API Fundamentals: A Tester's Guide

Whenever we talk about APIs, we often refer to "API web services."
So, what is a Web Service? And what is an API?

- A **Web Service** is a service that is available over the internet.
- An **API** (Application Programming Interface) acts as an intermediary between two different applications, allowing them to communicate.

Example of API: Bus Ticketing Applications

Let's take an example with a **bus ticketing application**.
Bus tickets can be booked through either the Tamil Nadu Transport official website or the RedBus website.

These applications may access the same database, but they operate independently. They might even be built using different programming languages. For example:

- The Tamil Nadu Transport website could be created using Java.
- The RedBus website could be created using Python.

However, both applications need to communicate. If a ticket is sold out for a route from Theni to Chennai on the RedBus website, it should be updated on the Tamil Nadu Transport website. Similarly, if a ticket is sold out or a seat is modified on the Tamil Nadu Transport website, it should reflect on RedBus in real-time.

So, how do they communicate, even though they are built using different programming languages?
The answer is **JSON/XML**, which is used for data exchange. In most **REST web services**, **JSON** is used.

JSON acts like a "translator" between the two websites, ensuring the ticket details are transferred correctly and keeping both platforms updated.

Some Key Terms:

- **REST Web Service**
  A REST (Representational State Transfer) web service is a type of web service that follows the principles of REST architecture. It allows applications to communicate using standard HTTP methods (GET, POST, PUT, DELETE), usually with JSON or XML as the data format.
- **JSON (JavaScript Object Notation)**
  JSON is a lightweight data-interchange format. It is easy for humans to read and write, and easy for machines to parse and generate. JSON is commonly used in REST web services to transfer data between servers and clients.

API Important Terminologies

1. **Client** – This is anyone or anything that makes a request. It could be a person using a website or a piece of software asking for information.
2. **Server** – A program that gets the client's request, processes it, and sends back a response.
3. **URL (Uniform Resource Locator)** – This is the web address you type in a browser to find something on the internet (like www.example.com).
4. **Request** – A message from the client to the server asking for some data or action.
5. **Response** – The message the server sends back to the client after receiving the request. It could include the data you asked for or an error message.
6. **Request Body** – This is the information the client sends to the server (like when you fill out a form and submit it).
7. **Response Body** – The information the server sends back to the client after it processes the request (like showing the results of a search).
8. **Headers** – Extra information that comes with both the request and response, like the type of data being sent (e.g., JSON or HTML). It's like the envelope of a letter that contains important details about the message inside.
9. **Content Type** – This tells the server what kind of data the client is sending or receiving. Some examples:

- **application/json** – Means the data is in JSON format.
- **application/xml** – Means the data is in XML format.
- **text/html** – Means the data is in HTML format (like a web page).

HTTP Request **(Hypertext Transfer Protocol)**

When a client (like your browser) sends a request over the internet, it uses HTTP. Think of HTTP as a language that lets the website ask for something, and the server responds with what's needed.

HTTP Methods

The web works with the help of HTTP requests and responses. The backbone of these requests is **HTTP Methods**. These methods define the action being requested. Let's go through the important ones:

GET

1. Used to retrieve information (like a webpage or data) without changing anything on the server. That's why it's called a "safe method."
2. If the resource exists on the server, you'll get a response along with a status code and the resource data (response body).
3. If the resource isn't found, you'll get a relevant status code like `404`.
4. **Example:**
5. `GET http://www.linkedin.com/users`

POST

1. Used to create a new resource on the server.
2. It modifies the server, so it's not a "safe method." For example, sending two identical POST requests creates two different resources with the same content but different IDs.
3. If the resource is successfully created, the server returns a status code like `201`.
4. POST requests include a **request body** containing the data to create the resource.
5. **Example:**

```
POST http://www.linkedin.com/users
Body: { "name": "Yogesh Pandian", "email": "yogeshpandian@mail.com" }
```

PUT

1. Used to update an existing resource on the server.
2. If the resource doesn't already exist, the server creates it.
3. Returns a status code indicating success or failure.
4. You must send the **updated request body** with the PUT request.
5. **Example:**

```
PUT http://www.linkedin.com/users/123
Body: { "name": " Yogesh Pandian ", "email":
"yogeshchandran@mail.com" }
```

