**Rest Assured**

RestAssured is an API/Library through which we can automate RestAPI.

***Pre-requisites:***

1. Java 9+ & Eclipse
2. TestNG
3. Maven

***Dependencies:***

**rest-assured**

<dependency>

<groupId>io.rest-assured</groupId>

<artifactId>rest-assured</artifactId>

<version>5.5.5</version>

<scope>test</scope>

</dependency>

**json-path**

<dependency>

<groupId>io.rest-assured</groupId>

<artifactId>json-path</artifactId>

<version>5.5.5</version>

<scope>test</scope>

</dependency>

**Json**

<!-- https://mvnrepository.com/artifact/org.json/json -->

<dependency>

<groupId>org.json</groupId>

<artifactId>json</artifactId>

<version>20250107</version>

</dependency>

**scribejava-apis**

<!--https://mvnrepository.com/artifact/com.github.scribejava/scribejava-apis -->

<dependency>

<groupId>com.github.scribejava</groupId>

<artifactId>scribejava-apis</artifactId>

<version>8.3.1</version>

<scope>test</scope>

</dependency>

**json-schema-validator**

<!--https://mvnrepository.com/artifact/com.github.java-json-tools/json schemavalidator -->

<dependency>

<groupId>com.github.java-json-tools</groupId>

<artifactId>json-schema-validator</artifactId>

<version>2.2.14</version>

<scope>test</scope>

</dependency>

**xml-schema-validator**

**Gson**

**Testng**

<!-- https://mvnrepository.com/artifact/org.testng/testng -->

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>7.8.0</version>

<scope>test</scope>

</dependency>

**http request**

*get*

*post*

*put*

*delete*

***gerkin – Keywords(Methods)***

*Static Package: ---*

**import** **static** io.restassured.RestAssured.\*;

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

**import** **static** org.hamcrest.Matchers.\*;

==========================================================

***How many ways we create request body: ----***

1. HashMap
2. Using org.json library
3. Using POJO (Plane Old Java Object)
4. Using External JSON file.

**Use Express.js to serve XML (closest to json-server idea)**

Create a simple Node.js server:

1. **Init project**

mkdir xml-server

cd xml-server

npm init -y

npm install express

npm install cookie-parser

1. server.js
2. const express = require("express");
3. const fs = require("fs");
4. const cookieParser = require("cookie-parser");
5. const app = express();
6. const port = 3000;
7. // middleware to parse cookies
8. app.use(cookieParser());
9. // Serve XML and set a cookie
10. app.get("/store", (req, res) => {
11. res.set("Content-Type", "application/xml");
12. // set a cookie (valid for 1 minute)
13. res.cookie("user", "TestUser", {
14. httpOnly: true,
15. maxAge: 60000
16. });
17. // read cookies from request
18. console.log("Cookies received from client:", JSON.stringify(req.cookies));
19. const user = req.cookies.user || "Guest";
20. // Load XML file
21. const xmlData = fs.readFileSync("store.xml", "utf8");
22. // send XML with a comment showing cookie info
23. res.send(`<!-- Hello ${user} -->\n${xmlData}`);
24. });
25. // clear cookies
26. app.get("/logout", (req, res) => {
27. res.clearCookie("user");
28. res.send("Cookie cleared!");
29. });
30. app.listen(port, () => {
31. console.log(`XML server running at http://localhost:${port}/store`);
32. });
33. **store.xml**

<store>

<book>

<title>Sayings of the Century</title>

<author>Nigel Rees</author>

<price>8.95</price>

</book>

<book>

<title>Sword of Honour</title>

<author>Evelyn Waugh</author>

<price>12.99</price>

</book>

<book>

<title>1984</title>

<author>George Orwell</author>

<price>6.99</price>

</book>

</store>

**//json ---🡪 jsonschema converter**

**//xml to xsd converter**

Serialization (POJO ----> JSON)

De-serialization (JSON ---> pojo)

Body(json) -------🡪 Request-----------🡪Response(json)

**Authorizations:-**

**Authentication – valid or not**

**Authorization – access**

**Authentication supported: ---**

Basic

Digest

Preempive

Bearer token

oauth 1.0, 2.0

API Key

**1. When headers are required**

Headers are needed if the API expects certain information to process the request. Common cases:

* **Content-Type**: Tells the server the format of the body data.
  + Example: Content-Type: application/json when sending JSON.
* **Authorization**: Needed if the API is protected.
  + Example: Authorization: Bearer <token> or Digest/Basic auth.
* **Custom headers**: Some APIs require custom headers for versioning or tracking.
  + Example: X-API-KEY: <your-api-key>

If these headers are missing, the API may return:

* 401 Unauthorized (authentication missing)
* 415 Unsupported Media Type (wrong or missing content type)
* 400 Bad Request (missing required headers)

**2. When headers are optional**

* If the API accepts default formats and doesn’t require authentication, you can POST without headers.
* Example: A public test API that accepts form-data or JSON may not require any headers.

**3. Summary**

| **Header** | **Mandatory?** | **Why** |
| --- | --- | --- |
| Content-Type | Often | To tell the server body type |
| Authorization | Only if API is protected | To validate user |
| Custom headers | Only if API docs specify | For API-specific logic |

**1️⃣ Basic vs Digest Authentication**

**Basic Authentication**:

* Sends the **username and password encoded in Base64** with every request.
* It's **stateless** and simple.
* Because the credentials are sent in the **first request**, you can use **preemptive authentication**. This avoids the extra round-trip to get a 401 Unauthorized.

**Digest Authentication**:

* Uses a **challenge-response mechanism**.
* The server sends a **nonce (a unique token)** in a 401 Unauthorized response.
* The client combines the username, password, nonce, request method, and URI into a **hashed response**.
* The server then verifies this hash.

**2️⃣ Why preemptive only works for Basic**

* **Preemptive authentication** means: “I’ll send my credentials immediately, without waiting for a 401 challenge.”
* Basic auth can do this safely because credentials are always the same and predictable.
* Digest auth **cannot be preemptive** because:
  1. It requires a **nonce from the server** for each session.
  2. The hash depends on this nonce.
  3. You **cannot compute the digest before the server provides the nonce**.

So, a method like:

.auth().preemptive().digest("adminDigest", "passwordDigest")

**doesn’t make sense**, because the client doesn’t yet know the nonce.

**3️⃣ Illustration**

1. **Basic preemptive**

GET /api/data HTTP/1.1

Authorization: Basic YWRtaW5CYXM6cGFzc3dvcmRCYXMA

* Sent directly, no 401 needed.

1. **Digest**

Client: GET /api/data

Server: 401 Unauthorized

WWW-Authenticate: Digest realm="Digest Area", nonce="abc123", qop="auth"

Client: GET /api/data

Authorization: Digest username="adminDigest", realm="Digest Area",

nonce="abc123", uri="/api/data", response="..."

* Cannot skip the first 401 because the nonce is unknown.

✅ **In short:**  
Preemptive authentication only works for **Basic**, because Digest needs server-supplied data (nonce) to calculate its response. Sending Digest credentials “early” is impossible.