Rest Assured API Testing – Complete Notes for Beginners to Advanced

What is Rest Assured? (Simple Explanation)

What it is:
Rest Assured is a Java library that helps us test REST APIs easily.
It allows us to send requests (GET, POST, PUT, DELETE) and check the responses.

Why we use it:
- To automate API testing without writing too much code.
- Makes API testing simple because it has in-built methods.
- Works well with testing frameworks like TestNG and JUnit.

How to implement:
1. Add Rest Assured dependency in your project (Maven/Gradle).
2. Write a test method.
3. Use Rest Assured methods like given(), when(), then() to send request and check response.

Example (easy to remember):
Think of Rest Assured like a "food delivery app":
- given() → you set details like address, food item (request setup).
- when() → you place the order (send the request).
- then() → you check if food arrived correctly (validate response).

Simple Code:

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

@Test
public void testGetUsers() {
   given()
      .when().get("https://reqres.in/api/users?page=2")
      .then().statusCode(200);
}
```

Memory Trick:
Remember 3 words → "given → when → then".
(It’s like: If you GIVE order, WHEN delivery happens, THEN you check food).

Method Syntax:

Method	Syntax	Parent
given()	public static RequestSpecification given()	RestAssured (class)
when()	public ResponseSpecification when()	RequestSpecification (IF)
then()	public ValidatableResponse then()	Response (class)
get()	public Response get(String path)	RequestSpecification (IF)
post()	public Response post(String path)	RequestSpecification (IF)
put()	public Response put(String path)	RequestSpecification (IF)
delete()	public Response delete(String path)	RequestSpecification (IF)
baseUri()	public RequestSpecification baseUri (String uri)	RequestSpecification (IF)

header()	public RequestSpecification header (String name, Object value)	RequestSpecification (IF)
body()	public RequestSpecification body(Object body)	RequestSpecification (IF)
log()	public RequestSenderOptions log()	RequestSpecification (IF)
all()	public RequestSpecification all()	RequestSenderOptions (cls)

Request Specification

1.What is Request Specification?

- It is a reusable setup for sending API requests in Rest Assured.
- You can define things like:
  - Base URL
  - Headers
  - Query parameters
  - Authentication
  - Content Type (like JSON)

2.Why do we use Request Specification?

- To avoid writing the same code again and again.
- Helps keep the test code clean and simple.
- Makes it easy to update common settings in one place.

3.Example Without Request Specification:

```
given()
    .baseUri("http://localhost:3000")
    .header("Content-Type", "application/json")
    .body("{ \"name\": \"John\" }")
.when()
    .post("/people")
.then()
    .statusCode(201);
```

4.Example With Request Specification:

```
RequestSpecification reqSpec = new RequestSpecBuilder()
    .setBaseUri("http://localhost:3000")
    .setContentType(ContentType.JSON)
    .build();

given()
    .spec(reqSpec)
    .body("{ \"name\": \"John\" }")
.when()
    .post("/people")
.then()
    .statusCode(201);
```

5.When to Use It:

- When your API tests have the same setup in many places.
- When you want to make your code short and easy to manage.

6.Summary:

- Request Specification is a template for your request setup.
- Saves time and reduces repetition.
- Useful for setting base URI, headers, auth, and content type.

Response Specification

1.What is Response Specification?

- A reusable way to check the response from an API.
- Helps you avoid writing the same validations again and again.
- Makes the test code shorter and cleaner.

2.Why Use Response Specification?

- To set expected response values in one place.
- To reuse the same response checks in multiple tests.
- To make your tests easy to read and manage.

3.What Can You Validate?

- Status code (example: 200, 201, 404)
- Content type (example: JSON, XML)
- Response body (optional)
- Logging (like printing the full response)

4.Example Code:

Create response specification:

```
ResponseSpecification resSpec = new ResponseSpecBuilder()
    .expectStatusCode(200)
    .expectContentType(ContentType.JSON)
    .build();
```

Use it in your test:

```
given()
    .spec(requestSpecification)
    .body("{...}")
.when()
    .post("/people")
.then()
    .spec(resSpec)
    .log().all();
```

5.Benefits:

- Saves time by avoiding repeated code
- Keeps all response checks in one place
- Makes the test more readable and organized

6.Summary:

- Response Specification is used to validate responses.
- You define it once and use it in many tests.
- Useful for status code, content type, and logging.

ValidatableResponse

1.What is ValidatableResponse?

- It is an interface in Rest Assured that allows you to validate API responses easily.

- Once you get a response from the server, you use ValidatableResponse to check:
  - Status codes
  - Headers
  - Response body
  - Response time

Think of it like a toolbox for checking the server’s reply.

2.Why do we use ValidatableResponse?

- To confirm that the API is working as expected.
- To apply multiple checks (status, headers, body) in a readable way.
- To make tests look clean and structured.

3.How to Get ValidatableResponse?

- First, send a request and capture the response.
- Then, call .then() on the response.
- .then() converts Response → ValidatableResponse.

4.Example Without Explanation:

```
Response response =
RestAssured.get("https://restful-booker.herokuapp.com/booking/5");

// Convert Response to ValidatableResponse
ValidatableResponse validatableResponse = response.then();

// Perform validations
validatableResponse.statusCode(200);
validatableResponse.contentType("application/json");
```

5.What Can You Validate?

- Status Code → statusCode(200)
- Content Type → contentType(ContentType.JSON)
- Response Body → body("firstname", equalTo("John"))
- Headers → header("Server", "Cowboy")
- Response Time → time(lessThan(2000L))

6.When to Use It?

- Whenever you want to check the response after hitting an API.
- Especially in test cases where validations matter (status code, body, headers).

7.Summary:

- ValidatableResponse = used for validations.
- Response → .then() → ValidatableResponse.
- Can validate status, headers, body, and response time.
- Keeps test assertions simple and readable.

QUICK COMPARISON TABLE
==========

RequestSpecification
- Purpose: Configure the request before sending.
- Created From / How You Get It: RestAssured.given().
- Analogy: Packing your delivery bag.

ResponseSpecification
- Purpose: Predefine response expectations.
- Created From / How You Get It: new ResponseSpecBuilder().build().

- Analogy: Your critic's checklist.

ValidatableResponse
- Purpose: Validate response after it’s received.
- Created From / How You Get It: .then() on a Response.
- Analogy: Actually tasting and verifying.

Static Imports in Java
======================

What it is:

- Static import is a feature in Java that allows you to use
  static members (methods or variables) of a class directly
  without writing the class name every time.
- Introduced in Java 5.

Why we use it:

- To make code short and more readable.
- Example: Instead of writing Math.sqrt(25),
  you can write just sqrt(25) after static import.

How to implement:
----
1. Write "import static" keyword before the class and its static member.
2. You can import:
   - A specific static member → import static java.lang.Math.sqrt;
   - All static members of a class → import static java.lang.Math.*;

Example:

Without static import:
```
    public class TestStaticImport {
        public static void main(String[] args) {
            System.out.println(Math.sqrt(25));   // Using class name Math
            System.out.println(Math.PI);         // Using class name Math
        }
    }
```

With static import:
```
    import static java.lang.Math.*;
    public class TestStaticImport {
        public static void main(String[] args) {
            System.out.println(sqrt(25));   // No need to write Math
            System.out.println(PI);         // Directly using PI
        }
    }
```

Easy Memory Trick:

Think of static import like a nickname:
- Normally you call someone with their full name → Math.sqrt
- With static import, you call them with just their short name → sqrt

Just remember: "Remove the class name, keep it short".

Rest Assured
============

What it is:

- Rest Assured is a Java library used for testing REST APIs.

- It allows us to send HTTP requests (GET, POST, PUT, DELETE, etc.)
  and validate responses easily.

Why we use it:

- To test APIs without writing complex code.
- It supports BDD style (Given - When - Then) which is simple and readable.
- Saves time for API automation compared to writing raw HTTP client code.
- Widely used in API Testing frameworks with TestNG, JUnit, or Cucumber.

How to implement:
----
1. Add Rest Assured dependency (Maven/Gradle).
2. Import Rest Assured static methods.
3. Write test cases using Given - When - Then syntax.

Example (GET request):

```
    import static io.restassured.RestAssured.*;
    import static org.hamcrest.Matchers.*;

    public class ApiTest {
        @Test
        public void testGetUsers() {
            given()
                .when()
                    .get("https://reqres.in/api/users?page=2")
                .then()
                    .statusCode(200)
                    .body("data[0].first_name", equalTo("Michael"));
        }
    }
```

Here:
- given() → setup request
- when() → perform action (GET, POST, etc.)
- then() → validate response

Easy Memory Trick:

Think of Rest Assured as "Ordering food online":
- given() → Choose restaurant and menu (setup)
- when() → Place the order (action)
- then() → Check the order delivered correctly (validation)

So: "Given - When - Then" = "Setup - Action - Check".

GET Request in Rest Assured
===========================

What it is:

- A GET request is used to fetch (read) data from the server.
- In Rest Assured, GET request means calling an API endpoint to retrieve
information.

Why we use it:

- To check if the API is giving correct data.
- To validate status code and response body.
- Most common request in API testing (used for reading data).

How to implement:

```
-----------------
1. Import Rest Assured library in your project.
2. Build the request using given().
3. Call the API using .get().
4. Validate the response using .then().

Example 1: Using RequestSpecification
-------------------------------------
    public void get_Request_With_Request_Specification() {

        // Build Request
        RequestSpecification requestSpecification = RestAssured.given();
        requestSpecification = requestSpecification.log().all();
        requestSpecification.baseUri("https://restful-booker.herokuapp.com/");
        requestSpecification.basePath("booking/{id}");
        requestSpecification.pathParam("id", 5);

        // Hit Request
        Response response = requestSpecification.get();

        // Validate Response
        ValidatableResponse validatableResponse =  response.then();
        validatableResponse.log().all();
        validatableResponse.statusCode(200);
    }

Example 2: Using Method Chaining
--------------------------------
    @Test
    public void get_Request_With_Method_Chaining() {

        RestAssured.given()
        .log().all()
        .baseUri("https://restful-booker.herokuapp.com")
        .basePath("/booking/{id}")
        .pathParam("id", 5)
        .get()
        .then()
        .log().all()
        .statusCode(200);
    }

Status Codes (for GET):
-----------------------
✅ Pass:
- 200 → OK (Request successful, data returned)

❌ Fail:
- 404 → Not Found (wrong URL or data missing)
- 500 → Internal Server Error (server problem)

Easy Memory Trick:
------------------
Think of *GET request like checking your WhatsApp messages*:
- You open the app (send GET request).
- You read the messages (response).
- If everything works fine → 200 OK.
- If chat not found → 404.
- If WhatsApp server is down → 500.

POST Request in Rest Assured
============================

What it is:
```

```
-----------
- A POST request is used to "send data to the server" to create a new resource.
- Example: creating a booking, adding a user, or submitting a form.

Why we use it:
--------------
- To check if the API is accepting and storing new data correctly.
- To validate that a resource is successfully created.
- To test how the server behaves when correct or incorrect data is sent.

How to implement:
-----------------
1. Start building the request with `given()`.
2. Add base URI and base path.
3. Set the Content-Type (usually JSON).
4. Add a request body with `.body()`.
5. Hit the request using `.post()`.
6. Validate response using `.then()`.

Examples (POST Request):
------------------------
Example 1:
    public void POST() {
        // Build request
        RequestSpecification requestSpecification = RestAssured.given();
        requestSpecification = requestSpecification.log().all();
        requestSpecification.baseUri("https://restful-booker.herokuapp.com");
        requestSpecification.basePath("/booking");
        requestSpecification.contentType(ContentType.JSON);
        requestSpecification.body("
              {
               "firstname": "Ravindra Yadav",
               "lastname": "Mahajan",
               "totalprice": 15,
               "depositpaid": false,
               "bookingdates": {
               "checkin": "2021-01-01",
               "checkout": "2021-01-01"
                    },
               "additionalneeds": "Lunch"
                    }
              ");

        // Send POST request
        Response response = requestSpecification.post();

        // Validate response
        ValidatableResponse validatableResponse = response.then();
        validatableResponse.statusCode(200);
    }

Status Codes (for POST):
------------------------
✅ Pass:
- 200 / 201 → Resource created successfully

❌ Fail:
- 400 → Bad Request (wrong or missing body data)
- 500 → Internal Server Error (server-side issue)

Easy Memory Trick:
------------------
Think of "POST request like placing an online order":
- You send your order details (body).
```

- If details are correct → order is created (200/201).
- If details are wrong (like missing address) → rejected (400).
- If the shop’s system crashes → server error (500).

PUT Request
==========

- PUT is a type of HTTP request used to update or create data.
- If the data already exists, PUT replaces it with the new data sent in the request.
- If the data does not exist, PUT can create it.
- PUT is idempotent, meaning doing it once or many times gives the same result.
- It is not a safe method, because it changes data.
- Possible response status codes: 201 (Created), 200 (OK), or 204 (No Content).

Explanation

- What PUT does:
  PUT is used to store the data you send at the specified URL.

- If the URL already has data:
  The server should update (modify) the existing data with the new data.

- If the URL doesn’t have data yet:
  The server can create new data at that URL using the data sent.

- What response you get:
  - If new data is created → Server must send back 201 (Created).
  - If existing data is updated → Server should send 200 (OK) or 204 (No Content).

- If the server can’t create or update:
  The server should send back an error message explaining what went wrong.

PATCH Request (Simplified)
=========================

- PATCH is a type of HTTP request used to partially update existing data.

- Unlike PUT (which replaces the whole data), PATCH only updates the fields you send.

- If a field is not sent in the request, it remains unchanged.
- PATCH is not idempotent in all cases (applying it multiple times might change the result differently depending on the request).

- It is not a safe method, because it changes data.

- Possible response status codes: 200 (OK) if updated, 204 (No Content) if successful without body, 404 (Not Found) if resource doesn’t exist.

PATCH Request explanation

- What PATCH does:
  PATCH is used to apply small changes (partial modifications) to the resource at the specified URL.

- If the URL already has data:
  The server should only update the given fields and leave the rest of the data as is.

- If the URL doesn’t have data yet:

  Usually PATCH is not used to create data (that’s PUT’s job).
  If the resource doesn’t exist, the server may return an error (like 404 Not Found).

- What response you get:
  - If data was successfully updated → server should return 200 (OK) (with response body) or 204 (No Content) (without body).
  - If resource is missing → server can return 404 (Not Found).
  - If request is invalid → 400 (Bad Request).

Easy Memory Trick

Think of PATCH like fixing a hole in your jeans 👖:
- You don’t replace the whole jeans (that’s PUT).
- You just patch the damaged part (only update specific fields).

DELETE Request
==============

What is a DELETE request?
- Used to delete data from the server.
- It can delete one item or all items (if the server allows).
- Common response codes:
  - 200 (OK)
  - 204 (No Content)
  - 404 (Not Found)

Your JSON File:

