**REST API Functional Test Automation**

**Webservices:**

***Services available over the web***

*Communication between Apps over the web*

Provides standard protocol/format for communication

Enables platform independent communication

Client – service consumer / Server – service provider

Request / Response

Medium – HTTP / Internet + Format - XML/JSON

|  |  |  |  |
| --- | --- | --- | --- |
| Implementations |  | Medium | FORMAT |
| SOAP | Simple Object Access Protocol | HTTP(POST) | XML |
| REST | Representational state transfer | HTTP | XML/JSON |

**Components of Web Services** : **WSDL** and **UDDI**

Consumer needs to know: services available, request and response parameters, how to call the webservice, structure and entire description.

Service provider publishes an **interface** for his web services that describes all the attributes and functionalities of webservices.

This is XML based interface and is known as **WSDL [Web Services Description Language].**

A web service provider publishes his web service through WSDL on **online** **Directory** from where consumers can query and search the web services. This online registry / directory is called **UDDI [Universal Description, Discovery and Integration].**

UDDI – XML based standard for publishing and finding web services.

**SOAP web services:**

Any web service that complies with SOAP web services specifications.

Specifications / standards defined by W3C (World wide web consortium)

Specifications: 1. Basic 2. Extended

Basic: SOAP/WSDL/UDDI

Extended: WS security/ WS policy etc...

SOAP: Protocol/ rules/ definitions how two applications communicates with each other over the web.

All messages/ information exchange over a common format i.e. XML.

XML message has to be a SOAP message: specific structure

SOAP message: consists of

Envelope – root element/ basic unit / consists of Header and body.

Header – info about message / includes authentication / optional.

Body – actual request and its parameters.

**REST web services:**

Representational State Transfer

A web service exchanges data b/w two apps using REST principles.

**An architectural style**

Defines **set of principles/rules while designing a service** for communication / data exchange b/w two apps.

This leads to RESTful web service.

**REST architecture: constraints/ principles**

**Uniform Interface:**

Identification of Resources – **Resources + URI + HTTP.**

Manipulation of resources using Representations.

*Self-descriptive messages.*

Resource: everything is resource.

Define any module / data as a resource.

URI: Uniform Resource Identifier

Access any Module/Data

HTTP:

Explicit use of HTTP methods [GET/POST/PUT/DELETE].

**Statelessness:**

All client -server communications are stateless.

*Each request from the client to the server must contain all of the data necessary to handle the request.*

Server should not require storing the state of the session –

improves Web service performance.

If the concept of a session is required all information should be stored at client side and sent with every subsequent request.

Every request (call) from client is independent and a complete entity in itself.

**Caching:**

*happens at* ***client******side****.*

data within a response to a request must be labelled implicitly / explicitly as cacheable or non-cacheable.

Server generates responses that indicate whether they are cacheable or not, to improve the performance by reducing the number of requests for duplicate resources.

Server does this by including a **Cache–control** and **Last -Modified** (data value) in **HTTP Response Headers**

**Note:** Etag: entity tag / ***every version of the resource will have Etag.***

Client uses Cache–control header to determine whether to cache the resource (make a local copy) or not.

**Layering:** to improve **scalability**.

Multiple layers can exist between client and server.

Layers are **HTTP intermediaries**, used for *message translations / improving performance with caching.*

Layers can be **Proxies** and **Gateways**.

**Proxies**: client can send request to proxy server, proxy server evaluates the request to simplify or control its complexity.

**Gateways**: for managing **traffic** on the N/w, protocol translations @ Server end.

**A layer for caching**: cache a response and stores for a particular time and if a new request for the same resource from a client comes within this time period and there is no changes in the resource then response will be sent from the layer which will improve the **performance** and **scalability**.

**Code-on-demand:** optional

Ability to download and execute code @ client side.

*To a JavaScript in a* ***web browser******Code-on-demand*** *allows to add a new functionality without re-deploying client software.*

Ex: client requests a resource and server returns resource with some JavaScript.

**REST web services: summary**

A Resource’s Representation is transferred b/w client and server.

