REST Assured API Automation – Setup, Validation, Authentication & Framework Design

A comprehensive instructor-led training programme designed for QA engineers, automation testers, and manual testers transitioning to API automation. This course will equip you with the practical skills needed to design, implement, and maintain robust REST Assured automation frameworks used in real-world projects.

Why Automate APIs?

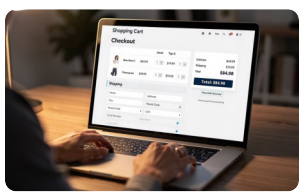
The Business Case

API automation delivers faster feedback, reduces testing time, and catches backend issues before they reach the UI layer. APIs are the backbone of modern applications, and testing them early in the development cycle is crucial for quality assurance.

Key Benefits

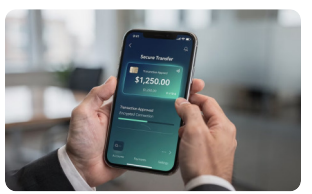
- Faster execution compared to UI tests
- Early defect detection in the development cycle
- Reduced maintenance effort
- Better test coverage with fewer resources
- Platform-independent validation
- Ideal for CI/CD pipeline integration

Real-World API Automation Use Cases



E-Commerce Platforms

Validating product catalogues, shopping cart operations, payment gateway integrations, and order processing workflows across multiple services.

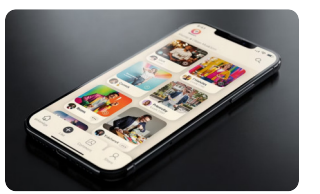


Banking & Finance

Testing fund transfers, account balance checks, transaction histories, and third-party payment integrations with stringent security requirements.

Healthcare Systems

Validating patient record management, appointment scheduling, prescription systems, and integration with insurance providers and diagnostic labs.



Social Media

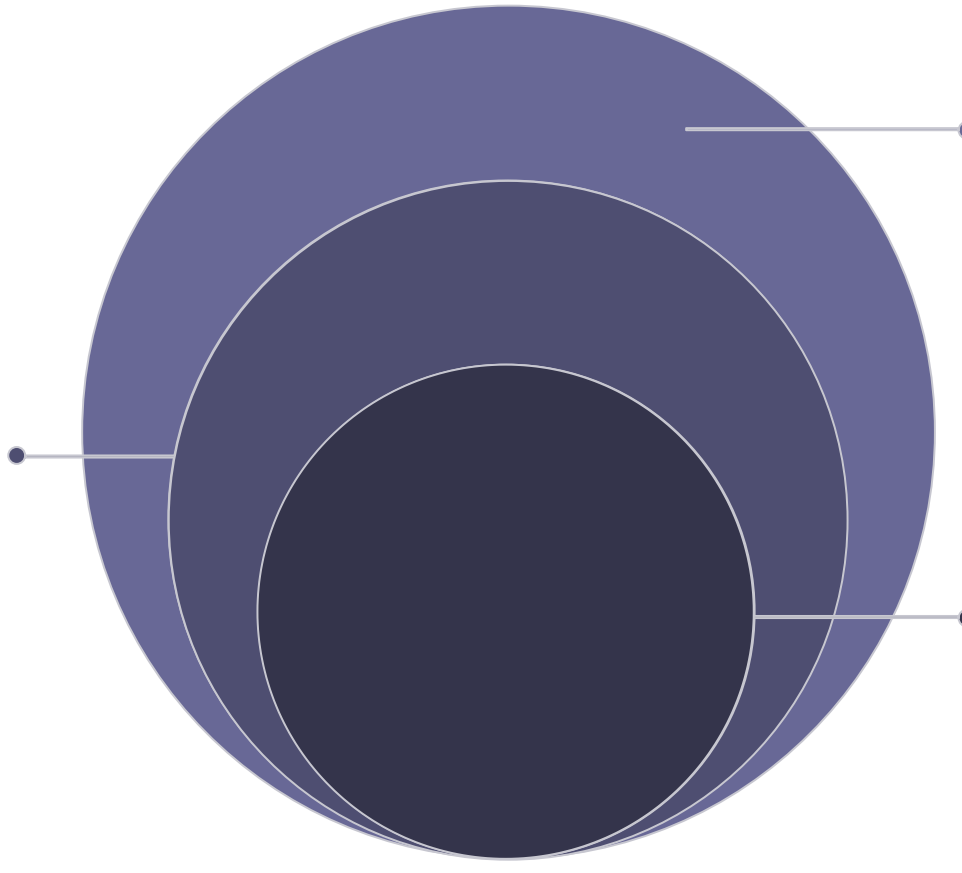
Testing user authentication, content posting, feed generation, notification systems, and real-time messaging across distributed microservices.

API Automation vs UI Automation

Aspect	API Automation	UI Automation
Speed	Very fast (milliseconds)	Slower (seconds to minutes)
Maintenance	Low – APIs change less frequently	High – UI changes often
Coverage	Focuses on business logic	Focuses on user experience
Reliability	Highly stable and consistent	Prone to flakiness
CI/CD Integration	Seamless and efficient	Requires more resources
Early Testing	Can start before UI exists	Requires complete UI
Dependencies	Minimal external dependencies	Browser, network, rendering

API automation complements UI automation by providing comprehensive test coverage across different layers of the application architecture.