```
{
  "people": [
    { "id": "1", "name": "Ravindra", "age": 31, "city": "Chennai" },
    { "id": "2", "name": "Bob", "age": 30, "city": "San Francisco" },
    { "id": "4", "name": "Diana", "age": 22, "city": "Los Angeles" },
    { "id": "5", "name": "Syrya", "age": 31, "city": "Mumbai" }
  ]
}
```

Operations You Can Do:

1.Delete one person by ID:
   - URL: http://localhost:3000/people/1
   - This will delete the person with id = "1"

2.Delete another person:
   - URL: http://localhost:3000/people/4
   - This will delete the person with id = "4"

3.Delete all people:
   - URL: http://localhost:3000/people
   - Not allowed in JSON Server (gives error)

4.Try to delete non-existing person:
   - URL: http://localhost:3000/people/99
   - Response: 404 Not Found

Important Points:

- You must use the DELETE method.
- It usually does not need a request body.
- You can delete one person using their ID.
- Deleting again has no effect (idempotent).
- Not a safe method — it changes data.

Rest Assured Script in BDD Format

What it is:

- BDD (Behavior Driven Development) format in Rest Assured means writing API tests in a natural language style:
  Given – When – Then.
- It makes test scripts more readable and closer to English sentences.

How to implement (step-by-step idea):

Build Request - Given

1.Given – Setup the request (base URI, headers, body, etc.).
Think of it as preparing the letter you want to send.

Hit the request and get the response - When

2.When – Perform the action (GET, POST, PUT, DELETE).
Think of it as posting the letter into the mailbox.

Validate the response - Then

3.Then – Validate the response (status code, body, headers).
Think of it as reading the reply letter and checking if it’s correct.

Get Response as a String format

1.What is extract() method?

- The `extract()` method is used in Rest Assured to get the response data from the API call so that you can use it later in your test.
- It allows you to extract:
  - Full response
  - Response body
  - Specific fields (like headers, cookies, status code, etc.)

Example:

```
Response response =
    given()
        .when()
        .get("http://localhost:3000/people/1")
        .then()
        .extract()
        .response();
```

2.Getting the response as String:

You can use:
- response.asString()
- response.asPrettyString()

✅ .asString()

- Converts the response body into a raw string format.
- Everything stays in a single line (no formatting).
- Useful for logging or quick checks.

Example:

```
System.out.println(response.asString());
```

✅ .asPrettyString()

- Converts the response body into a nicely formatted (indented) string.
- Easier to read for humans.
- Great for printing JSON/XML responses.

Example:
System.out.println(response.asPrettyString());

3.Key Difference: asString() vs asPrettyString()

Feature	asString()	asPrettyString()
Formatting	No (plain single-line)	Yes (multi-line, indented)
Use case	Logging, internal use	Displaying, reading easily
Output Style	Compact	Human-readable

4.Sample Output Comparison

Raw JSON response:
```
{
  "id": "1",
  "name": "Alice",
  "age": 25
}
```

- .asString() will print:
```
{"id":"1","name":"Alice","age":25}
```

- .asPrettyString() will print:
```
{
  "id": "1",
  "name": "Alice",
  "age": 25
}
```

5.When to Use Which?

- Use `.asString()` when storing or comparing responses.
- Use `.asPrettyString()` when displaying in logs or printing.

Response Time Validation
==========

1.What is it?

- It means checking how fast or slow the API gives a response.
- Used for performance testing in Rest Assured.

2.Parent Object:

- Response → This is the main object we get after calling the API.
- Example: Response response = given().get("url");

3.Methods from Response:

- response.time()
  → Gives time in milliseconds.

- response.timeIn(TimeUnit.SECONDS)

```
  → Gives time in seconds (or minutes, hours using TimeUnit).

- response.getTime()
  → Same as time(), gives milliseconds.

- response.getTimeIn(TimeUnit.SECONDS)
  → Same as timeIn(), gives seconds (or other units).

4.Validation using Matchers:
-----------------------------
- response.then().time(Matchers.lessThan(4000L))
  → Response time < 4000 ms.

- response.then().time(Matchers.greaterThan(1000L))
  → Response time > 1000 ms.

- response.then().time(Matchers.both(Matchers.lessThan(2000L))
     .and(Matchers.greaterThan(1000L)))
  → Response time between 1000 and 2000 ms.

- response.then().time(Matchers.lessThan(4L), TimeUnit.SECONDS)
  → Response time < 4 seconds.

5.Why use it?
---------------
- To check API speed.
- To make sure server is not too slow.
- Important for performance and SLA testing.

6.Keywords to Remember:
-------------------------
- time() → ms
- getTime() → ms
- timeIn() → seconds/minutes/hours
- getTimeIn() → seconds/minutes/hours
- Matchers → for validations like <, >, between

7.Example Code:
---------------
public void ResponseTimeValidation() {

    Response response = given()
                  .contentType(ContentType.JSON)
                  .when()
                  .get("http://localhost:3000/people");

    long responseInMS = response.time();
    System.out.println("Response time MS ----> " + responseInMS);

    long responseInSeconds = response.timeIn(TimeUnit.SECONDS);
    System.out.println("Response time Seconds ----> " + responseInSeconds);

    long responseInMS1 = response.getTime();
    System.out.println("Response time MS ----> " + responseInMS1);

    long responseInSeconds1 = response.getTimeIn(TimeUnit.SECONDS);
    System.out.println("Response time Seconds1 ----> " + responseInSeconds1);

    response.then().time(Matchers.lessThan(4000L));
    response.then().time(Matchers.greaterThan(1000L));
    response.then().time(Matchers.both(Matchers.lessThan(2000L))
                                  .and(Matchers.greaterThan(1000L)));
    response.then().time(Matchers.lessThan(4L), TimeUnit.SECONDS);
}
```

Rest Assured Static variables

1. What is it?

- These are static configuration variables in Rest Assured.
- They define the common parts of your API URL (host, port, path).
- Once set, you don’t need to repeat them in every test.

2. Parent / Where it belongs

- Belongs to the RestAssured class (all are static).
- Normally set inside @BeforeClass so they apply to all tests.

3. How it works (step-by-step)

1. Set once at class level:
   RestAssured.baseURI = "http://localhost";
   RestAssured.port = 8080;
   RestAssured.basePath = "/api";

2. In your test, just write the relative endpoint:
   given().get("/users");

3. Rest Assured automatically combines:
   baseURI + port + basePath + endpoint
   → http://localhost:8080/api/users

4. If you don’t want to use them in one test, you can still pass the full URL manually.

4. Example

```
@BeforeClass
public void setup() {
    RestAssured.baseURI = "http://localhost";
    RestAssured.port = 8080;
    RestAssured.basePath = "/api";
}

@Test
public void test1() {
    given()
        .get("/users")
        .then()
        .statusCode(200);
}
```

- Actual request sent → http://localhost:8080/api/users

5. Why use it?

- To avoid repeating host/port/path in every test.
- Makes tests cleaner and shorter.
- Easier maintenance → if server changes, update in one place only.
- Supports multiple tests automatically (works across all @Test methods in the same class).

6. Keywords / Memory Tricks

- baseURI = Website address (host)
- port = Door number of house (API server port)

- basePath = Entry gate (API base path)
- endpoint = Final room (specific resource)

Memory Trick → Like stamping an envelope:
- Stamp once in @BeforeClass → All letters (@Test) automatically carry the address.

Interview Tip:

Q: Do we always need to pass the full URL in Rest Assured?
A: No, we can set baseURI, port, and basePath once. Then Rest Assured automatically builds
    the full URL for every request.

Passing Headers in REST Assured
==========

1. What is it?

- Headers = Extra information we send with API request.
- They tell the server how to read or accept the request.

2. Why do we need it?

- To send Auth tokens (login key).
- To tell server request type (JSON/XML).
- To tell server response type we expect (JSON/XML).
- To send custom info (like Env=QA).

3. Where it belongs?

- Part of HTTP request.
- In REST Assured → we set it inside `given()`.

4. How to pass headers?
---
a) One header:
```
given().header("key","value")
```

b) Multiple headers:
```
.header("h1","v1")
.header("h2","v2")
```

c) With Header object:
```
Header h = new Header("key","value")
given().header(h)
```

d) With Headers list:
```
List<Header> list = new ArrayList<>();
list.add(new Header("k1","v1"))
list.add(new Header("k2","v2"))
Headers headers = new Headers(list)
given().headers(headers)
```

5. Example:

```
@Test
public void example() {
   RestAssured.baseURI="http://localhost:3000/people/2";

   Header h = new Header("Auth","Bearer123");

   given()
```

```
      .contentType(ContentType.JSON)
      .header("Env","QA")
      .header(h)
   .when()
      .get()
   .then()
      .statusCode(200);
}

6. Why use it?
--------------
- To pass login keys (Auth).
- To tell server the format (JSON/XML).
- To test with custom values.

7. Keywords / Easy Trick
------------------------
- header() → single
- headers() → many
- Header → one object
- Headers → group

Memory Trick:
-------------
Think of headers like a label on a courier box:
- "Fragile", "This side up", "To: John"
The delivery guy (server) reads the label → knows what to do.


REST Assured – Headers Overriding vs Merge Behavior
===================================================

1. What is it?
Passing headers in REST API requests is common. Sometimes multiple
  headers with the same name exist.

  RestAssured provides two ways to handle this:
- Overwrite: Replace old value with new value.
- Merge: Keep all values together.

2. Where it belongs:
- Used in RestAssured header management.
- Can be applied in RequestSpecification and .config().

3. Key Points / Methods:

a) Overwrite Headers
Example:

RestAssured.given()
    .config(RestAssured.config()
        .headerConfig(HeaderConfig.headerConfig()
            .overwriteHeadersWithName("token")))
    .header("token", "xyz123")   // first value
    .header("token", "xyz789")   // overwritten
    .header("env", "QA")         // another header
    .header("env", "Prod")       // last one wins (default)
    .log().all()
    .when().get();

Output:

 token: xyz789    <-- latest value kept (overwrite applied)
 env: Prod        <-- last value wins (overwrite not applied)
```

b) Merge Headers
Example:

```
RestAssured.given()
    .config(RestAssured.config()
        .headerConfig(HeaderConfig.headerConfig()
            .mergeHeadersWithName("token")))
    .header("token", "xyz123")
    .header("token", "xyz789")
    .header("env", "QA")
    .header("env", "Prod")
    .log().all()
    .when().get();
```

Output:

```
token: xyz123, xyz789   <-- all values merged
env: QA, Prod           <-- all values merged (if merge applied)
```

c) Using RequestSpecification
Example:

```
RequestSpecification req1 = RestAssured.given()
    .header("env", "QA")
    .header("token", "xyz111");

RequestSpecification req2 = RestAssured.given()
    .header("env", "Prod")
    .header("token", "xyz789");

RestAssured.given()
    .config(RestAssured.config()
        .headerConfig(HeaderConfig.headerConfig()
            .overwriteHeadersWithName("token")))
    .spec(req1)
    .spec(req2)
    .log().all()
    .when().get();
```

Output:

```
token: xyz789   <-- overwritten (last one wins for token)
env: Prod       <-- last one wins (overwrite not applied)
```

4. Why use it?
- To control which header values are used in API requests.
- Avoid mistakes when multiple headers with the same name exist.
- Useful for combining multiple RequestSpecifications.
- Merge is helpful when API supports multiple header values.

5. Summary:
- Overwrite → Replace old header value with new value.
- Merge → Keep all header values together.
- For headers not mentioned in overwrite or merge, last one wins.
- Apply .config(HeaderConfig) before specifying headers.

6. Memory Trick:
- Overwrite = Replace
- Merge = Combine
- Last one wins for others

Authentication
==========

1. Basic Authentication
- What: Sends username and password encoded in Base64.
- Use: To access APIs that need username and password.
- Why: Easy way to secure an endpoint.
- Code:
```
given().auth().basic("username", "password").when().get("/endpoint");
```

2. Preemptive Basic Authentication
- What: Sends username and password immediately, without waiting for server.
- Use: Makes login faster.
- Why: Saves time for known endpoints.
- Code:
```
given().auth().preemptive().basic("username",
"password").when().get("/endpoint");
```

3. Digest Authentication
- What: Secure authentication with challenge-response, password not sent directly.
- Use: For APIs needing more security.
- Why: Protects password.
- Code:
```
given().auth().digest("username", "password").when().get("/endpoint");
```

4. OAuth 1.0
- What: Uses tokens and signatures for access.
- Use: Older APIs like Twitter API v1.1.
- Why: Secure third-party access without sharing password.
- Code:
```
given().auth().oauth("consumerKey","consumerSecret","accessToken","secretToken")
.when().get("/endpoint");
```

5. OAuth 2.0
- What: Uses access token instead of username/password.
- Use: Modern APIs like Google, GitHub, Facebook.
- Why: Simple and secure.
- Code:
```
given().auth().oauth2("accessToken").when().get("/endpoint");
```

6. Form Authentication
- What: Sends credentials through a login form.
- Use: Web apps with login forms.
- Why: Lets us test web login flows.
- Code:
```
given().auth().form("username","password").when().get("/endpoint");
```

7. Preemptive Form Authentication
- What: Sends form credentials immediately.
- Use: Makes login faster.
- Why: Saves time when server allows.
- Code:
```
given().auth().preemptive().form("username","password").when().get("/endpoint");
```

8. Manual Bearer Token Authentication
- What: Uses bearer token in header.
- Use: APIs needing token access.
- Why: Secure access without username/password.
- Code:
```
given().header("Authorization","Bearer
your_token_here").when().get("/endpoint");
```

JSON Path
=========

1. What is JSON Path?
- JSON Path is used to extract data from a JSON response.
- It is similar to XPath, but for JSON instead of XML.
- You can search for specific values or elements in JSON using JSON Path.
- RestAssured uses Groovy’s GPath style for JSON Path.
- You do not need to install anything extra to use JSON Path in RestAssured.

2. Basic Rules:
- The root node is represented by a dollar sign ($).
- Child nodes are accessed using dot notation (e.g., parent.child).
- Square brackets [] are used to access elements in arrays.
- You cannot use square brackets to represent a child node (except for arrays).

3. Sample JSON Response:

```
{
  "person": {
    "id": 101,
    "name": "John",
    "contact": {
      "email": "john@example.com",
      "phone": "1234567890"
    },
    "skills": ["Java", "Selenium", "RestAssured"]
  }
}
```

4. Example Code in RestAssured:

```
import io.restassured.RestAssured;
import io.restassured.path.json.JsonPath;
import io.restassured.response.Response;
import org.testng.annotations.Test;

public class JsonPathExample {

    @Test
    public void readJsonData() {
        // Send GET request
        Response res = RestAssured.get("http://localhost:3000/person/101");

        // Convert response to JsonPath object
        JsonPath js = res.jsonPath();

        // Extract data using JSON Path
        int id = js.getInt("person.id");
        String name = js.getString("person.name");
        String email = js.getString("person.contact.email");
        String firstSkill = js.getString("person.skills[0]");

        // Print the values
        System.out.println("ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Email: " + email);
        System.out.println("First Skill: " + firstSkill);
    }
}
```

5. Notes for Testers:
- Use dot notation to navigate child nodes.
- Use square brackets [] only for array elements.
- JSON Path is case-sensitive.
- It is useful to validate response data in your API tests.
- You can use it to verify values, lists, and nested objects in JSON.

Some more points about Json
---

1. When you may encounter ClassCastException?
   - Happens when trying to cast one data type to an incompatible type.
   - Example: Casting a number to a string without conversion.

2. What is Root node?
   - "$"

3. What is Simple JSON Object?
   - JSON with only key-value pairs, no objects inside.
   Example: { "name": "John", "age": 25 }

4. What is Nested JSON Object?
   - JSON object that contains another JSON object inside.
   Example: { "name": "John", "address": { "city": "London" } }

5. JSON Path examples:
   - Simple: $.name → "John"
   - Nested: $.address.city → "London"

6.If trying with the below JSON:
   { "name": "John", "age": 25 }

 - String name1 = jsonPath.getString("name1");
  → Returns null and does not throw an exception.

 - int name1 = jsonPath.getInt("name1");
  → Throws NullPointerException because null cannot be assigned to a primitive int.

JSON Array Notes
==========

1. Definition:
A JSON Array is a collection of multiple values stored in an ordered list.
- Arrays are index-based, meaning each element has a position starting from 0.
- Arrays can contain strings, numbers, booleans, objects, or even other arrays.

2. Basic JSON Array Example:
["amod", "mukesh", "rahul"]
- "amod" is at index 0, "mukesh" at index 1, "rahul" at index 2.

3. JSON Array with Numbers:
[10, 20, 30, 40]
- Index 2 gives 30.

4. JSON Array with Mixed Types:
["John", 25, true, null]
- "John" → String
- 25 → Number
- true → Boolean
- null → Null value

5. JSON Array of Objects:
```
[
  {"name": "John", "age": 25},
  {"name": "Alice", "age": 30}
]
```
- Index 0 → {"name": "John", "age": 25}
- Index 1 → {"name": "Alice", "age": 30}

6. Nested JSON Array:

```
[
  "Amod",
  ["Java", "Selenium", "RestAssured"],
  100
]
```

- Index 0 → "Amod"
- Index 1 → Array ["Java", "Selenium", "RestAssured"]
- Index 2 → 100

7. Accessing JSON Array in Rest Assured using JsonPath

Example JSON Response:

```
{
  "students": ["John", "Alice", "Bob"]
}
```

Java Code:

```
JsonPath js = response.jsonPath();

// Access first element
String firstStudent = js.getString("students[0]");
System.out.println(firstStudent);  // Output: John

// Access all elements
List<String> allStudents = js.getList("students");
System.out.println(allStudents);   // Output: [John, Alice, Bob]
```

Another Example with Array of Objects:

```
{
  "students": [
    {"name": "John", "age": 25},
    {"name": "Alice", "age": 30}
  ]
}
```

Java Code:

```
JsonPath js = response.jsonPath();

// Access name of first student
String firstName = js.getString("students[0].name");
System.out.println(firstName); // Output: John

// Access age of second student
int secondAge = js.getInt("students[1].age");
System.out.println(secondAge);  // Output: 30
```

Key Points:

- JSON Array is ordered.
- Index starts from 0.
- Can contain multiple types including objects and arrays.
- Use get() or getString() with index to extract values.

Tricky Json array reading

==========

Example 1: Single JSON Array

JSON Array:

```
   [
      "10",
      "20",
```

```
      "30",
      "40",
      "50"
    ]

Code:
     String jsonArray = "above json array";
     JsonPath jsp = new JsonPath(jsonArray);
     System.out.println(jsp.getString("[3]"));
     System.out.println(jsp.getList("$").size());
     System.out.println((Object) jsp.get("$"));

Output:
 40
 5
 [10, 20, 30, 40, 50]


Example 2: Nested JSON Array
----------------------------

JSON Array:
  [
    ["10", "20", "30", "40", "50"],
    ["100", "200", "300", "400", "500", "600"]
  ]

Code:
   String jsa = "above json";

   JsonPath jsp = new JsonPath(jsa);
   System.out.println(jsp.getString("[1][3]"));
   System.out.println(jsp.getList("$").size());
   System.out.println(jsp.getList("$").get(1));
   List<Object> list = jsp.getList("$");
   List<Object> li = (List) list.get(1);
   System.out.println(li.size());

Output:
 400
 2
 [100, 200, 300, 400, 500, 600]
 6

Nested Json reading
===================

 [
  {
    "firstName": "Virat",
    "lastName": "Kohli",
    "age": 35,
    "address": [
      {
        "city": "Delhi",
        "country": "India"
      },
      {
        "city": "Mumbai",
        "country": "India"
      }
    ]
  },
  {
```

```
    "firstName": "Mahendra",
    "lastName": "Singh Dhoni",
    "age": 42,
    "address": [
      {
        "city": "Ranchi",
        "country": "India"
      },
      {
        "city": "Chennai",
        "country": "India"
      }
    ]
  }
 ]

   String array = "above array";

   JsonPath jp = new JsonPath(array);

   // Access specific city in nested array
   System.out.println(jp.getString("[0].address[1].city")); // Output: Mumbai
   System.out.println(jp.getString("[1].address[1].city")); // Output: Chennai

   // Access list of cities from nested array for each player
   System.out.println(jp.getList("[0].address.city")); // Output: [Delhi,
Mumbai]
   System.out.println(jp.getList("[1].address.city")); // Output: [Ranchi,
Chennai]

   // Correct way to get all cities from all players
   System.out.println(jp.getList("address.city")); // OP:[[Delhi, Mumbai],
[Ranchi, Chennai]]


  JsonPath Filter Expressions with $.[?(@.key operator value)]

Theory:
1. $ → Represents the root of the JSON (entire object or array).
2. [?()] → Filter expression used to filter elements from an array.
3. @ → Represents the current element in the array.
4. key operator value → Condition to filter elements.Common operators: ==, !=,
>, <, >=, <=.

Syntax:
   $.[?(@.key operator value)]

   - $ → root
   - [?()] → filter expression
   - @.key → key in current element
   - operator → comparison operator
   - value → value to compare

Example JSON:
  {
    "data": [
    {"id": 1, "first_name": "Virat",
    "last_name": "Kohli", "email": "vkohli@example.com", "gender": "Male"},

    {"id": 2, "first_name": "Shellie",
     "last_name": "Cowser", "email": "scowser1@163.com", "gender": "Female"},

    {"id": 3, "first_name": "Mithali",
    "last_name": "Raj", "email": "mithali.raj@example.com", "gender": "Female"},
```

```
    {"id": 4, "first_name": "Steve",
    "last_name": "Smith", "email": "steve.smith@example.com", "gender": "Male"},

    {"id": 5, "first_name": "Ellyse",
    "last_name": "Perry", "email": "ellyse.perry@example.com", "gender":
"Female"}
  ],
  "teams": [
    {"team_id": 101, "team_name": "India Cricket Team",
     "members": ["Virat", "Dhoni", "Rohit"]},
    {"team_id": 102, "team_name": "Australia Cricket Team",
     "members": ["Steve", "Ellyse", "Mitchell"]}
  ]
}


JsonPath Filters
================

Json Data-->
{
  "data": [
   {"id": 1, "first_name": "Virat", "last_name": "Kohli",
    "email": "kohli@example.com", "gender": "Male"},
   {"id": 2, "first_name": "Shellie", "last_name": "Cowser",
    "email": "scow@163.com", "gender": "Female"},
   {"id": 3, "first_name": "Mitha", "last_name": "Raj",
    "email": "mitha@example.com", "gender": "Female"},
   {"id": 4, "first_name": "Steve", "last_name": "Smith",
    "email": "steve@example.com", "gender": "Male"},
   {"id": 5, "first_name": "Elly", "last_name": "Perry",
    "email": "elly@example.com", "gender": "Female"}
  ],
  "teams": [
    {"team_id": 101, "team_name": "India Cricket Team",
     "members": ["Virat", "Dhoni", "Rohit"]},
    {"team_id": 102, "team_name": "Australia Cricket Team",
     "members": ["Steve", "Ellyse", "Mitchell"]}
  ]
}


Java Example using RestAssured JsonPath:
import io.restassured.path.json.JsonPath;
import java.util.List;

public class JsonPathFilterExample {
 public static void main(String[] args) {
  String json = "..."; // your JSON here
  JsonPath jp = new JsonPath(json);

  // Filter all females
  List<Object> femaleNames = jp.getList("data.findAll { it.gender ==
'Female' }.first_name");
  System.out.println("Female Names ---> " + femaleNames);

  // Filter single person
  String emailId = jp.getString("data.find { it.first_name == 'Mithali' &&
it.last_name == 'Raj' }.email");
  System.out.println("Email of Mithali Raj ---> " + emailId);

  // Size of data array
  int totalCount = jp.getInt("data.size()");
```

```
  System.out.println("Total Count ---> " + totalCount);

  // Filter using $.[?(@.gender == 'Female')]
  List<Object> femaleObjects = jp.getList("data.findAll { it.gender ==
'Female' }");
  System.out.println("Female Objects ---> " + femaleObjects);

  // Filter teams having 'Ellyse'
  List<Object> teamWithEllyse = jp.getList("teams.findAll
{ it.members.contains('Ellyse') }.team_name");
  System.out.println("Teams with Ellyse ---> " + teamWithEllyse);
    }
}

Expected Output:
   Female Names ---> [Shellie, Mithali, Ellyse]
   Email of Mithali Raj ---> mithali.raj@example.com
   Total Count ---> 5
   Female Objects ---> [{id=2, first_name=Shellie,
      last_name=Cowser, email=scowser1@163.com, gender=Female},
      {id=3, first_name=Mithali, last_name=Raj, email=mithali.raj@example.com,
gender=Female},
      {id=5, first_name=Ellyse, last_name=Perry, email=ellyse.perry@example.com,
gender=Female}]
   Teams with Ellyse ---> [Australia Cricket Team]

Key Notes:
1. findAll → returns all elements matching condition.
2. find → returns first element matching condition.
3. size() → returns total count of elements.
4. $.[?(@.key operator value)] → JSONPath style filter.
5. Combine conditions using && (AND) or || (OR).
6. Can query nested arrays using contains().

In-line Assertions in Rest Assured
==================================

What it is:
In-line assertions are checks you do right inside your API request to make sure
the response is correct immediately.
Think of it like checking your homework while writing it, not after finishing.
In Rest Assured, we write them after .then() using .body() and Matchers.

What it is used for:
- To check API response instantly while writing the request.
- To save code because you don’t need to extract response first and then write
separate checks.
- Makes your test short, clear, and easy to read.

Memory trick: "Check while you fetch" → inline assertions = check response while
fetching it.

How to implement:
1. Start your request: RestAssured.given().
2. Add base URL, headers, body if needed.
3. Send request: .when().get()/post()/put().
4. Start checking response: .then().
5. Use .body("field", Matchers.something()) to check any field.