***Representation = description of the current state of the resource.***

**states** can be: **XML/JSON/YML** etc.

**REST** was coined by **Roy Fielding** in 2000.

**Fundamentals of REST web services Test:**

Ensure API does what it is designed to do.

API can handle assigned load.

APIs work across **browsers**, **devices** and **systems** (**platforms**).

To verify whether API meets expectations on functionality, reliability, performance and security.

**API Test process:**

1. **API specification review**

Review API specification and use case documentation from Test perspective.

1. **Test environment set up with required parameters.**
2. **Combining Application data**

To ensure API functionality against possible inputs.

1. **Identify types of API Test to be performed.**
2. **Test execution and reporting.**

|  |  |
| --- | --- |
| ***API Test types*** | ***Description*** |
| **Functional** Test | correctness of functionality |
| **Reliability** Test | consistent Test results |
| **Load** Test | API can handle load |
| **UI and UX Test** | end to end functionality using GUI or CLI |
| **Interoperability Test** | **for SOAP services** |
| Security Test | API is safe against external Threats |
| Penetration Test | vulnerabilities of an application |
| **Negative Test** | for wrong inputs |

**Types of bugs:**

Duplicate functionality

Reliability issues

**Improper messaging**

**In compatible error handling**

Security issues

Performance issues

Multithreaded issues.

**Benefits of API Test:**

Language independent.

GUI independent.

Improved Test coverage

Reduces Testing costs.

Faster release – increases ROI.

**Challenges of API Test:**

No GUI

**Synchronous and Asynchronous dependency**.

Test data management.

Request / Response data formats [XML/JSON].

No documentation and Time constraints.

**JSON**

A data representation format

JavaScript Object Notation is an open-standard file format or data interchange format that uses human-readable text to transmit data objects consisting of **attribute–value pairs** and **array data types**.

*JSON (JavaScript Object Notation) is a lightweight data-interchange format.*

It is easy for humans to read and write.

It is easy for machines to parse and generate.

*JSON is a syntax for storing and exchanging data.*

*JSON is text, written with JavaScript object notation.*

**Exchanging Data**

*When exchanging data between a* ***browser and a server****, the data can only be text.*

JSON is text, and we can convert any JavaScript object into JSON, and send JSON to the server.

We can also convert any JSON received from the server into JavaScript objects.

This way we can work with the data as JavaScript objects, with no complicated parsing and translations.

**Sending Data**

If you have data stored in a JavaScript object, you can convert the object into JSON, and send it to a server:

var myObj = {name: "John", age: 31, city: "New York"};  
var myJSON = JSON.**stringify**(myObj);  
window.location = "demo\_json.php?x=" + myJSON;

**Receiving Data**

If you receive data in JSON format, you can convert it into a JavaScript object:

var myJSON = '{"name":"John", "age":31, "city":"New York"}';  
var myObj = JSON.**parse**(myJSON);  
document.getElementById("demo").innerHTML = myObj.name;

**Storing Data**

When storing data, the data has to be a certain format, and regardless of where you choose to store it, text is always one of the legal formats.

***JSON makes it possible to store JavaScript objects as text.***

Example:

**// Storing data:**  
myObj = {name: "John", age: 31, city: "New York"};  
myJSON = JSON.**stringify(myObj);**localStorage.**setItem**("testJSON", myJSON);

**// Retrieving data:**  
text = localStorage.getItem("testJSON");  
obj = JSON**.parse(text);**document.getElementById("demo").innerHTML = obj.name;

The **JSON format** was originally specified by **Douglas Crockford**.

**Why use JSON?**

Since the JSON format is text only, it can easily be sent to and from a server, and used as a data format by any programming language.

JavaScript has a built in function to convert a string, written in JSON format, into native JavaScript objects:

**JSON.parse()**

So, if you receive data from a server, in JSON format, you can use it like any other JavaScript object.

**JSON Syntax Rules**

JSON syntax is derived from JavaScript object notation syntax:

Data is in name/value pairs

Data is separated by commas

**Curly braces hold objects**

**Square brackets hold arrays**

**JSON Data - A Name and a Value**

JSON data is written as name/value pairs.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

**Example**

"name":"John"

JSON names require double quotes. JavaScript names don't.