Where REST Assured Fits in Test Automation Strategy



REST Assured occupies the critical middle layer of the testing pyramid. It provides comprehensive backend validation whilst maintaining execution speed and reliability. By automating API tests with REST Assured, teams can achieve broader coverage faster than UI-only approaches, making it essential for modern continuous delivery pipelines.

Test Your Knowledge: API Automation Fundamentals

Question 1

Which layer of the testing pyramid should contain the most number of automated tests?

- A) UI Tests
- B) API Tests
- C) Unit Tests
- D) Integration Tests

Correct Answer: C

Question 2

What is the primary advantage of API automation over UI automation?

- A) Better graphics rendering
- B) Faster execution and reliability
- C) Easier to learn
- D) No coding required

Correct Answer: B

Interview Keywords: API Automation

Backend Testing

Testing the server-side logic, databases, and APIs that power applications without involving the user interface.

API Automation

Automated validation of API endpoints, request-response cycles, data formats, and business logic using testing frameworks.

Testing Pyramid

A strategy that emphasises more unit tests, fewer API tests, and minimal UI tests for optimal coverage and efficiency.

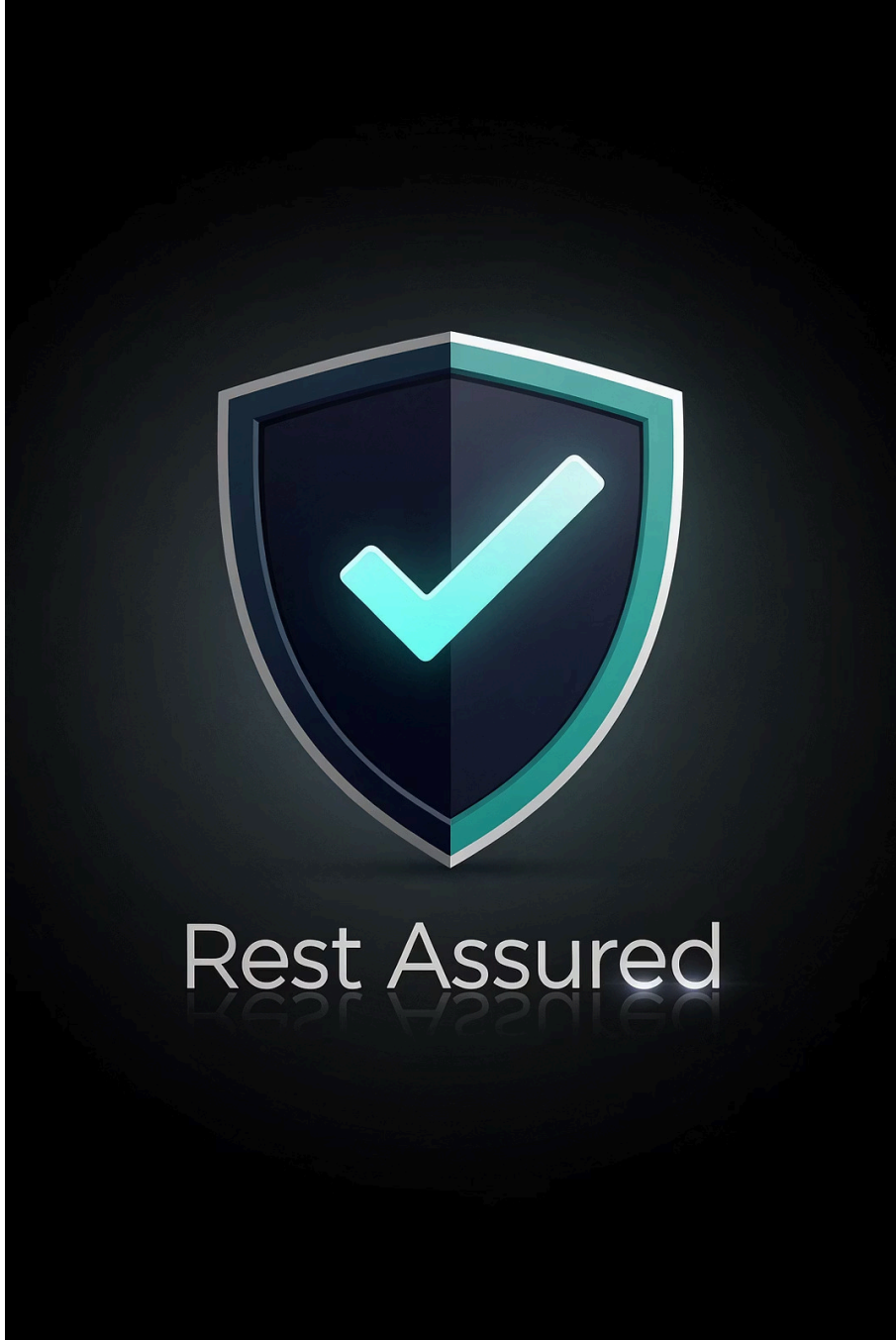
CI/CD Pipeline

Continuous integration and deployment workflows where automated API tests provide rapid feedback on code changes.