Example:

RestAssured.given()
    .log().all()
    .baseUri("https://restful-booker.herokuapp.com/auth")
```

```
    .body("{ \"username\":\"admin\", \"password\":\"password123\" }")
    .contentType(ContentType.JSON)
.when()
    .post()
.then()
    .log().all()
    .body("token", Matchers.notNullValue())      // token exists
    .body("token.length()", Matchers.is(15))    // token length is 15
    .body("token", Matchers.matchesRegex("^[a-z0-9]+$")); // token has only
letters and numbers
```

Why we use it:
- Quick debugging: If it fails, you know exactly where.
- Less code: No need to extract response and write multiple asserts.
- Clear view: Request + validation in one place.
- Works for JSON, XML, or any response type.

Memory trick: "Inline = Instant Check" → check response instantly, no waiting.

Examples:
1. Check token exists:
```
   .body("token", Matchers.notNullValue());
```

2. Check token length:
```
   .body("token.length()", Matchers.is(15));
```

3. Check token pattern:
```
   .body("token", Matchers.matchesRegex("^[a-z0-9]+$"));
```

4. Check a list contains certain IDs:
```
   .body("bookingid", Matchers.hasItems(91, 10));
```

Easy interview trick:
Think "Watch while you fetch" →
- You are watching the response while fetching it.
- Use .then().body() + Matchers to check values immediately.

Schema Validation in Rest Assured
==========

What it is:
Schema validation is checking if the API response has the correct structure and types of data.
Think of it like checking if a house is built exactly according to the blueprint.
In Rest Assured, we use JSON schema or XML schema to verify this.

What it is used for:
- To make sure the API returns data in the correct format.
- To catch errors in the API response structure early.
- To save time instead of checking each field manually.

Memory trick: "Blueprint check" → schema validation = check response matches the blueprint.

How to implement:
1. Create a JSON schema file (example: schema.json) with the expected structure.
2. Use RestAssured.given() to start the API request.
3. Set base URL, headers, and body if needed.
4. Send request with .when().get()/post()/put().
5. Add .then().assertThat().body(matchesJsonSchemaInClasspath("schema.json")) to check the response.

Example:

```
RestAssured.given()
    .log().all()
    .baseUri("https://restful-booker.herokuapp.com/booking/1")
.when()
    .get()
.then()
    .log().all()
    .assertThat()
    .body(matchesJsonSchemaInClasspath("bookingSchema.json"));
```

Why we use it (purpose/uses):
- Makes testing faster: You don’t check every field one by one.
- Ensures API reliability: Any missing or wrong field is detected.
- Easy maintenance: If API changes, update schema once.
- Works for big or complex responses: No manual checking needed.

Memory trick: "Schema = Blueprint" → verify response is built exactly as planned.

Examples:
1. Check if response has fields like id, name, price in correct types.
2. Ensure nested objects (like address inside user) exist and follow schema.
3. Validate lists/arrays have correct elements and types.

Easy interview trick:
Think of schema validation as "Check the blueprint":
- Before using the house (API response), make sure it is built exactly like the plan.
- Use matchesJsonSchemaInClasspath() in Rest Assured to check automatically.

SON Schema Resources
---

1. JSON Schema Grammar
   - https://json-schema.org/

2. Generate JSON Schema
   - https://www.jsonschema.net/home
   - https://www.liquid-technologies.com/online-json-to-schema-converter

3. Validate JSON Document Against Schema
   - https://jsonschemalint.com/#/version/draft-07/markup/json
   - https://www.jsonschemavalidator.net/

Payload in Rest Assured (Request and Response)
==========

What it is:
Payload is the data that travels between you and the API.
There are two types:
1. Request Payload – the data you send to the API (like a letter with instructions).
2. Response Payload – the data the API sends back to you (like a reply letter with results).

What it is used for:
- Request payload: to tell the API what to do (create, update, login, etc.).
- Response payload: to check what the API returned (success, token, details, errors).

Memory trick: “Send letter = request, Get reply = response”

How to implement:

1. Request Payload:
   - Create JSON or XML with data you want to send.
   - Use .body(payload) in Rest Assured before sending POST/PUT request.
   Example:
     { "username": "admin", "password": "password123" }

2. Response Payload:
   - Capture the response using .extract().asString() or .body() checks.
   - Validate or read fields using JsonPath or inline assertions.
   Example:
     String response = RestAssured.given()...
     JsonPath json = new JsonPath(response);
     String token = json.get("token");

Why we use it:
- Request payload: to give API input data.
- Response payload: to verify output and make sure API worked correctly.
- Helps automate tests and validate API behavior quickly.

Examples:
1. Request payload:
   - Send username/password to login API.
   - Send booking details (name, date, room) to create reservation.

2. Response payload:
   - Receive token after login.
   - Receive booking id and details after creating reservation.
   - Check errors if API fails.

Memory trick for interviews:
“Letter goes = request, Letter comes = response” → easy to remember.

Creating Payload with Map
----

1. Create a Map<String, Object>.
   Example:
      Map<String, Object> player = new HashMap<>();
      player.put("first_name", "Mahendra");
      player.put("last_name", "Dhoni");
      player.put("married", true);
      player.put("salary", 2200000.00);

2. Use RestAssured to send it:
      RestAssured.given()
          .contentType(ContentType.JSON)
          .baseUri("http://localhost:3000/players")
          .body(player)   // Map will convert to JSON automatically
          .log().all()
      .when()
          .post();

Creating nested Payload

        Map<String, Object> player = new LinkedHashMap<>();

            player.put("first_name", "Mahendra");
            player.put("last_name", "Dhoni");
            player.put("married", true);
            player.put("salary", 2200000.00);

            Map<String,Object> address = new LinkedHashMap();

```
            address.put("no", "Ranchi House");
            address.put("streetName","Harmu Road");
            address.put("streetName", "Harmu Road");
            address.put("state", "Jharkhand");

            player.put("address", address);

            RestAssured.given().contentType(ContentType.JSON)
            .baseUri("http://localhost:3000/players")
            .body(player)
            .log()
            .all()
            .when()
            .post();
```

Creating Payload by List

```
            Map<String, Object> data1 = new LinkedHashMap();

            data1.put("first_name", "Virat");
            data1.put("last_name", "Kohli");
            data1.put("email", "vkohli@bcci.in");
            data1.put("gender", "Male");

            List<Map<String, Object>> payLoad = new LinkedList<Map<String,
Object>>();
            payLoad.add(data1);

            Map<String, Object> data2 = new LinkedHashMap();

            data2.put("first_name", "Smriti");
            data2.put("last_name", "Mandhana");
            data2.put("email", "smandhana@bcci.in");
            data2.put("gender", "Female");
            payLoad.add(data2);

            RestAssured.given().contentType(ContentType.JSON)
            .baseUri("http://localhost:3000/players")
            .body(payLoad)
            .log()
            .all()
            .when()
            .post();
```

Creating complex json payload

```
            List<Map<String, Object>> payLoad = new LinkedList<>();

            Map<String, Object> object = new LinkedHashMap<>();
            object.put("id", 1);
            object.put("first_name", "cdennerley@uol.com.br");
            object.put("email", 1);
            object.put("gender", "Genderfluid");

            List<String> mobileNumber = new ArrayList<>();
            mobileNumber.add("1232432432");
            mobileNumber.add("324324324");
            // Alternatively: List<String> mobileNumber =
Arrays.asList("1232432432",
            // "324324324");

            object.put("mobile", mobileNumber);
```

```
Map<String, Object> object1 = new LinkedHashMap<>();
object1.put("name", "Testing");
object1.put("proficiency", "Medium");

object.put("skills", object1);

payLoad.add(object);

Map<String, Object> object2 = new LinkedHashMap<>();
object2.put("id", 2);
object2.put("first_name", "Cloe");
object2.put("last_name", "Stuehmeyer");
object2.put("email", "cstuehmeyerl@yellowpages.com");
object2.put("gender", "Female");

List<Map<String, Object>> skillsList = new ArrayList<>();

// Skill 1: Testing
Map<String, Object> skillA = new LinkedHashMap<>();
skillA.put("name", "Testing");
skillA.put("proficiency", "Medium");
skillsList.add(skillA);

// Skill 2: Java with Certifications
Map<String, Object> skillB = new LinkedHashMap<>();
List<String> certifications = new ArrayList<>();
certifications.add("OCJP 11");
certifications.add("OCJP 12");

skillB.put("name", "Java");
skillB.put("proficiency", "Medium");
skillB.put("certifications", certifications);
skillsList.add(skillB);

// Add skills to object2
object2.put("skills", skillsList);

RestAssured.given().contentType(ContentType.JSON)
.baseUri("http://localhost:3000/players")
.body(payLoad)
.when()
.post()
.then()
.log().body().toString();
```

```
Response in Rest Assured
========================

What it is:
- Response is the data you get back from the API after sending a request.
- Think of it as the answer from the server.
- It can include:
  - Status code (like 200, 404)
  - Response body (data in JSON or XML)
  - Headers (extra info like content type)

What is the use:
- To check if the API worked correctly.
- To get the data returned by the API and use it in tests.
- To validate that the API response matches what you expect.

How to implement:
```

1. Send a request using RestAssured.given().
2. Capture the response using .when().get()/post()/put() with .then() or directly .extract().response().
3. You can check:
   - Status code → response.getStatusCode()
   - Body → response.getBody().asString() or response.jsonPath().get("key")
   - Headers → response.getHeader("headerName")

Why we use it (purpose/uses):
- To verify API is working properly.
- To check returned data against expected values.
- To debug API issues quickly if the response is wrong.
- To read data from API and use it in further requests.

Examples:

Example 1: Simple GET request and get response

```
Response response = RestAssured.given()
                    .baseUri("https://restful-booker.herokuapp.com/booking/1")
                    .when()
                    .get();

System.out.println(response.getStatusCode());    // prints 200
System.out.println(response.getBody().asString()); // prints response body
```

Example 2: Validate response using jsonPath

```
String firstname = response.jsonPath().get("firstname");
System.out.println(firstname);  // prints the firstname value
```

Memory trick:
- Think: “Response = Reply from Server” → Like sending a letter (request) and getting a reply (response).
- You can quickly check the status, read the message, or check the info (headers/body).

Convert Response to Map Object

```
==============================

          Response res =  RestAssured.given().contentType(ContentType.JSON)
             .baseUri("http://localhost:3000/data")
             .log()
             .all()
             .when()
             .get()
             .then()
             .extract()
             .response();

             Map map = res.as(Map.class);

             System.out.println("Id ---> "+map.get("id"));

             Map<String,String> skils = (Map<String,String>) map.get("skills");
             System.out.println("Name --> " + skils.get("name"));
             System.out.println("Proficiency --> " + skils.get("proficiency"));
```

Convert Response to Map Object by typeRef

```
========================================

             Response res =  RestAssured.given().contentType(ContentType.JSON)
             .baseUri("http://localhost:3000/data")
             .log()
             .all()
             .when()
```

```
            .get()
            .then()
            .extract()
            .response();

            Map<String,Object> map = res.as(new TypeRef<Map<String,Object>>()
{});

            System.out.println("Id ---> "+map.get("id"));

            Map<String,String> skils = (Map<String,String>) map.get("skills");
            System.out.println("Name --> " + skils.get("name"));
            System.out.println("Proficiency --> " + skils.get("proficiency"));

            for (String s : map.keySet()) {
                  System.out.println(map.get(s));
            }
```

Convert Response to List Object by typeRef
==========

```
            Response res =  RestAssured.given().contentType(ContentType.JSON)
            .baseUri("http://localhost:3000/data")
            .log()
            .all()
            .when()
            .get()
            .then()
            .extract()
            .response();

            TypeRef<List<Map<String,Object>>> typRef =     new
TypeRef<List<Map<String,Object>>>() {};

            List<Map<String,Object>> list =  res.as(typRef);

            System.out.println("Id ---> "+list.get(0).get("id"));

             for(String s2 : list.get(0).keySet()) {
                  System.out.println(s2 +" : "+list.get(0).get(s2));
            }
```

Convert Response to Object by typeRef
==========

```
             Response res =  RestAssured.given().contentType(ContentType.JSON)
            .baseUri("http://localhost:3000/data")
            .log()
            .all()
            .when()
            .get()
            .then()
            .extract()
            .response();

            Object ob = res.as(Object.class);

            if(ob instanceof List ) {
                  List list = (List) ob;
                   System.out.println("List Object ----> " + list.size() );
            }else if(ob instanceof Map) {
                  Map mp = (Map)ob;
                  System.out.println("Map object ---> "+mp.keySet());
```

```
            }
```

Sharing Data to Other API (API Chaining)

What it is:
- Sometimes one API gives data (like token, id, booking id) that you need to use in another API.
- Sharing data means taking the output from one API (response) and using it as input
  (payload, header, path) in another API.
- Example: First call login API to get token, then use that token in booking API.

What it is used for:
- To connect multiple API calls in a flow.
- To reuse data between APIs instead of hardcoding.
- To test real-world scenarios (because most systems use one API’s output in another API’s input).

Memory trick: “First get the key, then open the door” → login API gives token (key),
booking API needs it (door).

How to implement:
1. Call the first API and capture response.
   Example: get token from login API.
   String token = response.jsonPath().get("token");

2. Use that token or data in the next API request.
   Example: send token in header of booking API.

```
   RestAssured.given()
       .header("Authorization", "Bearer " + token)
       .when()
       .get();
```

3. You can share any value like id, name, token from one API to another.

Why we use it (purpose/uses):
- APIs often work together, not alone.
- Many APIs need authentication token or id from a previous call.
- Makes automation tests realistic and connected.
- Avoids hardcoding values.

Examples:

Example 1: Login and use token

```
Response loginResponse = RestAssured.given()
     .baseUri("https://restful-booker.herokuapp.com/auth")
     .body("{ \"username\":\"admin\", \"password\":\"password123\" }")
     .contentType(ContentType.JSON)
     .when()
     .post();

String token = loginResponse.jsonPath().get("token");

     RestAssured.given()
    .baseUri("https://restful-booker.herokuapp.com/booking")
    .header("Cookie", "token=" + token)
    .when()
    .get()
```

```
    .then()
    .statusCode(200);

Example 2: Create booking and use booking id
-------------------------------------------

Response bookingResponse = RestAssured.given()
     .baseUri("https://restful-booker.herokuapp.com/booking")
     .body("{ \"firstname\":\"John\", \"lastname\":\"Doe\" }")
     .contentType(ContentType.JSON)
     .when()
     .post();

int bookingId = bookingResponse.jsonPath().get("bookingid");

RestAssured.given()
    .baseUri("https://restful-booker.herokuapp.com/booking/" + bookingId)
    .when()
    .get()
    .then()
    .statusCode(200);

Memory trick:
“API chain = Pass the baton” → Like a relay race, one API gives data (baton),
next API runs with it.