**JSON - Evaluates to JavaScript Objects**

The JSON format is almost identical to JavaScript objects.

In JSON, keys must be strings, written with double quotes:

**JSON**

{ "name":"John" }

In JavaScript, keys can be strings, numbers, or identifier names:

**JavaScript**

{ name:"John" }

**JSON Values**

In **JSON**, values must be one of the following data types:

a string

a number

an object (JSON object)

an array

a boolean

null

In **JavaScript** values can be all of the above, plus any other valid JavaScript expression, including:

a function

a date

undefined

In JSON, string values must be written with double quotes:

**JSON**

{ "name":"John" }

**JSON Uses JavaScript Syntax**

Because JSON syntax is derived from JavaScript object notation, very little extra software is needed to work with JSON within JavaScript.

With JavaScript you can create an object and assign data to it, like this:

var person = { name: "John", age: 31, city: "New York" };

You can access a JavaScript object like this:

// returns John  
person.name;

It can also be accessed like this:

// returns John  
person["name"];

Data can be modified like this:

person.name = "Gilbert";

It can also be modified like this:

person["name"] = "Gilbert";

**JSON Files**

The file type for JSON files is **".json**"

The MIME type for JSON text is "**application/json**"

**Note:**

**Multipurpose Internet Mail Extension** or **MIME** is an **internet standard, encoded file format used by email programs**. The mime format contains **8-bit encoded data** instead of commonly used 7-bit encoding for sending email.

**JSON vs XML**

Both JSON and XML can be used to receive data from a web server.

The following JSON and XML examples both define an employee’s object, with an array of 3 employees:

JSON Example

{"employees":[  
  { "firstName":"John", "lastName":"Doe" },  
  { "firstName":"Anna", "lastName":"Smith" },  
  { "firstName":"Peter", "lastName":"Jones" }  
]}

XML Example

<employees>  
  <employee>  
    <firstName>John</firstName> <lastName>Doe</lastName>  
  </employee>  
  <employee>  
    <firstName>Anna</firstName> <lastName>Smith</lastName>  
  </employee>  
  <employee>  
    <firstName>Peter</firstName> <lastName>Jones</lastName>  
  </employee>  
</employees>

 JSON is Like XML Because

Both JSON and XML are "self describing" (human readable)

Both JSON and XML are hierarchical (values within values)

Both JSON and XML can be parsed and used by lots of programming languages

Both JSON and XML can be fetched with an XMLHttpRequest

JSON is Unlike XML Because

JSON doesn't use end tag

JSON is shorter

JSON is quicker to read and write

JSON can use arrays

The biggest difference is:

***XML has to be parsed with an XML parser. JSON can be parsed by a standard JavaScript function.***

**Why JSON is Better Than XML?**

*XML is much more difficult to parse than JSON.  
JSON is parsed into a ready-to-use JavaScript object.*

*For AJAX applications, JSON is faster and easier than XML:*

**Using XML**

Fetch an XML document

Use the XML DOM to loop through the document

Extract values and store in variables

**Using JSON**

Fetch a JSON string

JSON.Parse the JSON string

**JSON** **Data Types**

**Valid Data Types**

In JSON, values must be one of the following data types:

a string

a number

an object (JSON object)

an array

a boolean

null

JSON values **cannot**be one of the following data types:

a function

a date

undefined

**JSON Strings**

Strings in JSON must be written in double quotes.

Example

{ "name":"John" }

**JSON Numbers**

Numbers in JSON must be an integer or a floating point.

Example

{ "age":30 }

**JSON Objects**

Values in JSON can be objects.

Example

{  
"employee":{ "name":"John", "age":30, "city":"New York" }  
}

*Objects as values in JSON must follow the same rules as JSON objects.*

**JSON Arrays**

Values in JSON can be arrays.

Example

{  
"employees":[ "John", "Anna", "Peter" ]  
}

**JSON Booleans**

Values in JSON can be true/false.