What is REST Assured?

REST Assured is a Java-based domain-specific language (DSL) designed specifically for testing RESTful web services. It simplifies API testing by providing an intuitive, readable syntax that mirrors natural language, making it accessible for both developers and testers.

Unlike general-purpose HTTP libraries, REST Assured is purpose-built for validation. It integrates seamlessly with testing frameworks like TestNG and JUnit, supports various authentication mechanisms, and provides powerful assertion capabilities for JSON and XML responses. Its declarative approach allows testers to focus on what to validate rather than how to implement the validation logic.



Key Features of REST Assured

BDD-Style Syntax

Given-When-Then format makes tests readable and maintainable, even for non-programmers.

Built-in Validation

Comprehensive assertion methods for status codes, headers, response body, and response time.

JSON & XML Support

Native parsing and validation capabilities for both JSON and XML response formats.

Authentication Support

Out-of-the-box support for Basic, OAuth, Bearer tokens, and API key authentication.

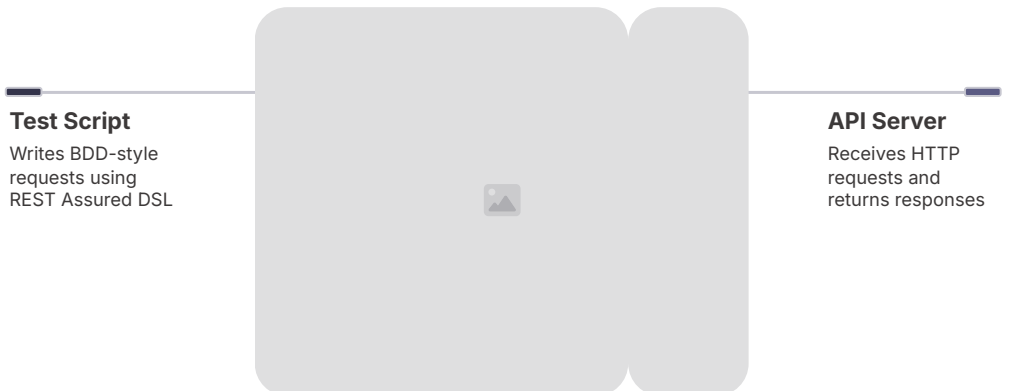
File Upload Handling

Simple methods to handle multipart form data and file uploads in API requests.

Framework Integration

Works seamlessly with TestNG, JUnit, Cucumber, and CI/CD tools like Jenkins.

REST Assured Architecture



How It Works

REST Assured acts as an abstraction layer between your test scripts and HTTP communication. When you write a test, REST Assured translates your BDD-style code into HTTP requests, sends them to the target API, receives responses, and provides convenient methods to validate them.

Under the hood, it uses HTTP Client libraries but shields you from low-level implementation details, allowing you to focus purely on test logic and validation.

REST Assured vs Postman vs SoapUI

Feature	REST Assured	Postman	SoapUI
Type	Code-based library	GUI tool	GUI + code tool
Language	Java	JavaScript	Groovy
CI/CD Integration	Excellent	Good (via Newman)	Moderate
Learning Curve	Moderate	Easy	Steep
Version Control	Easy (code files)	Requires export	Requires export
Collaboration	Excellent (Git)	Good (Workspaces)	Limited
Debugging	IDE debugger	Built-in console	Built-in debugger
Best For	Automation frameworks	Manual + automation	SOAP + REST

Test Your Knowledge: REST Assured Basics

Question 1

Which programming language is REST Assured primarily used with?

- A) Python
- B) JavaScript
- C) Java
- D) C#

Correct Answer: C

Question 2

What syntax style does REST Assured follow?

- A) Object-Oriented
- B) Functional
- C) BDD (Given-When-Then)
- D) Procedural

Correct Answer: C

REST Assured Setup: Prerequisites

Before beginning REST Assured automation, ensure your development environment is properly configured with the following essential components:

01

Java Development Kit (JDK)

JDK 8 or higher installed and JAVA_HOME environment variable configured. Verify installation using: java -version

02

Apache Maven

Maven 3.6 or higher for dependency management and build automation. Verify installation using: mvn -version

03

Integrated Development Environment

IntelliJ IDEA, Eclipse, or any Java IDE with Maven support for efficient code development and debugging.

04

Testing Framework

TestNG or JUnit knowledge for organising and executing test cases with proper annotations and assertions.

Adding REST Assured Dependency in pom.xml

Maven simplifies dependency management by automatically downloading required libraries. Add the following dependency to your pom.xml file within the dependencies section:

```
<dependencies>
    <!-- REST Assured for API Testing -->
    <dependency>
        <groupId>io.rest-assured</groupId>
        <artifactId>rest-assured</artifactId>
        <version>5.3.0</version>
        <scope>test</scope>
    </dependency>

    <!-- TestNG for Test Execution -->
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.7.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

After adding dependencies, run `mvn clean install` to download libraries to your local repository.

Maven Dependency Management Explained

Understanding Maven Coordinates

- **groupId:** Organisation or project identifier (e.g., io.rest-assured)
- **artifactId:** Specific library name (e.g., rest-assured)
- **version:** Library version number (e.g., 5.3.0)
- **scope:** When the dependency is needed (test, compile, runtime)

Version Compatibility Matters

Always verify compatibility between REST Assured, Java version, and testing frameworks. Using incompatible versions can lead to runtime errors or missing features. Check the official REST Assured documentation for version compatibility matrices before upgrading dependencies.

REST Assured Project Structure