Json array payload by POJO class
================================

  public class Employee {

        private int id;
        private String name;
        private String city;
        private String country;
        private String role;

      {Setters getters methods}
  }

Scripting class
---------------

  public class ScriptingClass {

      @Test
      public void getVlaues() {

            Employee emp = new Employee();
            emp.setId(100);
            emp.setName("Ravindra Jadeja");
            emp.setCity("Baroda");
            emp.setRole("All Rounder");
            emp.setCity("India");

            RestAssured.given().contentType(ContentType.JSON)
            .body(emp)
            .log()
            .all()
            .when().get();
      }
}

JSON Array Payload with POJO List
```

```
================================

    public class Employee {

      private int id;
      private String name;
      private String city;
      private String country;
      private String role;

    {Setters getters methods}
 }

 Scripting class
 ---------------

    public void getVlaues() {

            Employee emp1 = new Employee();
            emp1.setId(100);
            emp1.setName("Ravindra Jadeja");
            emp1.setCity("Baroda");
            emp1.setRole("All Rounder");
            emp1.setCountry("India");

            Employee emp2 = new Employee();
            emp2.setId(200);
            emp2.setName("Ravindra Nath Tagur");
            emp2.setCity("Kolkatha");
            emp2.setRole("Freedom Fighter");
            emp2.setCountry("India");

            List<Employee> list = new LinkedList();
            list.add(emp1);
            list.add(emp2);

            RestAssured.given().contentType(ContentType.JSON)
            .body(list)
            .log()
            .all()
            .when().get();
      }

JSON Array Payload with Nested POJO
==================================

  public class Employee {

      private String firstName;
      private String lastname;
      private String profession;
      private List<Address> address;

      {Setters getters methods}

      }

  public class Address {

      private int houseNo;
      private String streetName;
      private String city;
      private String state;
      private String country;
```

```
     {Setters getters methods}
 }

 Payload method
 --------------

    public void payLoad() {

            Employee emp = new Employee();
            emp.setFirstName("Mahendra singh");
            emp.setLastname("Dhoni");
            emp.setProfession("Cricketer");

            Address addr1 = new Address();
            addr1.setHouseNo(103);
            addr1.setStreetName("Ranchi Street");
            addr1.setState("JH");
            addr1.setCity("Ranchi");
            addr1.setCountry("India");

            Address addr2 = new Address();
            addr2.setHouseNo(104);
            addr2.setStreetName("Marvel Street");
            addr2.setCity("TN");
            addr2.setState("Tamil nadu");
            addr2.setCountry("India");

            List<Address> list = new LinkedList();
            list.add(addr1);
            list.add(addr2);

            emp.setAddress(list);



            RestAssured.given().contentType(ContentType.JSON)
            .body(emp)
            .log()
            .all()
            .when()
            .get();

      }

Jackson Databind in Rest Assured
================================

1. What it is
- Jackson Databind is a Java library.
- It converts between Java Objects and JSON (both ways).
- Object → JSON (Serialization)
- JSON → Object (Deserialization)
- Memory trick: Jackson is like a translator between Java and JSON.

2. What is the use
- APIs mostly use JSON.
- In Java, we use objects (POJOs).
- Jackson makes it easy to map JSON to Java objects and vice versa.

3. How to implement
- Step 1: Add Maven dependency
  <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
```

```
      <artifactId>jackson-databind</artifactId>
      <version>2.x.x</version>
  </dependency>

- Step 2: Create a POJO class (like Employee).

- Step 3: Use ObjectMapper
  ObjectMapper mapper = new ObjectMapper();

  // Object → JSON
  String json = mapper.writeValueAsString(emp);

  // JSON → Object
  Employee empObj = mapper.readValue(jsonString, Employee.class);

4. Why we use it (purpose/uses)
- To handle JSON easily in Java tests.
- Avoids manual parsing of JSON.
- Works well with Rest Assured because it auto-converts objects to JSON.
- Shortcut to remember:
  Serialization = Save (Object → JSON)
  Deserialization = Download (JSON → Object)

5. Examples

Example 1: Object → JSON
Employee emp = new Employee();
emp.setId(101);
emp.setName("Virat");

ObjectMapper mapper = new ObjectMapper();
String json = mapper.writeValueAsString(emp);
System.out.println(json);

Output:
{"id":101,"name":"Virat"}

Example 2: JSON → Object
String json = "{\"id\":102,\"name\":\"Rohit\"}";

ObjectMapper mapper = new ObjectMapper();
Employee emp = mapper.readValue(json, Employee.class);

System.out.println(emp.getName()); // Rohit

Quick Recall Tip:
- Jackson = JSON Translator
- writeValueAsString → Write Object as JSON
- readValue → Read JSON into Object


Response JSON Object to POJO
============================

  public class Employee {

      private String firstName;
      private String lastname;
      private String profession;

      {Setters getters methods}

 }
```

```
 Payload class
 -------------

 public class PayLoadClass {

      RequestSpecification specification;

      @BeforeClass
      public void setUp() {

            specification = RestAssured.given()
                        .contentType(ContentType.JSON)
                        .baseUri("http://localhost:3000/data/1");
      }

    @Test
    public void payLoad() {
        Employee emp = specification
                .get()
                .as(Employee.class);

        System.out.println(emp.getName());
        System.out.println(emp.getCountry());
    }
}

Nested response to POJO
=======================

  public class Employee {

      private String firstName;
      private String lastname;
      private String profession;
      private List<Address> address;

      {Setters getters methods}

  }

  public class Address {

      private int houseNo;
      private String streetName;
      private String city;
      private String state;
      private String country;

      {Setters getters methods}

 }

 Payload class
 -------------

  public class PayLoadClass {

      RequestSpecification specification;

      @BeforeClass
      public void setUp() {

            specification = RestAssured.given()
                        .contentType(ContentType.JSON)
```

```
                        .baseUri("http://localhost:3000/data/1");
      }

    @Test
    public void payLoad() {
        Employee emp = specification
                .get()
                .as(Employee.class);

        System.out.println(emp.getName());
        System.out.println(emp.getEmail());

        Address addr =  emp.getAddress();
        System.out.println(addr.getCity());
    }
}


Extracting Specific Part of Response
====================================

  public class Employee {

      private String firstName;
      private String lastname;
      private String profession;
      private List<Address> address;

      {Setters getters methods}

  }

  public class Address {

      private int houseNo;
      private String streetName;
      private String city;
      private String state;
      private String country;

      {Setters getters methods}

 }

Payload class
-------------

  public class PayLoadClass {

      RequestSpecification specification;

      @BeforeClass
      public void setUp() {

            specification = RestAssured.given()
                        .contentType(ContentType.JSON)
                        .baseUri("http://localhost:3000/data/1");
      }

    @Test
    public void payLoad() {
        Address addr = specification
                .get()
                .jsonPath()
```

```
                .getObject("address", Address.class);

        System.out.println(addr.getCity());
    }
}
```

Response JSON Array to POJO by Java Array
==========

```
  public class Employee {

      private String firstName;
      private String lastname;
      private String profession;
      private Address [] address;

      {Setters getters methods}

  }

  public class Address {

      private int houseNo;
      private String streetName;
      private String city;
      private String state;
      private String country;

    {Setters getters methods}

 }
```

Payload class

```
  public class PayLoadClass {

      RequestSpecification specification;

      @BeforeClass
      public void setUp() {

            specification = RestAssured.given()
                        .contentType(ContentType.JSON)
                        .baseUri("http://localhost:3000/data");
      }

    @Test
    public void payLoad() {
      Address[] address = specification
                .get()
                .as(Address[].class);

      System.out.println(address[0].getCity());
      System.out.println(address[1].getCity());
    }
}
```

JSON Response Array to POJO by Java List
==========

```
   public class Employee {

        private Address[] address;
```

```
        {Setters getters methods}

 }

 public class Address{

      private int houseNo;
      private String streetName;
      private String city;
      private String state;
      private String country;
      private String id;

      {Setters getters methods}
}
```

Payload class

```
public class PayLoadClass {

      RequestSpecification specification;

      @BeforeClass
      public void setUp() {

            specification = RestAssured.given()
                        .contentType(ContentType.JSON)
                        .baseUri("http://localhost:3000/data");
      }

    @Test
    public void payLoad() {
      List<Address> address = specification
                .get()
                .as(new TypeRef<List<Address>>() {});

      System.out.println(address.get(0).getCity());
      System.out.println(address.get(1).getCity());
    }
}
```

Handling Dynamic Payload in Response (Rest Assured)

==========

1.What it is:
When an API response changes each time (not fixed), it is called a dynamic response.
Example:
- Order ID changes every request
- Token/Session value changes
- Date/time fields update automatically

So we need to handle these changing values in tests.

2. What is the use:
- To correctly test APIs even if response values change
- To avoid failures due to dynamic values (e.g., test expecting 123 but actual is 456)
- To extract values from one API response and use them in another (API chaining)

3. How to implement:
(a) Using JsonPath

```
   String id = response.jsonPath().getString("id");
```

(b) Using POJO
   User user = response.as(User.class);
   System.out.println(user.getId());

(c) Using Map
   Map<String, Object> map = response.as(Map.class);
   System.out.println(map.get("id"));

(d) Ignore dynamic fields in assertion
   response.then().body("status", equalTo("success"));

4. Examples:
- API returns a new "orderId" each time
   Extract it:
      String orderId = response.jsonPath().getString("orderId");
   Use it in next request payload:
      .body("{ \"orderId\": \"" + orderId + "\" }")

- Token handling
   String token = response.jsonPath().getString("token");
   Use it in header:
      .header("Authorization", "Bearer " + token)

5. Subtopics:
- API Chaining = pass dynamic response data to next API
- Ignore or mask fields like "timestamp" in validation
- Use JsonPath filters for nested dynamic arrays
- Store dynamic values in variables for reuse

Memory Trick:
Dynamic Response = Train Ticket Number 🚆
Every booking gives a new number → you don’t hardcode, you just pick it and use it.

Example:

```
public class PayLoadClass {

    RequestSpecification specification;

    @BeforeClass
    public void setUp() {
        specification = RestAssured.given()
                .contentType(ContentType.JSON)
                .baseUri("http://localhost:3000/orders"); // Mock API
    }

    @Test
    public void handleDynamicResponse() {
        // Get Response
        Response response = specification.get();

        // Convert JSON Response to POJO
        Order order = response.as(Order.class);

        // Extract dynamic values
        String orderId = order.getOrderId();
        String status = order.getStatus();
        int amount = order.getAmount();

        // Print values
        System.out.println("Dynamic Order ID: " + orderId);
```

```
        System.out.println("Status: " + status);
        System.out.println("Amount: " + amount);

        // Use dynamic value in next request (API chaining example)
        specification.basePath("/track/" + orderId)
                     .get()
                     .then()
                     .statusCode(200);

        // Ignore dynamic field (timestamp) in validation
        response.then().assertThat().body("status",
org.hamcrest.Matchers.equalTo("confirmed"));
    }
}
```

Handling Dynamic JSON Payload Using Map without POJO

```
  public class Address {

      private int houseNo;
      private String streetName;
      private String city;
      private String state;
      private String country;

   {setters and getters}

 }

     public void payLoad() throws Exception {

      ObjectMapper om= new ObjectMapper();
      Map<String,Object> address = om.readValue(
                  new File("C:\\Users\\ravgilak\\eclipse-workspace\\data.json"),
                  new TypeReference<Map<String,Object>>() {
                        });

      System.out.println(address.get("city"));
      address.put("city","Dilhi");

      System.out.println(address.get("city"));

      address.put("city", "Delhi");
      address.put("pin", "12345");
      address.remove("houseNo");

      String updated =
om.writerWithDefaultPrettyPrinter().writeValueAsString(address);
      System.out.println(updated);

    }
}
```

Handling Dynamic JSON Payload Using POJO

```
  public class Employee {

      private int id;
      private String name;
      private String country;
      private String email;
      private String role;
```

```
      private String batting_style;

      {getters and setters}
}

Payload class
-------------

 public class PayLoadClass {

      @Test
      public void payLoad() throws JsonProcessingException {

            Employee emp = new Employee();
            emp.setId(100);
            emp.setName("Surya Kumar Yadav");
            emp.setEmail("syrya.yadav@gmail.com");
            emp.setBatting_style("Right hand bating");
            emp.setRole("Batsmen");
            emp.setCountry("India");

            ObjectMapper om = new ObjectMapper();
            String data =
om.writerWithDefaultPrettyPrinter().writeValueAsString(emp);
            System.out.println(data);

      }
}

@JsonInclude(JsonInclude.Include.NON_DEFAULT)
==============================================

1) What it is:
- A Jackson annotation used when converting Java object → JSON.
- It controls which fields are included in the JSON output.

2) What it does:
- NON_DEFAULT means: do not include fields that have default values.
- Default values:
   - Numbers (int, double) → 0
   - Boolean → false
   - Object/String → null
- Only non-default values will appear in the JSON.

3) Why we use it (purpose):
- To make JSON cleaner by skipping unnecessary fields.
- To reduce payload size when sending/receiving APIs.
- To avoid confusion when default values don’t carry useful info.