DELETE

1. Used to delete a resource from the server.
2. If the resource exists, it deletes it and sends a relevant status code (like `200` for success).
3. **Example:**

```
DELETE http://www.linkedin.com/users/123
```

HTTP Response

When the client submits a request, the server responds with:

1. **Resource data** (e.g., JSON, XML, or plain text).
2. **Status code** indicating whether the request was successful or not.
3. Let's break down some important status codes:

Status Codes:

- **1xx (Informational)** – The server is still processing the request.
- **2xx (Success)** – The request was successful:
  - **200**: The GET request was successful, and the resource was found.
  - **201**: A POST request successfully created a resource.
- **3xx (Redirection)** – The client is redirected to another URL.

- **4xx (Client Error)** – There's an issue with the client's request:
  - **400**: Bad request (e.g., wrong data sent).
  - **401**: Unauthorized (e.g., you're not logged in).
  - **403**: Forbidden (e.g., employees can view only their details, but admins can see all).
  - **404**: Not found (e.g., broken or invalid link).
- **5xx (Server Error)** – Something is wrong on the server side:
  - **500**: Server error (e.g., the server is down). Only the server maintenance team can fix this.

---

1. Endpoint

Where the API sends a request to get or send data. Think of it as the URL address for a specific feature.

Example (LinkedIn):

- `https://api.linkedin.com/v2/users`
  This endpoint is used to get LinkedIn user data.

---

2. Path Parameter

A part of the URL used to identify specific data.

Example (LinkedIn):

- `https://api.linkedin.com/v2/users/{userId}`
  Replace `{userId}` with the actual user ID like `12345`:
  `https://api.linkedin.com/v2/users/12345`
  This gets the profile of user ID 12345.

---

3. Query Parameter

Extra info added to the URL to filter results. It comes after a `?`.

Example (LinkedIn):

- `https://api.linkedin.com/v2/users?location=India&jobTitle=Engineer`
  This searches for LinkedIn users in **India** with the job title **Engineer**.

---

4. Header

Additional info sent with a request, like login details or the type of data you want.

Example (LinkedIn):

- A header might look like this:
  `Authorization: Bearer <your_access_token>`
  It tells LinkedIn you're logged in and allowed to access the data.

---

5. Authentication

Confirms who you are so the server knows you're allowed to use the API. Types:

1. **No Auth**: No login required (rare).
2. **Basic Auth**: Username and password sent in the header.
   Example:
   - `Authorization: Basic <encoded_username_password>`
3. **API Token**: A key (like a password) to identify you.
   Example:
   - Header: `Authorization: Token <your_api_key>`
4. **Bearer Token**: A type of API token that's more secure.
   Example (LinkedIn):
   - `Authorization: Bearer <your_access_token>`
5. **OAuth**: Advanced method where you log in through LinkedIn to get access.
   Example:
   - When you click “Sign in with LinkedIn” on other apps, that's OAuth.

Understanding API Testing

Now that we know what APIs and web services are, and how HTTP works, let’s connect this to **API Testing**.

What is API Testing?

API testing ensures the APIs work as expected. We test the requests and responses between the client and server. In simple terms:

1. We check if the API returns the correct data.
2. We verify that the API handles errors gracefully (e.g., what happens if the resource doesn’t exist?).
3. We validate the status codes, response times, and data formats (like JSON).

Manually test API:

Test Case: Verify if the Google homepage is successfully loaded.

1. **Manual Testing (via Browser)**:
    - Open your browser and go to `http://google.co.in`.
    - Right-click anywhere on the page and select **Inspect**.
    - Navigate to the **Network** tab.
    - Reload the page (press **F5**) to capture the requests.
    - Check the status code for the request to `http://google.co.in`.

      Expected Output:

        - Status Code: `200` (OK)
        - Response: HTML content for the Google homepage.
2. **Perform the Same with Postman**:

Using Postman for API Testing

1. GET Request

Purpose: To retrieve the Google homepage.