```
api-automation-project/
 │
 ├─── src/
 │    ├─── main/
 │    │    └─── java/
 │    │         └─── utils/
 │    │              ├─── ConfigReader.java
 │    │              └─── JsonUtils.java
 │    │
 │    └─── test/
 │         ├─── java/
 │         │    ├─── base/
 │         │    │    └─── BaseTest.java
 │         │    ├─── tests/
 │         │    │    ├─── UserAPITests.java
 │         │    │    └─── OrderAPITests.java
 │         │    └─── builders/
 │         │         └─── RequestBuilder.java
 │         │
 │         └─── resources/
 │              ├─── testdata/
 │              │    └─── user-data.json
 │              └─── config.properties
 │
 ├─── pom.xml
 └─── testng.xml
```

This modular structure separates concerns: utilities for reusable code, base classes for common setup, test classes for actual test cases, and resources for test data and configuration.

Common REST Assured Setup Issues

Dependency Download Failures

Issue: Maven cannot download dependencies due to network or proxy issues.

Solution: Configure Maven proxy settings in settings.xml or use a local repository manager like Nexus.

Version Conflicts

Issue: Multiple versions of the same library causing conflicts (e.g., different JSON parsers).

Solution: Use `mvn dependency:tree` to identify conflicts and exclude transitive dependencies explicitly.

JAVA_HOME Not Set

Issue: Maven or IDE cannot find Java installation.

Solution: Set JAVA_HOME environment variable pointing to JDK installation directory and add to PATH.

IDE Not Recognising Dependencies

Issue: Dependencies added but IDE shows import errors.

Solution: Refresh Maven project (right-click project → Maven → Reload Project) or reimport the project.

Test Your Knowledge: REST Assured Setup

Question 1

Which file is used to manage dependencies in a Maven project?

- A) build.gradle
- B) pom.xml
- C) package.json
- D) settings.xml

Correct Answer: B

Question 2

What is the minimum Java version required for REST Assured 5.x?

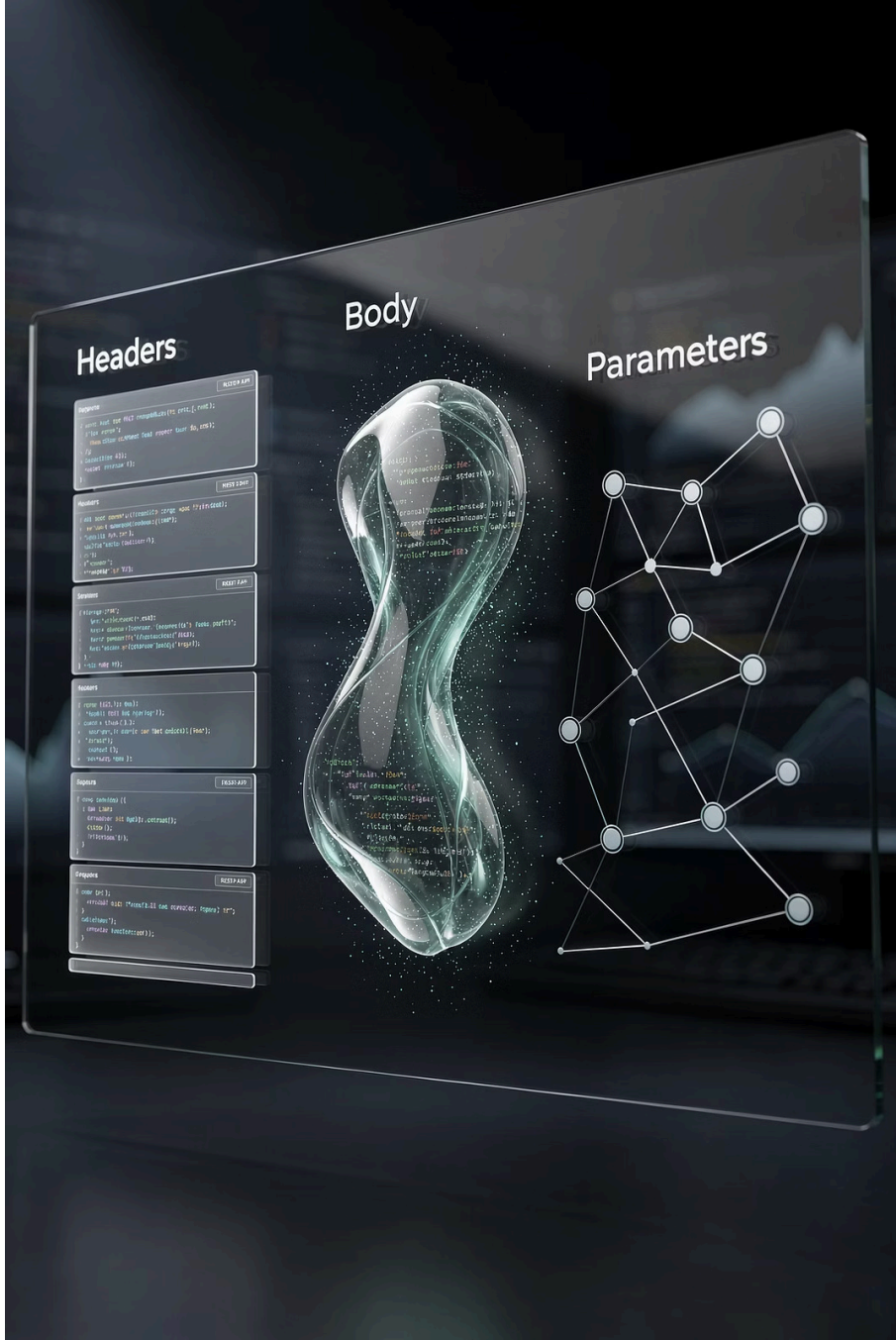
- A) Java 6
- B) Java 7
- C) Java 8
- D) Java 11

Correct Answer: C

Building Requests in REST Assured

Every API test begins with constructing a proper request. REST Assured provides a fluent, intuitive interface to build requests that include all necessary components: base URI, endpoints, headers, query parameters, and request body.

Understanding how to construct requests correctly is fundamental to API automation. REST Assured's builder pattern allows you to chain methods together, creating readable and maintainable test code that clearly expresses the intent of each API call.



Request Specification Concept

What is RequestSpecification?

RequestSpecification is a reusable configuration object that defines common request attributes such as base URI, headers, authentication, and default parameters. Instead of repeating these details in every test, you define them once and reuse across multiple tests.

This approach promotes the DRY (Don't Repeat Yourself) principle and makes test maintenance easier when base URLs or common headers change.

Basic Example