Example

{ "sale":true }

**JSON null**

Values in JSON can be null.

Example

{ "middlename":null }

**JSON.parse()**

A common use of JSON is to exchange data to/from a web server.

**When receiving data from a web server, the data is always a string.**

Parse the data with **JSON.parse(),** and the data becomes a **JavaScript object**.

**Example - Parsing JSON**

Imagine we received this text from a web server:

'{ "name":"John", "age":30, "city":"New York"}'

Use the JavaScript function JSON.parse() to convert text into a JavaScript object:

var obj = JSON.parse('{ "name":"John", "age":30, "city":"New York"}');

Make sure the text is written in JSON format, or else you will get a syntax error.

Use the JavaScript object in your page:

**Example**

<p id="demo"></p>  
  
<script>  
document.getElementById("demo").innerHTML = obj.name + ", " + obj.age;  
</script>

**JSON From the Server**

You can request JSON from the server by using an AJAX request

As long as the response from the server is written in JSON format, you can parse the string into a JavaScript object.

**Example**

Use the XMLHttpRequest to get data from the server:

var xmlhttp = new XMLHttpRequest();  
xmlhttp.onreadystatechange = function() {  
  if (this.readyState == 4 && this.status == 200) {  
    var myObj = JSON.parse(this.responseText);  
    document.getElementById("demo").innerHTML = myObj.name;  
  }  
};  
xmlhttp.open("GET", "json\_demo.txt", true);  
xmlhttp.send();

**Array as JSON**

When using the **JSON.parse()** on a **JSON derived from an array**, the method will return a **JavaScript array**, instead of a JavaScript object.

**Example**

The JSON returned from the server is an array:

var xmlhttp = new XMLHttpRequest();  
xmlhttp.onreadystatechange = function() {  
  if (this.readyState == 4 && this.status == 200) {  
    var myArr = JSON.parse(this.responseText);  
    document.getElementById("demo").innerHTML = myArr[0];  
  }  
};  
xmlhttp.open("GET", "json\_demo\_array.txt", true);  
xmlhttp.send();

**Exceptions**

**Parsing Dates**

Date objects are not allowed in JSON.

If you need to include a date, write it as a string.

You can convert it back into a date object later:

**Example**

Convert a string into a date:

var text = '{ "name":"John", "birth":"1986-12-14", "city":"New York"}';  
var obj = JSON.parse(text);  
obj.birth = new Date(obj.birth);  
  
document.getElementById("demo").innerHTML = obj.name + ", " + obj.birth;

Or, you can use the second parameter, of the JSON.parse() function, called **reviver.**

***The*reviver*parameter is a function that checks each property, before returning the value.***

**Example**

Convert a string into a date, using the reviver function:

var text = '{ "name":"John", "birth":"1986-12-14", "city":"New York"}';  
var obj = JSON.parse(text, function (key, value) {  
  if (key == "birth") {  
    return new Date(value);  
  } else {  
    return value;  
  }  
});  
  
document.getElementById("demo").innerHTML = obj.name + ", " + obj.birth;

**Parsing Functions**

Functions are not allowed in JSON.

If you need to include a function, write it as a string.

You can convert it back into a function later:

**Example**

Convert a string into a function:

var text = '{ "name":"John", "age":"function () {return 30;}", "city":"New York"}';  
var obj = JSON.parse(text);  
obj.age = eval("(" + obj.age + ")");  
  
document.getElementById("demo").innerHTML = obj.name + ", " + obj.age();

You should avoid using functions in JSON, the functions will lose their scope, and you would have to use **eval()** to convert them back into functions.

**Browser Support**

The JSON.parse() function is included in all major browsers and in the latest ECMAScript (JavaScript) standard.

The numbers in the table below specifies the first browser version that fully supports the JSON.parse() function:

JSON.stringify()

A common use of JSON is to exchange data to/from a web server.

When sending data to a web server, the data has to be a string.

Convert a JavaScript object into a string with JSON.stringify().