Prerequisites:

- **Header**: `Content-Type: application/json`
- **Request Method**: `GET`
- **Endpoint**: `http://google.co.in`

Steps:

1. Open Postman and select `GET`.
2. Enter the URL: `http://google.co.in`.
3. Add a header:
    - Key: `Content-Type`
    - Value: `application/json`.
4. Click **Send**.

Expected Output:

Response Body: HTML code of the Google homepage.
Example:

```
<!doctype html>
<html>
  <head>...</head>
  <body>
```

```
    <h1>Google</h1>
  </body>
</html>
```

Status Code: `200` (OK).

---

2. POST Request

Purpose: To create a new resource on the server.

Dummy Example:
Let's say we are creating a new user on a dummy API.

Prerequisites:

- **Header**: `Content-Type: application/json`
- **Request Method**: `POST`
- **Endpoint**: `http://dummyapi/users`
- **Request Body**:

```
{
  "name": "Yogesh Pandian",
  "email": "yogeshchandran@mail.com"
}
```

Steps:

1. Open Postman and select `POST`.
2. Enter the URL: `http://dummyapi/users`.
3. Add a header:
    - Key: `Content-Type`
    - Value: `application/json`.
4. Add the **Request Body**.
5. Click **Send**.

Expected Output:

Response Body:

```
{
  "id": 123,//The unique identifier for the newly created user
(generated by the server)
  "name": "Yogesh Pandian",
  "email": "yogeshchandran@mail.com",
  "message": "User created successfully"
}
```

Status Code: `201` (Created).

---

3. PUT Request

Purpose: To update an existing resource.

Dummy Example:
Let's say we are updating a user's email address.

Prerequisites:

- **Header**: `Content-Type: application/json`
- **Request Method**: `PUT`
- **Endpoint**: `http://dummyapi/users/123--> This 123 is unique id`
- **Request Body**:

```
{
  "email": "yogeshchandran@mail.com"
}
```

Steps:

1. Open Postman and select `PUT`.
2. Enter the URL: `http://dummyapi/users/123`.
3. Add a header:
    - Key: `Content-Type`
    - Value: `application/json`.
4. Add the **Request Body**.
5. Click **Send**.

Expected Output:

Response Body:

```
{
  "id": 123,
  "name": "Yogesh Pandian",
  "email": "yogeshchandran@mail.com",
  "message": "User updated successfully"
}
```

Status Code: `200` (OK).

---

4. DELETE Request

Purpose: To delete a resource from the server.

Dummy Example:
Let's say we are deleting a user.

Prerequisites:

- **Header**: `Content-Type: application/json`
- **Request Method**: `DELETE`
- **Endpoint**: `http://dummyapi/users/123`

Steps:

1. Open Postman and select `DELETE`.
2. Enter the URL: `http://dummyapi/users/123`.
3. Add a header:
    - Key: `Content-Type`
    - Value: `application/json`.
4. Click **Send**.

Expected Output:

Response Body:

```
{
  "message": "User deleted successfully"
}
```

Status Code: `200` (OK).

Why is this considered manual API testing?

When using Postman, you're manually:

1. Selecting the request method (GET, POST, PUT, DELETE).
2. Entering headers, request body, and endpoint.
3. Clicking "Send" to view the response.

Since no scripts or automation tools (like Selenium or REST Assured) are used to repeat these tests, it's **manual testing**.

Automating API Tests:

Now that we understand API testing, the next step is to write scripts to automate and repeatedly execute the tests.

Common Misconception:

There is a common misconception that Selenium is the solution for all types of automation. However, **Selenium** is specifically designed for automating **web applications** (UI/browser automation), not API testing.

So, how do we automate API testing?

We can automate API tests using **native Java** or **third-party libraries**. Here are some options:

1. **HttpURLConnection (java.net)**: This is a built-in class in Java that allows you to send HTTP requests and handle responses. It's simple but not as feature-rich as other libraries.
2. **UniRest**: A third-party library that simplifies sending HTTP requests and handling responses. It provides a more concise API than HttpURLConnection.
3. **RestAssured**: A popular and widely used library for automating RESTful API tests. It has many built-in features for validating responses, setting headers, and making requests in a more readable way.