```
RequestSpecification reqSpec =
 new RequestSpecBuilder()
 .setBaseUri("https://api.example.com")
 .setContentType(ContentType.JSON)
 .addHeader("Accept", "application/json")
 .build();

given()
 .spec(reqSpec)
 .when()
 .get("/users")
 .then()
 .statusCode(200);
```

Base URI and Base Path

Base URI

The base URI is the root URL of your API (e.g., https://api.example.com). Set it once using `RestAssured.baseURI` or in RequestSpecification to avoid repetition.

```
RestAssured.baseURI =
"https://api.example.com";
```

Base Path

The base path is a common path prefix for related endpoints (e.g., /v1/api). Combine it with base URI for complete endpoint paths.

```
RestAssured.basePath = "/v1/api";
```

Complete Example

When you set both base URI and base path, your test code becomes cleaner:

```
// Setup
RestAssured.baseURI =
"https://api.example.com";
RestAssured.basePath = "/v1";

// Test - only endpoint needed
given().when().get("/users").then().statusCo
de(200);
// Actual URL:
https://api.example.com/v1/users
```

Headers and Query Parameters

Headers

Headers provide metadata about the request such as content type, authentication tokens, and accepted response formats.

```
given()
  .header("Content-Type", "application/json")
  .header("Authorization", "Bearer token123")
  .header("Accept-Language", "en-GB")
.when()
  .get("/users");
```

Use `.headers()` to add multiple headers at once from a Map.

Query Parameters

Query parameters filter or modify the API response and appear after the ? in the URL.

```
given()
  .queryParam("page", 2)
  .queryParam("limit", 10)
  .queryParam("sort", "name")
.when()
  .get("/users");
// URL: /users?page=2&limit=10&sort=name
```

Use `.queryParams()` for multiple parameters from a Map.

Request Body (JSON)

For POST, PUT, and PATCH requests, you typically need to send data in the request body. REST Assured supports multiple ways to specify request body content:

String Body

```
String jsonBody = "
{\"name\":\"John\",\"email\":\"john@exa
mple.com\"}";
given().body(jsonBody).post("/users");
```

POJO Object

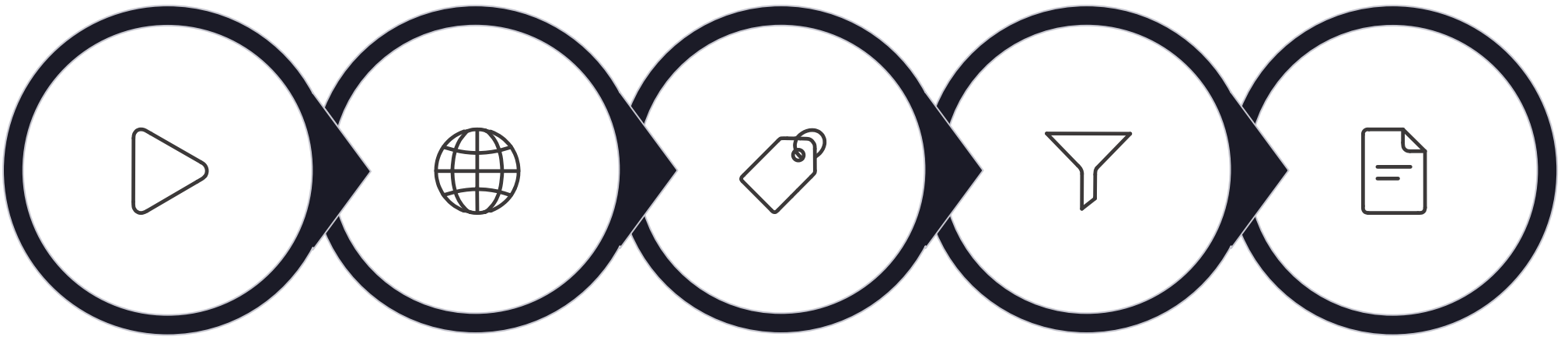
```
User user = new User("John",
"john@example.com");
given().body(user).post("/users");
// Automatically serialised to JSON
```

External File