Memory Trick → NON_DEFAULT = "Don’t send default values (0, false, null)".

4) How to implement:
--------------------
import com.fasterxml.jackson.annotation.JsonInclude;

@JsonInclude(JsonInclude.Include.NON_DEFAULT)
public class Employee {
    private String name;
    private int age;       // default = 0
    private boolean active; // default = false

    // getters and setters
}
```

```
5) Example in action:
---------------------

Employee emp = new Employee();
emp.setName("Ravi"); // age not set, active not set

ObjectMapper mapper = new ObjectMapper();
String json = mapper.writeValueAsString(emp);
System.out.println(json);

Output JSON:
{"name":"Ravi"}

Notice:
- age (0) and active (false) are not included.
- Only name is included because it has a non-default value.

Easy way to remember:
NON_DEFAULT = "Skip boring defaults"

@JsonInclude(JsonInclude.Include.NON_NULL)
==========================================

1) What it is:
- A Jackson annotation used when converting Java object → JSON.
- It decides which fields should be included in the JSON output.

2) What it does:
- NON_NULL means: Do not include fields that are null.
- If a field has null value, it will be skipped in the JSON.

3) Why we use it (purpose):
- To avoid sending empty/null fields in API responses.
- Makes JSON cleaner and lighter.
- Prevents confusion about whether a null means "missing" or "intentional".

Memory Trick → NON_NULL = "Skip null fields".

4) How to implement:
--------------------

import com.fasterxml.jackson.annotation.JsonInclude;

@JsonInclude(JsonInclude.Include.NON_NULL)
public class Employee {
    private String name;
    private String email;  // can be null
    private Integer age;

    // getters and setters
}


5) Example in action:
---------------------

Employee emp = new Employee();
emp.setName("Ravi"); // email and age not set (null)

ObjectMapper mapper = new ObjectMapper();
String json = mapper.writeValueAsString(emp);
System.out.println(json);

Output JSON:
```

```
{"name":"Ravi"}

Notice:
- email and age fields are skipped because they are null.
- Only name appears.

Easy way to remember:
NON_NULL = "Don’t send null values"


@JsonInclude(JsonInclude.Include.NON_EMPTY)
==========================================

1) What it is:
- A Jackson annotation used when converting Java object → JSON.
- It tells Jackson to skip fields that are "empty".

2) What it does:
- NON_EMPTY means: Do not include fields if they are empty.
- "Empty" can mean:
   - null
   - empty string ("")
   - empty collection ([], {})
   - empty map

3) Why we use it (purpose):
- To avoid sending useless empty fields in JSON.
- Makes the JSON clean and compact.
- Helps APIs avoid confusion (e.g., empty list vs missing list).

Memory Trick → NON_EMPTY = "Skip if nothing inside".

4) How to implement:
--------------------
import com.fasterxml.jackson.annotation.JsonInclude;

@JsonInclude(JsonInclude.Include.NON_EMPTY)
public class Employee {
    private String name;
    private String email;      // can be empty
    private List<String> skills;  // can be empty list

    // getters and setters
}

5) Example in action:
---------------------

Employee emp = new Employee();
emp.setName("Ravi");
emp.setEmail(""); // empty string
emp.setSkills(new ArrayList<>()); // empty list

ObjectMapper mapper = new ObjectMapper();
String json = mapper.writeValueAsString(emp);
System.out.println(json);

Output JSON:
{"name":"Ravi"}

Notice:
- email is "" → skipped
- skills is [] → skipped
- Only name appears.
```

Easy way to remember:
NON_EMPTY = "Don’t send empty/null/blank values"

INCLUDING @JsonInclude AT PROPERTY LEVEL
==========

1) What it is:
- Normally, we put @JsonInclude on top of the class (so it applies to all fields).
- But we can also put it on a single property/field.
- That means only that property will follow the include rule (NON_NULL, NON_EMPTY, etc.).

2) Why we use it:
- Sometimes you want different rules for different fields in the same class.
- Example:
   - For "email" → skip if null.
   - For "skills" → skip if empty.
   - For other fields → always include.

3) How to implement:

```
public class Employee {

    private String name;

    @JsonInclude(JsonInclude.Include.NON_NULL)
    private String email;  // will not show if null

    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private List<String> skills;  // will not show if empty

    private int age;  // always included

    // getters and setters
}
```

4) Example in action:

```
Employee emp = new Employee();
emp.setName("Ravi");
emp.setEmail(null);          // null → skipped
emp.setSkills(new ArrayList<>()); // empty list → skipped
emp.setAge(30);

ObjectMapper mapper = new ObjectMapper();
String json = mapper.writeValueAsString(emp);
System.out.println(json);
```

Output JSON:

```
{"name":"Ravi","age":30}
```

Notice:
- email (null) is skipped (because NON_NULL).
- skills (empty list) is skipped (because NON_EMPTY).
- name and age included.

5) Easy way to remember:
- Class level → rule applies to ALL fields.

- Property level → rule applies only to THAT field.

Think of it like:
"Big rule for everyone" vs. "Special rule for one person".

Customizing JSON output using Jackson annotations at class/property level
==========

```
@JsonInclude(JsonInclude.Include.NON_EMPTY)
public class Employee {

	private String name;
	private int id;
	private String city;
	private String state;
	private String country;

	private List<String> list;

}

@JsonIncludeProperties(value = {"name","skill1"})
public class EmployeeSkills extends Employee{

	private String skill1;
	public String getSkill1() {
		return skill1;
	}
	public void setSkill1(String skill1) {
		this.skill1 = skill1;
	}
	public String getSkill2() {
		return skill2;
	}
	public void setSkill2(String skill2) {
		this.skill2 = skill2;
	}
	private String skill2;
}


public class PayLoadClass {

	@Test
	public void layLoad() throws JsonProcessingException {

		EmployeeSkills emp = new EmployeeSkills();
		emp.setName("Ravindra Jadeja");
		//emp.setId(103);
		emp.setCity("Baroda");
		emp.setState("GJ");
		emp.setList(new ArrayList<String>());
		emp.setMap(new HashMap<String,Object>());
		emp.setSkill1("java");
		emp.setSkill2("selenium");


		ObjectMapper om= new ObjectMapper();
		String data =
om.writerWithDefaultPrettyPrinter().writeValueAsString(emp);
		System.out.println(data);
	}
}
```

What is the final output

Even though you set many fields, because of the annotations:
@JsonIncludeProperties → only name and skill1 are included.
@JsonInclude(NON_EMPTY) → if they were empty, they would be removed.
So the final JSON looks like this:

```
{
  "name": "Ravindra Jadeja",
  "skill1": "java"
}
```

Ignore Unknown Property from response
==========

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Employee {

      private String name;
      private int id;
      private String city;
      private String state;
      private String country;

      {getters and Setters}
}
```

```
   public void layLoad() throws JsonProcessingException {

            String data ="json data";

            ObjectMapper om= new ObjectMapper();
            Employee emp1 = om.readValue(data, Employee.class);

            System.out.println(emp1.getId());
            System.out.println(emp1.getName());
            System.out.println(emp1.getCity());
            System.out.println(emp1.getState());
            System.out.println(emp1.getCountry());

      }
}
```

Create POJO for Not valid property
==========

```
    public class Employee {

      @JsonProperty("first name")
      private String firstname;

      @JsonProperty("birth place")
      private String birthplace;

}
```

Payload class

```
public class PayLoadClass {
```

```
      @Test
      public void layLoad() throws JsonProcessingException {

            Employee emp = new Employee();
            emp.setFirstname("Ravindra jadeja");
            emp.setBirthplace("Saurashtra");

            ObjectMapper om = new ObjectMapper();

            String jsonObject =
om.writerWithDefaultPrettyPrinter().writeValueAsString(emp);

            System.out.println(jsonObject);

            Employee emp1 = om.readValue(jsonObject, Employee.class);
            System.out.println(emp1.getFirstname());
            System.out.println(emp1.getBirthplace());

      }
}


Serialization and Deserialization
=================================

1. Serialization
   - This means changing a Java object into a series of bytes (0s and 1s).
   - We do this so that the object can be stored in a file, kept in a database,
or sent
     across the internet.

   Example:
   Employee emp = new Employee("Ravi", 101);
   After serialization → it becomes raw byte data like 11010100101...

2. Deserialization
   - This is the opposite of serialization.
   - It takes the bytes and changes them back into the original Java object.
   - Example: Reading the Employee object back from a file into Java memory.

3. Serializable vs Externalizable
   - Serializable: This is a marker interface (no methods to implement). It is
the common
     way to allow Java objects to be serialized.
   - Externalizable: Gives you more control. You must write your own code to
decide how
     to save (write) and read (restore) the object.

   Example:
   class Employee implements Serializable {
      private String name;
      private int id;
   }

4. POJO (Plain Old Java Object)
   - A simple Java class that just has fields (variables), and getter/setter
methods.
   - It does not need to implement Serializable unless you want to save it as
bytes.
   - Example:
     class Student {
        private String name;
        private int rollNo;
       // getters & setters
```

```
    }

5. JSON + POJO
   - When working with JSON, a POJO can be easily converted to JSON and back.
   - POJO → JSON is called Serialization.
     Example:
     Student st = new Student("Ravi", 21);
     JSON looks like: { "name": "Ravi", "rollNo": 21 }

   - JSON → POJO is called Deserialization.
     Example:
     JSON { "name": "Ravi", "rollNo": 21 }
     becomes a Student object in Java.

6. Tools/Libraries
   - Some common libraries that help with JSON conversion are:
     Jackson, Gson, and Moshi.

Summary:
- Serialization means Java Object → Bytes or JSON.
- Deserialization means Bytes or JSON → Java Object.
- Serializable is only required for normal Java byte-based serialization.
- POJOs are simple Java classes and are often used with JSON libraries for
  easy serialization and deserialization.


Assertions in RestAssured for JSON Object vs JSON Array
=======================================================

1. JSON Object Assertion
------------------------
- In Java, a JSON Object is treated like a Map (key-value pairs).
- Use: Matchers.instanceOf(Map.class)
- Example JSON:
  {
    "id": 101,
    "name": "Ravi"
  }
- Test:
  .then().body("", Matchers.instanceOf(Map.class));

2. JSON Array Assertion
-----------------------
- In Java, a JSON Array is treated like a List.
- Use: Matchers.instanceOf(List.class)
- Example JSON:
  [
    { "id": 101, "name": "Ravi" },
    { "id": 102, "name": "Sita" }
  ]
- Test:
  .then().body("", Matchers.instanceOf(List.class));

3. Partial JSON Assertion
-------------------------
- You can assert only part of the response (a specific field).
- If the field is an Object → use Map.class
- If the field is an Array → use List.class

Examples:
---------
a) When field is Object:
{
  "name": "Ravi",
```

```
  "mobileNo": { "home": "12345", "office": "67890" }
}
Test:
.then().body("mobileNo", Matchers.instanceOf(Map.class));

b) When field is Array:
{
  "name": "Ravi",
  "mobileNo": ["12345", "67890"]
}
Test:
.then().body("mobileNo", Matchers.instanceOf(List.class));

Summary
-------
- Map.class = JSON Object
- List.class = JSON Array
- These assertions only check the response type, not actual values.

Example:
    public class Assertions {

      @Test
      public static void assertions() {

            // To verify response is a JSON Object i.e. Map
             RestAssured.given().contentType(ContentType.JSON)
            .baseUri("http://localhost:3000/data").log().all()
            .when().get()
                        .then().body("", Matchers.instanceOf(Map.class));


            // To verify response is a JSON Array i.e. List
             RestAssured.given().contentType(ContentType.JSON)
            .baseUri("http://localhost:3000/data").log().all()
            .when().get()
                        .then().body("", Matchers.instanceOf(List.class));


            // To verify part of response is a JSON Array i.e. List
             RestAssured.given().contentType(ContentType.JSON)
            .baseUri("http://localhost:3000/data").log().all()
            .when().get()
                        .then().body("mobileNo",
Matchers.instanceOf(Map.class));
      }

}

Lombok in Java
==============

1. What is Lombok?
   Lombok is a Java library that helps you avoid writing common code like
getters, setters, toString,
   constructors, etc.
   It automatically creates this code for you when you compile.

2. Main uses:
   - @Getter and @Setter → Makes getter and setter methods for variables.
   - @NoArgsConstructor → Makes an empty (no-argument) constructor.
   - @AllArgsConstructor → Makes a constructor with all fields.
   - @RequiredArgsConstructor → Makes a constructor for only final fields.
   - @ToString → Creates a toString() method that shows the variable values.
```

- @EqualsAndHashCode → Creates equals() and hashCode() methods.
- @Data → Combines Getter, Setter, ToString, EqualsAndHashCode, and RequiredArgsConstructor in one annotation.
- @Slf4j → Creates a logger object for logging messages.
- @Builder → Creates a builder pattern for making objects easily.

3. Why use Lombok?
- Saves time
- Makes code shorter and cleaner
- Reduces mistakes
- Keeps the code consistent

4. Example:
Without Lombok:

```
public class Person {
    private String name;
    private int age;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    public String toString() { return "Person{name=" + name + ", age=" + age + "}"; }
}
```

With Lombok:

```
@Data
public class Person {
    private String name;
    private int age;
}
```

@Builder in Java
==========

1. What is @Builder?
@Builder is a Lombok tool that helps make objects step-by-step using the Builder pattern.
It makes the code look cleaner and avoids problems when there are too many inputs in a constructor.

