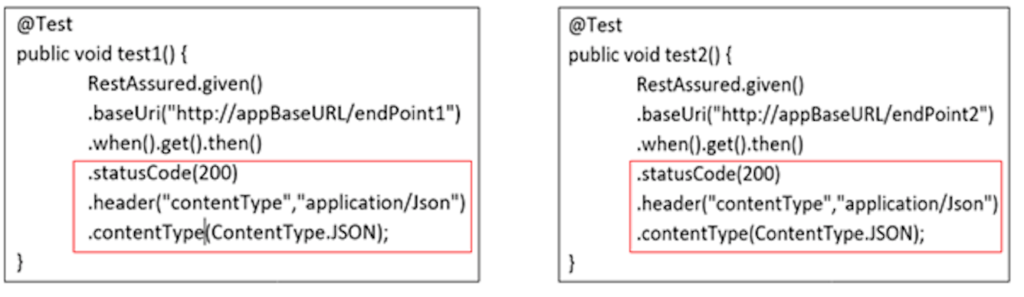
As depicted in above figure, we extracted the common request setup in a separate base test case using RequestSpecification interface and used it in the actual test cases. Hence overcoming redundancy and enhancing reusability and maintainability of the code.

**Response Specification**

‘ResponseSpecification’ interface comes handy in situation where similar set of assertions need to be done for several Rest requests. It achieves this by grouping common assertions into a ResponseSpecBuilder instance and using this instance for validations in multiple tests.

**Illustration:**

Consider these tests for a hypothetical API.



As depicted, both these tests have 3 common assertions (as highlighted by red rectangle) which are repeated. Now if the API behaviour changes to return different content type, the assertions at 2 places would need to be corrected. This becomes a maintenance overhead.

Below diagram illustrates how the Response Specification helps in removing this redundancy.