```
File jsonFile = new
File("src/test/resources/user.json");
given().body(jsonFile).post("/users");
```

REST Assured automatically handles JSON serialisation when you provide Java objects, making it seamless to work with test data.

Request Builder Flow



The builder pattern in REST Assured follows a logical flow that mirrors how you would think about constructing an API call. Each method returns the request object, allowing you to chain additional configuration steps until you're ready to execute the request with an HTTP method.

Simple Request Example

Here's a complete example demonstrating a typical GET request with all common components:

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

public class SimpleRequestExample {

    @Test
    public void getUserById() {
        // Given: Setup the request
        given()
            .baseUri("https://jsonplaceholder.typicode.com")
            .header("Accept", "application/json")
            .pathParam("userId", 1)

        // When: Execute the request
        .when()
            .get("/users/{userId}")

        // Then: Validate the response
        .then()
            .statusCode(200)
            .body("name", equalTo("Leanne Graham"))
            .body("email", containsString("@"))
            .log().all();
    }
}
```

This example demonstrates the Given-When-Then syntax that makes REST Assured tests highly readable.

Test Your Knowledge: Building Requests

Question 1

Which method is used to add query parameters to a REST Assured request?

- A) .param()
- B) .queryParam()
- C) .addParam()
- D) .setParam()

Correct Answer: B

Question 2

What does RequestSpecification help achieve in API automation?

- A) Faster test execution
- B) Code reusability and reduced duplication
- C) Better error handling
- D) Automatic test data generation

Correct Answer: B

HTTP Methods in REST Assured

HTTP methods define the type of operation to perform on a resource. REST Assured provides straightforward methods to automate all standard HTTP operations. Understanding when and how to use each method is crucial for comprehensive API testing.

Each HTTP method has specific semantics and expected behaviours. Proper API testing requires validating not just successful operations but also error scenarios, idempotency characteristics, and adherence to REST principles.

Automating HTTP Methods

Method	Purpose	REST Assured Syntax	Common Use Case
GET	Retrieve data	`get("/users")`	Fetch user list or specific user details
POST	Create new resource	`post("/users")`	Create new user account or order
PUT	Replace entire resource	`put("/users/1")`	Update all user fields
PATCH	Partially update resource	`patch("/users/1")`	Update specific user fields only
DELETE	Remove resource	`delete("/users/1")`	Delete user account or order

Each method returns a Response object that can be validated using REST Assured's assertion capabilities.

When to Use Each HTTP Method

GET

Use when you need to retrieve data without modifying server state. Should be safe and idempotent. Never send sensitive data in URL parameters.

POST

Use when creating new resources or triggering actions that change server state. Not idempotent – multiple identical requests create multiple resources.

PUT

Use when replacing an entire resource. Send all fields, even unchanged ones. Idempotent – multiple identical requests yield the same result.

PATCH

Use when updating specific fields of a resource. Send only changed fields. More efficient than PUT for partial updates.

DELETE

Use when removing resources. Idempotent – deleting same resource multiple times has same effect as deleting once.

Understanding Idempotency

What is Idempotency?

An operation is idempotent if executing it multiple times produces the same result as executing it once. This is a critical concept in API design and testing, particularly for retry logic and ensuring system reliability in distributed environments.

Idempotency by Method

- **GET:** Idempotent – reading data multiple times doesn't change it
- **POST:** NOT idempotent – creates new resource each time
- **PUT:** Idempotent – updating same data multiple times same result
- **PATCH:** Can be idempotent depending on implementation
- **DELETE:** Idempotent – deleting already deleted resource returns 404

When testing APIs, always verify idempotent behaviour by executing the same request multiple times and checking that the system state remains consistent.

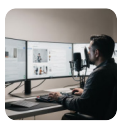
Sample API Test Scenarios

User Registration Flow

POST to create user → **GET** to verify user exists → **PUT** to update profile → **DELETE** to remove account

E-Commerce Order

GET product catalogue → **POST** add to cart → **PATCH** update quantity → **POST** checkout → **GET** order status



Content Publishing

POST create draft article → **PATCH** update content → **PUT** publish article → **GET** retrieve published article

Task Management

POST create task → **GET** list tasks → **PATCH** update status → **DELETE** remove completed task

Test Your Knowledge: HTTP Methods

Question 1

Which HTTP method is NOT idempotent?

- A) GET
- B) PUT
- C) POST
- D) DELETE

Correct Answer: C

Question 2

Which method should be used to update only specific fields of a resource?