**Stringify a JavaScript Object**

Imagine we have this object in JavaScript:

var obj = { name: "John", age: 30, city: "New York" };

Use the JavaScript function JSON.stringify() to convert it into a string.

var myJSON = JSON.stringify(obj);

The result will be a string following the JSON notation.

myJSON is now a string, and ready to be sent to a server:

**Example**

var obj = { name: "John", age: 30, city: "New York" };  
var myJSON = JSON.stringify(obj);  
document.getElementById("demo").innerHTML = myJSON;

**Stringify a JavaScript Array**

It is also possible to stringify JavaScript arrays:

Imagine we have this array in JavaScript:

var arr = [ "John", "Peter", "Sally", "Jane" ];

Use the JavaScript function JSON.stringify() to convert it into a string.

var myJSON = JSON.stringify(arr);

The result will be a string following the JSON notation.

myJSON is now a string, and ready to be sent to a server:

**Example**

var arr = [ "John", "Peter", "Sally", "Jane" ];  
var myJSON = JSON.stringify(arr);  
document.getElementById("demo").innerHTML = myJSON;

**Exceptions**

**Stringify Dates**

In JSON, date objects are not allowed. The JSON.stringify() function will convert any dates into strings.

**Example**

var obj = { name: "John", today: new Date(), city : "New York" };  
var myJSON = JSON.stringify(obj);  
  
document.getElementById("demo").innerHTML = myJSON;

You can convert the string back into a date object at the receiver.

**Stringify Functions**

In JSON, functions are not allowed as object values.

The JSON.stringify() function will remove any functions from a JavaScript object, both the key and the value:

**Example**

var obj = { name: "John", age: function () {return 30;}, city: "New York"};  
var myJSON = JSON.stringify(obj);  
  
document.getElementById("demo").innerHTML = myJSON;

This can be omitted if you convert your functions into strings before running the JSON.stringify() function.

**Example**

var obj = { name: "John", age: function () {return 30;}, city: "New York" };  
obj.age = obj.age.toString();  
var myJSON = JSON.stringify(obj);  
  
document.getElementById("demo").innerHTML = myJSON;

If you send functions using JSON, the functions will lose their scope, and the receiver would have to use eval() to convert them back into functions.

JSON Objects

**Object Syntax**

**Example**

{ "name":"John", "age":30, "car":null }

JSON objects are surrounded by curly braces {}.

JSON objects are written in key/value pairs.

Keys must be strings, and values must be a valid JSON data type (string, number, object, array, boolean or null).

Keys and values are separated by a colon.

Each key/value pair is separated by a comma.

**Accessing Object Values**

You can access the object values by using dot (.) notation:

**Example**

myObj = { "name":"John", "age":30, "car":null };  
x = myObj.name;

You can also access the object values by using bracket ([]) notation:

**Example**

myObj = { "name":"John", "age":30, "car":null };  
x = myObj["name"];

**Looping an Object**

You can loop through object properties by using the for-in loop:

**Example**

myObj = { "name":"John", "age":30, "car":null };  
for (x in myObj) {  
  document.getElementById("demo").innerHTML += x;  
}

In a for-in loop, use the bracket notation to access the property values:

**Example**

myObj = { "name":"John", "age":30, "car":null };  
for (x in myObj) {  
  document.getElementById("demo").innerHTML += myObj[x];  
}

**Nested JSON Objects**

Values in a JSON object can be another JSON object.

**Example**

myObj = {  
  "name":"John",  
  "age":30,  
  "cars": {  
    "car1":"Ford",  
    "car2":"BMW",  
    "car3":"Fiat"  
  }  
 }

You can access nested JSON objects by using the dot notation or bracket notation:

**Example**

x = myObj.cars.car2;  
// or:  
x = myObj.cars["car2"];

**Modify Values**

You can use the dot notation to modify any value in a JSON object:

**Example**

myObj.cars.car2 = "Mercedes";

You can also use the bracket notation to modify a value in a JSON object:

**Example**

myObj.cars["car2"] = "Mercedes";

**Delete Object Properties**

Use the delete keyword to delete properties from a JSON object:

**Example**

delete myObj.cars.car2;

JSON Arrays

**Arrays as JSON Objects**

**Example**

[ "Ford", "BMW", "Fiat" ]

Arrays in JSON are almost the same as arrays in JavaScript.