2. Why use it?
- Makes the code easy to read
- You can set only the values you need (no need to give all values in order)
- Avoids mistakes when multiple inputs have the same type
- Good for making objects that cannot be changed later (immutable)

3. How it works:
When you put @Builder on a class or constructor, Lombok creates:
- A special Builder class inside your class
- Methods to set values for each field
- A build() method that makes and returns the object

4. Example without @Builder:

```
Person p = new Person("John", 25, "New York");
```

Example with @Builder:

```
Person p = Person.builder()
                 .name("John")
                 .age(25)
                 .city("New York")
                 .build();
```

5. Where to put @Builder:
   - On a class → makes builder for all fields
   - On a constructor or method → makes builder for only those inputs

Example:

```
//@Builder
//@Getters --> we can add once install the Lombok plugin
public class Employee {

	private String firstname;
	private String birthplace;

	public String getFirstname() {
		return firstname;
	}
	public String getBirthplace() {
		return birthplace;
	}
}

public class PayLoadClass {

	@Test
	public void payLoad() {

		Employee emp = Employee.builder()
				.firstName("Ravindra")
				.birthPlace("Baroda");

		ObjectMapper om = new ObjectMapper();
		String data  =
om.writerWithDefaultPrettyPrinter().writeValueAsString(emp);
		String name  = data.getName();
		String birthPlace  = data.getBirthPlace();
	}

}
```

Retry Mechanism Using Awaitility in Rest Assured
==========

1. What is a retry mechanism?
   - Sometimes an API does not return the expected result immediately.
   - Retry mechanism means trying the request again until:
       a) We get the correct result, or
       b) The maximum wait time is reached.

2. What is Awaitility?
   - A Java library that waits until a condition is true.
   - Removes the need for writing manual loops and sleep() code.

3. How it works with Rest Assured:
   - Send an API request.
   - If the result is not as expected, Awaitility retries after a short delay.
   - Stops when the condition is met or time runs out.

4. Example:

```
   await().atMost(30, TimeUnit.SECONDS)      // wait max 30 seconds
          .pollInterval(5, TimeUnit.SECONDS) // check every 5 seconds
          .until(() ->
              get("https://api.example.com/status")
                  .then()
```

```
                .extract()
                .path("status")
                .equals("READY")
        );
```

5. Benefits:
   - Cleaner and more readable code.
   - No need for complex retry loops.
   - Easy to change wait time, intervals, and conditions.

Methods Explanation

01. given()
Signature: static RequestSpecification given()
Return type: RequestSpecification
Arguments: none
Parent: RestAssured (Class)
When to call: At the start, before sending any request.
What it does: Creates a fresh request specification.
Use: To prepare a new request setup.
Helpful when: You need a blank request to start adding headers, body, or URI.

02. when()
Signature: RequestSpecification when()
Return type: RequestSpecification
Arguments: none
Parent: RequestSpecification (Interface)
When to call: After preparing request details.
What it does: Marks the point where the request will be sent.
Use: To begin calling GET, POST, PUT, DELETE, etc.
Helpful when: You are ready to actually send the request.

03. then()
Signature: ValidatableResponse then()
Return type: ValidatableResponse
Arguments: none
Parent: Response (Class)
When to call: After the request has been sent and response is received.
What it does: Starts validation of the response.
Use: To perform checks on status code, body, headers, etc.
Helpful when: You want to assert or validate server response.

04. get()
Signature: Response get()
Overload: Response get(String path)
Return type: Response
Arguments: none OR String path
Parent: RequestSpecification (Interface)
When to call: After when().
What it does: Sends a GET request to server.
Use: To retrieve data.
Helpful when: You want to fetch resources from an endpoint.

05. post()
Signature: Response post()
Overload: Response post(String path)
Return type: Response
Arguments: none OR String path
Parent: RequestSpecification (Interface)
When to call: After when(), usually after setting body.

What it does: Sends a POST request.
Use: To create a resource on the server.
Helpful when: You want to add or create new data.

06. put()
Signature: Response put()
Overload: Response put(String path)
Return type: Response
Arguments: none OR String path
Parent: RequestSpecification (Interface)
When to call: After when(), with body.
What it does: Sends a PUT request.
Use: To replace an existing resource completely.
Helpful when: You want to update full record of data.

07. delete()
Signature: Response delete()
Overload: Response delete(String path)
Return type: Response
Arguments: none OR String path
Parent: RequestSpecification (Interface)
When to call: After when().
What it does: Sends a DELETE request.
Use: To remove a resource from server.
Helpful when: You want to delete record by id/path.

08. baseUri()
Signature: RequestSpecification baseUri(String uri)
Return type: RequestSpecification
Arguments: String uri
Parent: RequestSpecification (Interface)
When to call: While building the request.
What it does: Sets the main server address.
Use: To define base server endpoint.
Helpful when: All requests go to same domain.

09. header()
Signature: RequestSpecification header(String name, Object value)
Overload: RequestSpecification header(Map<String, ?> headers)
Return type: RequestSpecification
Arguments: key-value OR Map
Parent: RequestSpecification (Interface)
When to call: During request setup.
What it does: Adds headers to the request.
Use: To pass authentication token, content-type, etc.
Helpful when: API requires extra information in headers.

10. body()
Signature: RequestSpecification body(Object body)
Overload: RequestSpecification body(String body)
Return type: RequestSpecification
Arguments: Object or String
Parent: RequestSpecification (Interface)
When to call: Before sending request.
What it does: Attaches data payload to request.
Use: To send JSON, XML, form data.
Helpful when: You need to send input data to server.

11. log()
Signature: RequestSpecification log()
Return type: RequestSpecification
Arguments: none
Parent: RequestSpecification (Interface)
When to call: Before sending request.

What it does: Enables logging.
Use: To display request/response details.
Helpful when: You want to debug or see what is being sent/received.

12. all()
Signature: RequestSpecification all()
Return type: RequestSpecification
Arguments: none
Parent: LogSpecification (Interface)
When to call: After log().
What it does: Logs all details of request or response.
Use: To show full log (headers, body, params).
Helpful when: You want complete visibility for debugging.

13. statusCode()
Signature: ValidatableResponse statusCode(int expectedStatusCode)
Return type: ValidatableResponse
Arguments: int expectedStatusCode
Parent: ValidatableResponse (Interface)
When to call: After then().
What it does: Checks if response code matches expected.
Use: To validate server returned correct HTTP status.
Helpful when: You want to confirm request was successful or failed correctly.

14. basePath()
Signature: RequestSpecification basePath(String path)
Return type: RequestSpecification
Arguments: String path
Parent: RequestSpecification (Interface)
When to call: While building request.
What it does: Sets common resource path.
Use: To avoid writing full URL every time.
Helpful when: All endpoints share same path prefix.

15. pathParam()
Signature: RequestSpecification pathParam(String name, Object value)
Return type: RequestSpecification
Arguments: String name, Object value
Parent: RequestSpecification (Interface)
When to call: During request setup.
What it does: Replaces placeholder in URL with actual value.
Use: To set dynamic values in path.
Helpful when: Endpoint requires id or dynamic value.

16. contentType()
Signature: RequestSpecification contentType(String type)
Return type: RequestSpecification
Arguments: String type
Parent: RequestSpecification (Interface)
When to call: Before sending request.
What it does: Sets request body type.
Use: To tell server whether data is JSON, XML, etc.
Helpful when: API requires specific content type.

17. extract()
Signature: ExtractableResponse<Response> extract()
Return type: ExtractableResponse<Response>
Arguments: none
Parent: ValidatableResponse (Interface)
When to call: After then().
What it does: Gets response object from validations.
Use: To reuse response data for later steps.
Helpful when: You want to save response details.

18. response()
Signature: Response response()
Return type: Response
Arguments: none
Parent: ExtractableResponse (Interface)
When to call: After extract().
What it does: Returns the raw response object.
Use: To access headers, body, cookies.
Helpful when: You need raw data from server.

19. asString()
Signature: String asString()
Return type: String
Arguments: none
Parent: Response (Class)
When to call: After getting response.
What it does: Converts body to plain string.
Use: To read raw response text.
Helpful when: You want unformatted plain response.

20. asPrettyString()
Signature: String asPrettyString()
Return type: String
Arguments: none
Parent: Response (Class)
When to call: After getting response.
What it does: Converts body to pretty formatted string.
Use: To make JSON or XML easy to read.
Helpful when: You want response in readable format.

21. time()
Signature: long time()
Return type: long
Arguments: none
Parent: Response (Class)
When to call: After response is received.
What it does: Returns response time in milliseconds.
Use: To check how long request took.
Helpful when: You are measuring API speed.

22. timeIn()
Signature: long timeIn(TimeUnit unit)
Return type: long
Arguments: TimeUnit unit
Parent: Response (Class)
When to call: After response is received.
What it does: Returns response time in chosen unit.
Use: To check performance in seconds, microseconds, etc.
Helpful when: You need flexible unit for performance test.

23. getTime()
Signature: long getTime()
Return type: long
Arguments: none
Parent: Response (Class)
When to call: After response is received.
What it does: Same as time(), gives response time in ms.
Use: To know how fast server responded.
Helpful when: You check performance with default ms.

24. getTimeIn()
Signature: long getTimeIn(TimeUnit unit)
Return type: long
Arguments: TimeUnit unit

Parent: Response (Class)
When to call: After response is received.
What it does: Same as timeIn(), gives response time in unit you choose.
Use: To measure API performance with specific unit.
Helpful when: You want time in seconds, minutes, etc.

25. headers()
Signature: Headers headers()
Return type: Headers
Arguments: none
Parent: Response (Class)
When to call: After response is received.
What it does: Returns all response headers.
Use: To check metadata (content-type, server, cache info).
Helpful when: You need to validate or debug response headers.

26. Method: jsonPath
Syntax: public JsonPath jsonPath()
Return Type: JsonPath
Arguments: None
Parent: Response
Parent Type: Class
When to Call: After receiving the API response
Use: To convert the response body into a JSON structure for easy data extraction
What It Does: Parses the response body and gives a JsonPath object to extract values
Helpful Situation: When you want to extract fields like id, name, or token from a JSON response

27. Method: getInt
Syntax: public int getInt(String path)
Return Type: int
Arguments: String path (JSON path expression)
Parent: JsonPath
Parent Type: Class
When to Call: After calling jsonPath() on a response
Use: To extract an integer value from the JSON response
What It Does: Reads the value at the given path and returns it as an integer
Helpful Situation: When you want to extract numeric fields like userId, age, or count

28. Method: getString
Syntax: public String getString(String path)
Return Type: String
Arguments: String path (JSON path expression)
Parent: JsonPath
Parent Type: Class
When to Call: After calling jsonPath() on a response
Use: To extract a text value from the JSON response
What It Does: Reads the value at the given path and returns it as a string
Helpful Situation: When you want to extract fields like name, email, or token

29. Method: getList
Syntax: public <T> List<T> getList(String path)
Return Type: List<T>
Arguments: String path (JSON path expression)
Parent: JsonPath
Parent Type: Class
When to Call: After calling jsonPath() on a response
Use: To extract a list of values from a JSON array
What It Does: Reads the array at the given path and returns it as a Java list
Helpful Situation: When you want to extract multiple items like a list of users, products, or bookings

30. Method: findAll
Syntax: public List<?> findAll(Closure<Boolean> condition)
Return Type: List<?>
Arguments: Closure<Boolean> condition (Groovy expression)
Parent: List (Groovy-enhanced)
Parent Type: Interface
When to Call: Inside a JSON path expression when filtering a list
Use: To get all items from a list that match a condition
What It Does: Filters the list and returns only matching elements
Helpful Situation: When you want to extract users with age > 30 or products with price < 100

41. Method: public Set<K> keySet()
Return Type: Set<K>
Arguments: None
Parent: Map
Parent Type: Interface
When to Call: After you have a Map object
Use: To get all the keys from a Map
What It Does: Returns a set containing all the keys present in the map
Helpful Situation: When you want to loop through or validate the keys in a JSON object or Java Map

42. Keyword: instanceof
Syntax: object instanceof ClassName
Return Type: boolean
Arguments: One object and one class type
Parent: Java language keyword
Parent Type: Built-in
When to Call: During conditional checks
Use: To check if an object belongs to a specific class or interface
What It Does: Returns true if the object is of the specified type
Helpful Situation: When you want to confirm the type of a response or variable before casting or processing

43. Method: public String writeValueAsString(Object value)
Return Type: String
Arguments: Object value
Parent: ObjectMapper
Parent Type: Class
When to Call: When converting Java objects to JSON
Use: To serialize a Java object into a JSON string
What It Does: Converts the given object into a JSON-formatted string
Helpful Situation: When you need to send a Java object as a JSON payload in an API request

44. Method: public <T> T readValue(String content, Class<T> valueType)
Return Type: T (generic type)
Arguments: String content, Class<T> valueType
Parent: ObjectMapper
Parent Type: Class
When to Call: When converting JSON string to Java object
Use: To deserialize a JSON string into a Java object
What It Does: Parses the JSON string and returns a Java object of the given type
Helpful Situation: When you receive a JSON response and want to convert it into a Java class for further use

45. Method: public <T> T getObject(String path, Class<T> type)
Return Type: T (generic type)
Arguments: String path, Class<T> type
Parent: JsonPath
Parent Type: Class
When to Call: After calling jsonPath() on a response
Use: To extract a complex object from the JSON response

What It Does: Reads the value at the given path and converts it into the specified Java class
Helpful Situation: When you want to extract nested objects like a user profile, address, or order details

45. Method: public <T> T getObject(String path, Class<T> type)
Return Type: T (generic type)
Arguments: String path, Class<T> type
Parent: JsonPath
Parent Type: Class
When to Call: After calling jsonPath() on a response
Use: To extract a complex object from the JSON response
What It Does: Reads the value at the given path and converts it into the specified Java class
Helpful Situation: When you want to extract nested objects like a user profile, address, or order details