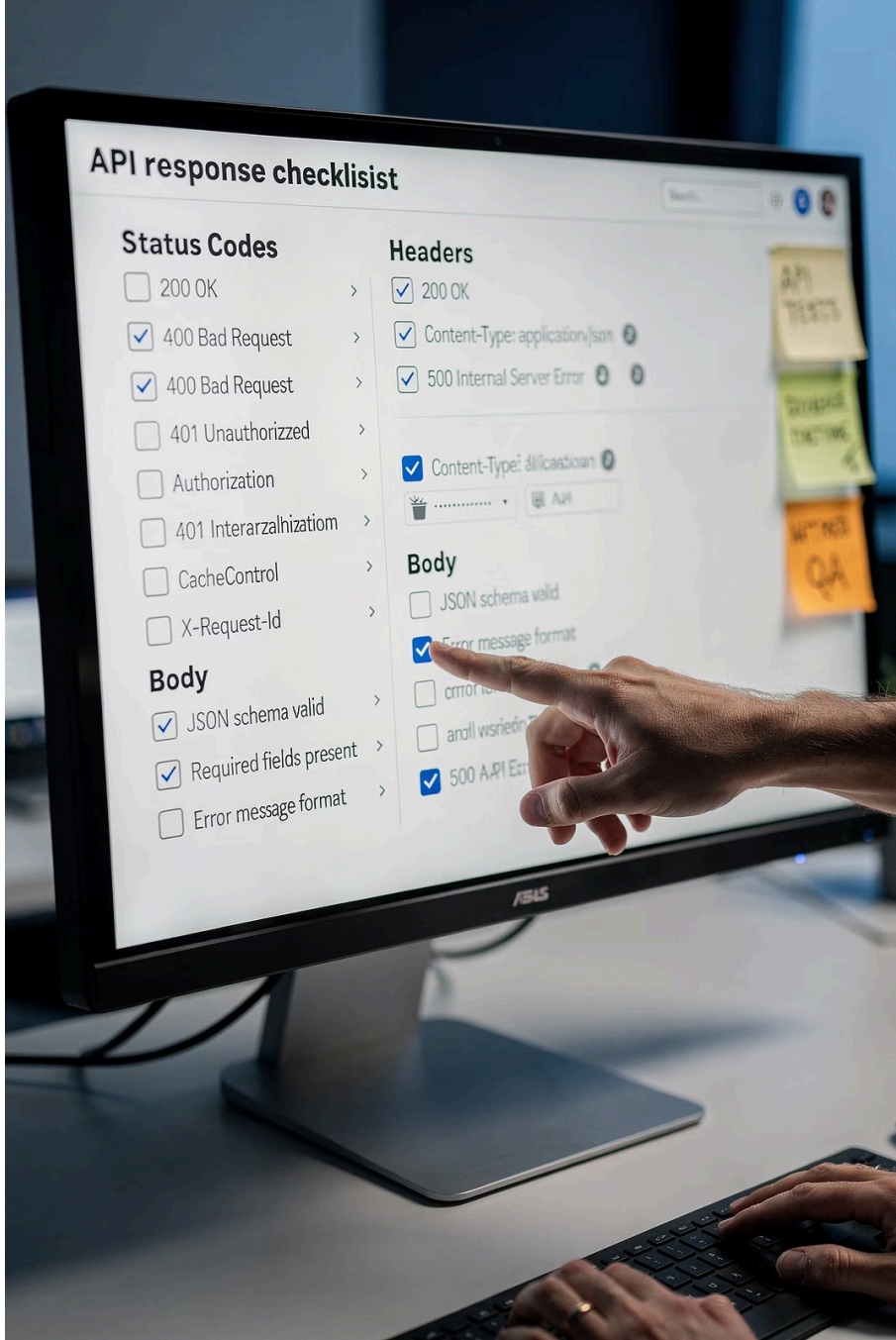
- A) POST
- B) PUT
- C) PATCH
- D) UPDATE

Correct Answer: C

Response Validation: The Heart of API Testing

Response validation is the most critical aspect of API automation. It's not enough to simply call an API – you must verify that it returns the correct status codes, headers, data, and performance characteristics. Comprehensive validation ensures your APIs behave correctly under various conditions.

Effective response validation goes beyond checking status codes. It involves validating business logic, data integrity, response structure, security headers, and performance metrics. Missing any of these aspects can allow critical bugs to slip into production.



Why Response Validation is Critical

Ensures Functional Correctness

Validates that the API returns expected data and behaves according to business requirements. Without validation, you're merely exercising code, not testing it.

Catches Regression Bugs Early

Automated validation detects unintended changes in API behaviour immediately, preventing bugs from reaching production and affecting end users.

Validates Data Integrity

Ensures data types, formats, and relationships are correct. Validates that mandatory fields exist and optional fields behave correctly when absent.

Performance Monitoring

Response time validation helps identify performance degradation early, ensuring APIs meet SLA requirements and deliver good user experience.