In JSON, array values must be of type string, number, object, array, boolean or null.

In JavaScript, array values can be all of the above, plus any other valid JavaScript expression, including functions, dates, and undefined.

**Arrays in JSON Objects**

Arrays can be values of an object property:

**Example**

{  
"name":"John",  
"age":30,  
"cars":[ "Ford", "BMW", "Fiat" ]  
}

**Accessing Array Values**

You access the array values by using the index number:

**Example**

x = myObj.cars[0];

**Looping Through an Array**

You can access array values by using a **for-in** loop:

**Example**

for (i in myObj.cars) {  
  x += myObj.cars[i];  
}

Or you can use a for loop:

**Example**

for (i = 0; i < myObj.cars.length; i++) {  
  x += myObj.cars[i];  
}

**Nested Arrays in JSON Objects**

Values in an array can also be another array, or even another JSON object:

**Example**

myObj = {  
  "name":"John",  
  "age":30,  
  "cars": [  
    { "name":"Ford", "models":[ "Fiesta", "Focus", "Mustang" ] },  
    { "name":"BMW", "models":[ "320", "X3", "X5" ] },  
    { "name":"Fiat", "models":[ "500", "Panda" ] }  
  ]  
 }

To access arrays inside arrays, use a for-in loop for each array:

**Example**

for (i in myObj.cars) {  
  x += "<h1>" + myObj.cars[i].name + "</h1>";  
  for (j in myObj.cars[i].models) {  
    x += myObj.cars[i].models[j];  
  }  
}

**Modify Array Values**

Use the index number to modify an array:

**Example**

 myObj.cars[1] = "Mercedes";

**Delete Array Items**

Use the delete keyword to delete items from an array:

**Example**

delete myObj.cars[1];

JSON HTML

JSON can very easily be translated into JavaScript.

JavaScript can be used to make HTML in your web pages.

**HTML Table**

Make an HTML table with data received as JSON:

**Example**

obj = { table: "customers", limit: 20 };  
dbParam = JSON.stringify(obj);  
xmlhttp = new XMLHttpRequest();  
xmlhttp.onreadystatechange = function() {  
  if (this.readyState == 4 && this.status == 200) {  
    myObj = JSON.parse(this.responseText);  
    txt += "<table border='1'>"  
    for (x in myObj) {  
      txt += "<tr><td>" + myObj[x].name + "</td></tr>";  
    }  
    txt += "</table>"  
    document.getElementById("demo").innerHTML = txt;  
  }  
}  
xmlhttp.open("POST", "json\_demo\_db\_post.php", true);  
xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");  
xmlhttp.send("x=" + dbParam);

**Dynamic HTML Table**

**Make the HTML table based on the value of a drop down menu:**

**Example**

<select id="myselect" onchange="change\_myselect(this.value)">  
  <option value="">Choose an option:</option>  
  <option value="customers">Customers</option>  
  <option value="products">Products</option>  
  <option value="suppliers">Suppliers</option>  
</select>  
  
<script>  
function change\_myselect(sel) {  
  var obj, dbParam, xmlhttp, myObj, x, txt = "";  
  obj = { table: sel, limit: 20 };  
  dbParam = JSON.stringify(obj);  
  xmlhttp = new XMLHttpRequest();  
  xmlhttp.onreadystatechange = function() {  
    if (this.readyState == 4 && this.status == 200) {  
      myObj = JSON.parse(this.responseText);  
      txt += "<table border='1'>"  
      for (x in myObj) {  
        txt += "<tr><td>" + myObj[x].name + "</td></tr>";  
      }  
      txt += "</table>"  
      document.getElementById("demo").innerHTML = txt;  
    }  
  };  
  xmlhttp.open("POST", "json\_demo\_db\_post.php", true);  
  xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");  
  xmlhttp.send("x=" + dbParam);  
}  
</script>

**HTML Drop Down List**

Make an HTML drop down list with data received as JSON:

**Example**

obj = { table: "customers", limit: 20 };  
dbParam = JSON.stringify(obj);  
xmlhttp = new XMLHttpRequest();  
xmlhttp.onreadystatechange = function() {  
  if (this.readyState == 4 && this.status == 200) {  
    myObj = JSON.parse(this.responseText);  
    txt += "<select>"  
    for (x in myObj) {  
      txt += "<option>" + myObj[x].name;  
    }  
    txt += "</select>"  
    document.getElementById("demo").innerHTML = txt;  
  }  
}  
xmlhttp.open("POST", "json\_demo\_db\_post.php", true);  
xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");  
xmlhttp.send("x=" + dbParam);

**Read JSON Object in Java**

// Java program for write JSON to a file

import java.io.FileNotFoundException;

import java.io.PrintWriter;

import java.util.LinkedHashMap;

import java.util.Map;

import org.json.simple.JSONArray;

import org.json.simple.JSONObject;

public class JSONWriteExample

{

public static void main(String[] args) throws FileNotFoundException

{

// creating JSONObject

JSONObject jo = new JSONObject();

// putting data to JSONObject

jo.put("firstName", "John");

jo.put("lastName", "Smith");

jo.put("age", 25);

// for address data, first create LinkedHashMap

Map m = new LinkedHashMap(4);

m.put("streetAddress", "21 2nd Street");

m.put("city", "New York");

m.put("state", "NY");

m.put("postalCode", 10021);

// putting address to JSONObject

jo.put("address", m);

// for phone numbers, first create JSONArray

JSONArray ja = new JSONArray();

m = new LinkedHashMap(2);

m.put("type", "home");

m.put("number", "212 555-1234");

// adding map to list

ja.add(m);

m = new LinkedHashMap(2);

m.put("type", "fax");

m.put("number", "212 555-1234");

// adding map to list

ja.add(m);

// putting phoneNumbers to JSONObject

jo.put("phoneNumbers", ja);

// writing JSON to file:"JSONExample.json" in cwd

PrintWriter pw = new PrintWriter("JSONExample.json");

pw.write(jo.toJSONString());

pw.flush();

pw.close();

}

}

**Parsing JSON**

*Note*: Parsing means analyse or break down the code or convert the value of one data type to another data type.

java compiler parses source code of .java file and creates tokens that match the java grammar.

Parsing can be considered as a synonym of "*Breaking down into small pieces" and then analysing* what is there or using it in a modified way.

**JSON.parse()**

When receiving data from a web server, the data is always a string.

Parse the data with JSON.parse(), and the data becomes a JavaScript object.

Example

received text from a web server:

'{ "name":"John", "age":30, "city":"New York"}'

the JavaScript function JSON.parse() to convert text into a JavaScript object:

var obj = JSON.parse('{ "name":"John", "age":30, "city":"New York"}');

Use the JavaScript object in web page:

Example

<p id="demo"></p>  
  
<script>  
document.getElementById("demo").innerHTML = obj.name + ", " + obj.age;  
</script>

RestAssured

**Cookie:**

An HTTP **cookie** (also called web **cookie**, Internet **cookie**, browser **cookie**, or simply **cookie**) is a small piece of data sent from a website and stored on the user's computer by the user's web browser while the user is browsing.

Cookies are most commonly used to track website activity.

a web server can gather information about which web pages are used the most, and which pages are gathering the most repeat hits.

Servers can use cookies to provide personalized web pages.

Cookies are messages that web servers pass to your web browser when you visit Internet sites. Your browser stores each message in a small file, called cookie.txt. When you request another page from the server, your browser sends the cookie back to the server.

**Token :**

JSON Web Token is an Internet standard for creating data with optional signature and/or optional encryption whose payload holds JSON that asserts some number of claims. The tokens are signed either using a private secret or a public/private key.