46. Method: public static <T> Matcher<T> equalTo(T operand)
Return Type: Matcher<T>
Arguments: T operand (expected value)
Parent: Matchers (Hamcrest)
Parent Type: Class
When to Call: Inside then().body() or with assertThat() for validation
Use: To check if a value is exactly equal to the expected value
What It Does: Compares actual and expected values for equality
Helpful Situation: When you want to validate that a response field matches an exact expected value

47. Method: public static <T> void assertThat(T actual, Matcher<? super T> matcher)
Return Type: void
Arguments: T actual, Matcher<? super T> matcher
Parent: Assert (Hamcrest or JUnit)
Parent Type: Class
When to Call: During test assertions
Use: To assert that a value meets a condition defined by a matcher
What It Does: Throws an error if the actual value does not match the expected condition
Helpful Situation: When writing test cases to validate API responses or logic

48. Method: public ObjectWriter writerWithDefaultPrettyPrinter()
Return Type: ObjectWriter
Arguments: None
Parent: ObjectMapper
Parent Type: Class
When to Call: Before writing JSON output
Use: To get a writer that formats JSON with indentation and line breaks
What It Does: Returns a writer that produces human-readable (pretty) JSON
Helpful Situation: When you want to log or print JSON in a readable format for debugging or documentation

49. Method: public V put(K key, V value)
Return Type: V
Arguments: K key, V value
Parent: Map
Parent Type: Interface
When to Call: When adding or updating entries in a Map
Use: To insert a new key-value pair or update an existing one
What It Does: Adds the key-value pair to the map and returns the previous value (if any)
Helpful Situation: When building request payloads or storing extracted data in a Map

50. Method: public V remove(Object key)

Return Type: V
Arguments: Object key
Parent: Map
Parent Type: Interface
When to Call: When you want to delete an entry from a Map
Use: To remove a key-value pair from the map
What It Does: Deletes the entry with the given key and returns its value
Helpful Situation: When cleaning up data or modifying a payload before sending it

51. Method: public static Matcher<Object> instanceOf(Class<?> type)
Return Type: Matcher<Object>
Arguments: Class<?> type
Parent: Matchers (Hamcrest)
Parent Type: Class
When to Call: Inside assertThat() or then().body() for type validation
Use: To check if an object is an instance of a specific class
What It Does: Validates that the actual value is of the expected type
Helpful Situation: When you want to confirm that a response field is of a certain type like String, Integer, etc.

52. Method: public static FluentWait<Object> builder()
Return Type: FluentWait<Object>
Arguments: None
Parent: Awaitility (or custom builder class depending on context)
Parent Type: Class
When to Call: When starting a wait configuration
Use: To begin building a wait condition with custom settings
What It Does: Returns a builder object to configure wait time, polling interval, and condition
Helpful Situation: When you want to wait for a condition to be true in asynchronous testing or delayed responses

53. Method: public FluentWait<Object> build()
Return Type: FluentWait<Object>
Arguments: None
Parent: Builder class (used in Awaitility or custom wait logic)
Parent Type: Class
When to Call: After configuring the builder with desired settings
Use: To finalize and create the wait object
What It Does: Builds and returns the configured wait instance
Helpful Situation: When you want to apply the wait logic to a condition in your test

54. Method: public ConditionFactory await()
Return Type: ConditionFactory
Arguments: None
Parent: Awaitility
Parent Type: Class
When to Call: At the start of a wait condition
Use: To begin defining a wait condition
What It Does: Returns a factory to configure how long to wait and what to wait for
Helpful Situation: When testing asynchronous systems or waiting for a delayed response

55. Method: public ConditionFactory atMost(Duration timeout)
Return Type: ConditionFactory
Arguments: Duration timeout
Parent: ConditionFactory (Awaitility)
Parent Type: Class
When to Call: After calling await()
Use: To set the maximum time to wait for a condition
What It Does: Limits the wait time to the specified duration

Helpful Situation: When you want to avoid infinite waits and fail the test if the condition is not met in time

56. Method: public ConditionFactory pollInterval(Duration interval)
Return Type: ConditionFactory
Arguments: Duration interval
Parent: ConditionFactory (Awaitility)
Parent Type: Class
When to Call: After calling await()
Use: To set how often the condition should be checked
What It Does: Configures the polling frequency during the wait
Helpful Situation: When you want to control how frequently the system checks for the condition to be true

57. Method: public void until(Callable<Boolean> condition)
Return Type: void
Arguments: Callable<Boolean> condition
Parent: ConditionFactory (Awaitility)
Parent Type: Class
When to Call: At the end of the wait configuration
Use: To define the condition that must become true
What It Does: Waits until the given condition returns true or the timeout is reached
Helpful Situation: When you want to wait for a specific event, response, or state change in your test

Classes Explanation

01. Class: Response
Type: Class
Package: io.restassured.response
Use: Represents the response returned by an API call
What It Does: Provides methods to access status code, headers, body, and JSON data
Helpful Situation: When you want to validate or extract data from an API response

02. Class: RestAssured
Type: Class
Package: io.restassured.RestAssured
Use: Main entry point for Rest Assured API testing
What It Does: Allows setting base URI, authentication, and sending requests
Helpful Situation: When you want to configure and send HTTP requests in tests

03. Class: Header
Type: Class
Package: io.restassured.http
Use: Represents a single HTTP header
What It Does: Stores a name-value pair for a header
Helpful Situation: When you want to add or validate specific headers in a request or response

04. Class: Headers
Type: Class
Package: io.restassured.http
Use: Represents a collection of HTTP headers
What It Does: Holds multiple Header objects together
Helpful Situation: When you want to manage or validate multiple headers at once

05. Class: JsonPath
Type: Class

Package: io.restassured.path.json
Use: Allows extracting values from JSON using path expressions
What It Does: Parses JSON and provides methods like getInt(), getString(), getList()
Helpful Situation: When you want to extract specific fields from a JSON response

06. Class: TypeRef<T>
Type: Class
Package: io.restassured.common.mapper
Use: Represents a generic type reference for deserialization
What It Does: Helps convert JSON into complex types like List<Map<String, Object>>
Helpful Situation: When you want to deserialize a JSON array or nested object

07. Class: ObjectMapper
Type: Class
Package: com.fasterxml.jackson.databind
Use: Converts between Java objects and JSON
What It Does: Provides methods like writeValueAsString() and readValue()
Helpful Situation: When you want to serialize or deserialize JSON in your tests

08. Annotation: BeforeClass
Type: Annotation
Package: org.junit.BeforeClass
Use: Marks a method to run once before all tests in the class
What It Does: Initializes setup logic before test execution
Helpful Situation: When you want to set base URI or load test data before running tests

09. Class: File
Type: Class
Package: java.io
Use: Represents a file or directory path
What It Does: Allows reading, writing, and checking file existence
Helpful Situation: When you want to load JSON payloads or schemas from external files

10. Class: TypeReference<T>
Type: Class
Package: com.fasterxml.jackson.core.type
Use: Represents a generic type for deserialization
What It Does: Helps ObjectMapper convert JSON into complex types
Helpful Situation: When you want to deserialize JSON into List<User>, Map<String,Object>, etc.

11. Class: JsonProcessingException
Type: Class
Package: com.fasterxml.jackson.core
Use: Exception thrown during JSON parsing or generation
What It Does: Indicates a problem while converting JSON
Helpful Situation: When handling errors during serialization or deserialization

12. Annotation: JsonIncludeProperties
Type: Annotation
Package: com.fasterxml.jackson.annotation
Use: Specifies which properties to include during serialization
What It Does: Filters fields in the output JSON
Helpful Situation: When you want to include only selected fields in the JSON response

13. Enum: TimeUnit
Type: Enum
Package: java.util.concurrent
Use: Represents time units like SECONDS, MILLISECONDS

What It Does: Used for time conversions and delays
Helpful Situation: When setting wait times, delays, or timeouts in tests

14. Class: Matchers
Type: Class
Package: org.hamcrest
Use: Provides matcher methods like equalTo(), notNullValue(), hasItems()
What It Does: Helps validate values in assertions
Helpful Situation: When writing test validations for response fields

15. Class: ArrayList
Type: Class
Package: java.util
Use: A resizable array implementation of List
What It Does: Stores ordered elements and allows dynamic resizing
Helpful Situation: When collecting multiple values from a response or building payloads

16. Class: MatcherAssert
Type: Class
Package: org.hamcrest
Use: Provides assertThat() method for validations
What It Does: Asserts that a value matches a condition
Helpful Situation: When writing test assertions with matchers

17. Class: JsonSchemaValidator
Type: Class
Package: io.restassured.module.jsv
Use: Validates JSON response against a schema
What It Does: Provides method matchesJsonSchemaInClasspath()
Helpful Situation: When you want to ensure the response structure is correct

18. Class: HashMap
Type: Class
Package: java.util
Use: Stores key-value pairs with no guaranteed order
What It Does: Allows fast access and updates using keys
Helpful Situation: When building request payloads or storing extracted data

19. Class: LinkedHashMap
Type: Class
Package: java.util
Use: Stores key-value pairs with insertion order preserved
What It Does: Maintains the order in which entries were added
Helpful Situation: When you need predictable order in your payload or response data

20. Class: LinkedList
Type: Class
Package: java.util
Use: Doubly-linked list implementation of List
What It Does: Allows efficient insertions and deletions
Helpful Situation: When managing ordered data with frequent updates

21. Class: Awaitility
Type: Class
Package: org.awaitility
Use: Provides fluent API for waiting on asynchronous conditions
What It Does: Allows configuration of wait time, polling, and condition
Helpful Situation: When testing delayed responses or background processes

Interfaces
==========

01. Interface: RequestSpecification
Package: io.restassured.specification
Use: Defines how to build and configure an HTTP request
What It Does: Allows setting headers, query params, body, authentication, etc.
Helpful Situation: When you want to prepare a reusable request setup before sending it

02. Interface: ValidatableResponse
Package: io.restassured.response
Use: Defines methods for validating an API response
What It Does: Provides fluent methods like statusCode(), body(), header() for assertions
Helpful Situation: When you want to validate the response after sending a request

03. Interface: ExtractableResponse<T>
Package: io.restassured.response
Use: Defines methods to extract data from a response
What It Does: Allows converting response into JsonPath, String, or custom objects
Helpful Situation: When you want to extract values from the response for further use

04. Interface: List<E>
Package: java.util
Use: Represents an ordered collection of elements
What It Does: Allows storing, accessing, and modifying a sequence of items
Helpful Situation: When working with arrays or multiple values from a JSON response

05. Interface: Serializable
Package: java.io
Use: Marks a class as capable of being serialized
What It Does: Enables converting an object into a byte stream for storage or transfer
Helpful Situation: When saving objects to files or sending them over a network

06. Interface: Externalizable
Package: java.io
Use: Extends Serializable with custom control over serialization
What It Does: Requires implementing writeExternal() and readExternal() methods
Helpful Situation: When you need full control over how object data is saved and restored

Annotations

01. Annotation: @JsonInclude
Package: com.fasterxml.jackson.annotation
Use: Controls which fields are included during JSON serialization
What It Does: Skips null or empty fields based on configuration
Helpful Situation: When you want to avoid sending unnecessary fields in API payloads

02. Annotation: @JsonIncludeProperties
Package: com.fasterxml.jackson.annotation
Use: Specifies which properties to include during serialization
What It Does: Filters the output to include only selected fields
Helpful Situation: When you want to expose only specific fields in the JSON response

03. Annotation: @JsonProperty
Package: com.fasterxml.jackson.annotation

Use: Maps a Java field to a specific JSON property name
What It Does: Allows renaming fields or handling mismatched names
Helpful Situation: When the JSON field name is different from the Java variable name

04. Annotation: @JsonIgnoreProperties
Package: com.fasterxml.jackson.annotation
Use: Ignores specified fields during serialization or deserialization
What It Does: Skips unwanted fields when converting between JSON and Java
Helpful Situation: When the JSON contains extra fields you don’t want to process

05. Annotation: @Data
Package: lombok
Use: Generates getters, setters, toString, equals, hashCode, and constructors
What It Does: Reduces boilerplate code for POJOs
Helpful Situation: When creating model classes for request/response payloads

06. Annotation: @Getter
Package: lombok
Use: Generates getter methods for all fields
What It Does: Adds public get methods automatically
Helpful Situation: When you want read access to private fields without writing code manually

07. Annotation: @Setter
Package: lombok
Use: Generates setter methods for all fields
What It Does: Adds public set methods automatically
Helpful Situation: When you want to update field values without writing boilerplate code

08. Annotation: @NoArgsConstructor
Package: lombok
Use: Generates a no-argument constructor
What It Does: Adds a default constructor with no parameters
Helpful Situation: When frameworks like Jackson need a default constructor for deserialization

09. Annotation: @AllArgsConstructor
Package: lombok
Use: Generates a constructor with all fields as parameters
What It Does: Adds a constructor that sets all fields
Helpful Situation: When you want to quickly create objects with all values set

10. Annotation: @RequiredArgsConstructor
Package: lombok
Use: Generates a constructor for final and non-null fields only
What It Does: Adds a constructor with required fields
Helpful Situation: When you want to enforce initialization of important fields

11. Annotation: @ToString
Package: lombok
Use: Generates a toString() method
What It Does: Returns a string representation of the object
Helpful Situation: When you want to log or print object details easily

12. Annotation: @EqualsAndHashCode
Package: lombok
Use: Generates equals() and hashCode() methods
What It Does: Helps compare objects and use them in collections
Helpful Situation: When you want to compare objects or store them in sets/maps

13. Annotation: @Slf4j
Package: lombok.extern.slf4j

Use: Adds a logger instance named 'log' to the class
What It Does: Enables logging using SLF4J
Helpful Situation: When you want to log messages for debugging or tracking

14. Annotation: @Builder
Package: lombok
Use: Enables builder pattern for object creation
What It Does: Allows creating objects step-by-step using chained methods
Helpful Situation: When you want to build complex objects in a readable way