HttpUrlConnection

```
public static void main(String[] args) throws IOException {
    URL url = new URL( spec: "http://google.co.in"); // Converting string to URL
    // Open connection to the URL and cast it to HttpURLConnection for HTTP specific methods
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    // Establish connection
    connection.connect();
    // Get the HTTP response code (e.g., 200 for success, 404 for not found)
    int statusCode = connection.getResponseCode();
    System.out.println("Status Code: " + statusCode); // Print status code
    // Get the response message associated with the status code (e.g., "OK" for 200)
    String responseMessage = connection.getResponseMessage();
    System.out.println("Response Message: " + responseMessage); // Print response message
    // Read the response body (in this case, it's the HTML of the page)
    InputStream inputStream = connection.getInputStream();
    // Create an InputStreamReader to read from the InputStream
    InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
    // Use BufferedReader to read the input stream line by line
    BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
    // Variable to hold each line of the response body
    String responseBodyLine;
    // StringBuffer to hold the entire response body
    StringBuffer responseBody = new StringBuffer();
    // Read all lines of the response body
    while ((responseBodyLine = bufferedReader.readLine()) != null) {
        responseBody.append(responseBodyLine); // Append each line to the StringBuffer
    }
    // Close the BufferedReader
    bufferedReader.close();
    // Print the complete response body
    System.out.println("Response Body: " + responseBody);
```

UniRest:

Add below dependency to automate API using UniRest Library

```
<dependency>
    <groupId>com.konghq</groupId>
    <artifactId>unirest-java</artifactId>
    <version>3.11.11</version>
</dependency>
```

```
public class UniRestGetExample {
    public static void main(String[] args) {
        // Sending GET request to the URL and expecting the response in JSON format
        HttpResponse<JsonNode> response = Unirest.get("http://google.co.in")
                .header( s: "accept",  s1: "application/json") // Expecting JSON response
                .asJson(); // asJson() makes the response JSON-formatted

        // Extracting status code from the response
        int statusCode = response.getStatus();
        System.out.println("Status Code: " + statusCode);

        // Extracting status message (text) from the response
        String statusMessage = response.getStatusText();
        System.out.println("Response Message: " + statusMessage);

        // Checking if the response body is not null and is a valid JSON response
        if (response.getBody() != null) {
            // Extracting and printing response body as JSON
            JsonNode responseBody = response.getBody();
            System.out.println("Response Body (JSON): " + responseBody.toString());
        } else {
            // If the response body is null, it likely contains HTML. Handling it as a string.
            HttpResponse<String> stringResponse = Unirest.get("http://google.co.in").asString();
            System.out.println("Response Body (HTML): " + stringResponse.getBody());
        }
    }
}
```

Rest Assured:

Add below dependency to automate API using RestAssured Library

```
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>5.2.0</version>
</dependency>
```

```
public class RestAssuredGetExample {
    public static void main(String[] args) {
        // Send a GET request to the "/images" endpoint.
        // The 'given()' method is used to configure the request (like adding headers, query params, etc.).
        // The 'when()' method specifies the HTTP method (in this case, GET).
        // The 'get()' method specifies the relative endpoint we are hitting.
        // The 'then()' method allows us to validate the response or extract the response.
        Response response = given() RequestSpecification
                .baseUri("http://google.co.in")
                .when()
                .get( path: "/images")  // We're hitting the /images endpoint of google.co.in
                .then() ValidatableResponse
                .log().all() // This will log the entire response details (status, headers, body) for debugging
                .extract().response(); // Extract the response object for further processing

        // Print the HTTP status code of the response.
        // This tells us whether the request was successful (e.g., 200 for success)
        System.out.println("Status Code: " + response.getStatusCode());

        // Print the full status line of the response.
        // This includes the HTTP version and the response status (e.g., "HTTP/1.1 200 OK")
        System.out.println("Status Message: " + response.statusLine());

        // Print the response body as a string.
        // Since the response is HTML and not JSON, we use asString() to convert the HTML content to a string.
        System.out.println("Response Body: " + response.getBody().asString());
    }
}
```