**OAuth:** Is for **Authorisation b/w services.**

|  |  |  |
| --- | --- | --- |
|  | **OAuth1.0** | **OAuth2** |
|  | requires client to send two security tokens for each API call, and use both to generate the signature. | signatures are not required for the actual API calls once the token has been generated. |
| roles | consumer  user, service provider | client, authorization server, resource server, and resource owner |

***Note:***

**OAuth** is a standard that enables **access delegation**.

**OAuth** is not itself a technology that does authentication.

**OAuth** allows a user to **delegate** some level of **access** to his or her data to a third-party entity without handing over complete credentials.

**OAuth Access tokens:**

User allowed permissions.

Trustable.

**OAuth flows:**

1. **Authorization Control** flow:

Auth Token + Access Token

1. **Implicit** flow

Access Token only (short lived)

1. **Client credential** flow

B/w **Microservices (**Big use case**), for**  confidential clients (well trusted).

**Terminologies:**

1. **Resource: Protected resource**

Which is being sought.

1. **Resource owner:**

User having access resource.

1. **Resource Server:**

Where resource is placed.

Has the burden of **Security.**

1. **Client:**

Application making request to the protected resource.

1. **Authorisation Server:**

*Which generates Auth and Access tokens for client.*

*It is coupled with Resource Server.*

**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 allows you to* ***write the tests*** *using* ***Gherkin language*** *(****Given-When-Then*** *syntax)*

*Rest assured is* ***implemented*** *in* ***Groovy.***

Official site of REST Assured: **http://rest-assured.io/**

**Static imports**

In order to use REST assured effectively it's recommended to statically import methods from the following classes:

**io.restassured.RestAssured.\***

**io.restassured.matcher.RestAssuredMatchers.\***

**org.hamcrest.Matchers.\***

1. **Basic Type Test using REST Assured.**
2. ***POST Request using Rest Assured***

Steps:

*1.* ***Create a Request*** *pointing to the* ***Service Endpoint***

*2. Create a* ***JSON request*** *which contains* ***all the fields***

*3.* ***Add JSON header and body*** *in the request.*

4. **Post** the request and validate the response

1. ***PUT Request using Rest Assured***
2. *Create a* ***variable empid*** *which we intend to update with our PUT request.*

***Note***: The ***variable empid***has been previously created as a resource on the server, and we will update the associated information in the PUT request.

1. *Create a Request pointing to the Service Endpoint*
2. *Create a JSON request which contains all the fields which we wish to update.*
3. *Send JSON content in the body of Request and pass PUT Request*
4. *Validate the PUT Request response received*
5. ***DELETE Request using Rest Assured***
6. *Create a variable empid and specify the value to be deleted.*
7. *Specify a Request pointing to the Service Endpoint.*
8. *Send the Delete Request.*
9. *Validate the Delete Request response received.*
10. ***GET Request Using RestAssured***

1. Using Rest-Assured class, RequestSpecification is generated for the URL:

2. HTTP method is specified.

3. Request is send to server.

4. Response is received from server.

5. validate the response .

***Note:***

**RequestSpecification** is an **interface** and in a Rest-Assured library,

**all** **requests** are **represented** by this interface.

**2. Gherkin Style Test using REST Assured:**

**Performance Testing of REST APIs using JMeter:**

*Precondition for JMeter*: **Java**

**Apache JMeter,** as a **standalone** application with a graphical user interface **(GUI),** allows you to **create tests from the GUI.**

This can be handy especially because of the entities hierarchy introduced from JMeter—everything you create inside your test will be shown in a **tree-like representation** in JMeter GUI.

**Steps to perform Load Test:**

1. Test Plan 🡪 Add a Thread group

**(**Number of Thread = Number of concurrent users**)**

1. In Thread group add a Sampler 🡪 HTTP request.

Add the end point in “Server name” or in “Path”

1. Thread group 🡪 Listener 🡪 View Results Tree.
2. Select the HTTP method.
3. Set the thread / users count.
4. Save the file and file will be saved as .jmx type.
5. Run the Test.

**Note:**

**We can run multiple requests together of different HTTP methods.**

**HTTP Request defaults:** *to set the* ***Path*** *as* ***default*** *for all* ***the end points.***

***To validate add Assertions:***

Request 🡪 Add 🡪 Assertions 🡪 **“Response Assertions” / “JSON Assertions”**