Validation Areas: Status Code

Common HTTP Status Codes

- **200 OK:** Successful GET, PUT, PATCH
- **201 Created:** Successful POST that created resource
- **204 No Content:** Successful DELETE or update with no response body
- **400 Bad Request:** Invalid input data
- **401 Unauthorised:** Authentication required or failed
- **403 Forbidden:** Authenticated but not authorised
- **404 Not Found:** Resource doesn't exist
- **500 Internal Server Error:** Server-side error

Status Code Validation Example

```
// Positive scenario
given()
 .when()
 .get("/users/1")
 .then()
 .statusCode(200);

// Negative scenario
given()
 .when()
 .get("/users/999999")
 .then()
 .statusCode(404);

// Multiple acceptable codes
given()
 .when()
 .post("/users")
 .then()
 .statusCode(anyOf(is(200), is(201)));
```

Validation Areas: Response Body

Response body validation ensures the API returns correct data with proper structure and values. REST Assured provides powerful assertion methods using Hamcrest matchers:

```
given()
 .when()
 .get("/users/1")
 .then()
 .statusCode(200)
 // Validate specific field value
 .body("id", equalTo(1))
 .body("name", equalTo("John Doe"))
 .body("email", containsString("@example.com"))

 // Validate field exists
 .body("$", hasKey("address"))

 // Validate nested fields
 .body("address.city", equalTo("London"))
 .body("address.postcode", matchesPattern("[A-Z]{1,2}[0-9]{1,2}.*"))

 // Validate array/list
 .body("hobbies", hasSize(3))
 .body("hobbies", hasItem("reading"))

 // Validate data types
 .body("age", isA(Integer.class))
 .body("active", isA(Boolean.class));
```

Validation Areas: Headers

Why Validate Headers?

Headers contain important metadata about the response including content type, caching policies, security settings, and API versioning. Validating headers ensures proper API behaviour and security compliance.

Common headers to validate:

- Content-Type (response format)
- Content-Length (response size)
- Cache-Control (caching behaviour)
- Authorization tokens
- Custom API headers

Header Validation Example

```
given()
 .when()
 .get("/users")
 .then()
 .statusCode(200)

 // Validate header exists
 .header("Content-Type", notNullValue())

 // Validate header value
 .header("Content-Type",
 containsString("application/json"))

 // Validate multiple headers
 .headers("Content-Type",
 "application/json",
 "Cache-Control",
 "no-cache");
```

Validation Areas: Response Time

Response time validation ensures your APIs meet performance requirements and SLAs. Slow APIs lead to poor user experience and can indicate performance bottlenecks or inefficient database queries.

Setting Time Thresholds

```
import static java.util.concurrent.TimeUnit.*;

given()
  .when()
  .get("/users")
  .then()
  .statusCode(200)
  .time(lessThan(2000L), MILLISECONDS);

// Or using seconds
given()
  .when()
  .get("/search")
  .then()
  .time(lessThan(5L), SECONDS);
```

Best Practices

- Set realistic thresholds based on SLA requirements
- Account for network latency in test environments
- Use different thresholds for different endpoint types
- Monitor trends over time, not just pass/fail
- Consider 95th percentile, not just average

Complete Validation Example

Here's a comprehensive example demonstrating all validation areas together:

```
@Test
public void validateUserDetailsCompletely() {
 given()
 .baseUri("https://api.example.com")
 .pathParam("userId", 1)
 .when()
 .get("/users/{userId}")
 .then()
 // Status code validation
 .statusCode(200)

 // Header validation
 .header("Content-Type", "application/json; charset=utf-8")
 .header("Cache-Control", containsString("max-age"))

 // Response body validation
 .body("id", equalTo(1))
 .body("name", notNullValue())
 .body("email", matchesPattern(".+@.+\\..+"))
 .body("address.country", equalTo("UK"))
 .body("orders", hasSize(greaterThan(0)))

 // Response time validation
 .time(lessThan(2L), SECONDS)

 // Log for debugging
 .log().ifValidationFails();
}
```

Interview Focus: Status Code vs Business Validation

Common Interview Trap

Many candidates only validate status codes (200, 404, etc.) and consider the test complete. This is insufficient. A 200 status code only means the server processed the request successfully – it doesn't validate business logic correctness.

Complete Validation Approach

- **Technical Validation:** Status codes, headers, response structure
- **Business Validation:** Verify actual data values, calculations, relationships
- **Data Integrity:** Check mandatory fields, data types, constraints
- **Edge Cases:** Boundary values, null handling, special characters

Always validate that the API returns the *correct* data, not just *any* data with a 200 status.

Test Your Knowledge: Response Validation

Question 1

Which status code indicates a successful POST request that created a new resource?

- A) 200 OK
- B) 201 Created
- C) 204 No Content
- D) 202 Accepted

Correct Answer: B

Question 2

What should be validated in addition to status codes?

- A) Only response body
- B) Only headers
- C) Body, headers, response time, and business logic
- D) Nothing else is needed

Correct Answer: C

Question 3

Which Hamcrest matcher checks if a field exists in response?

- A) equalTo()
- B) hasKey()
- C) contains()
- D) notNull()

Correct Answer: B

JSON Parsing & Data Extraction

JSON (JavaScript Object Notation) is the most common data format for REST APIs. Understanding how to parse and extract data from JSON responses is essential for effective API testing and enabling data-driven test scenarios.

REST Assured provides powerful JsonPath support that allows you to navigate complex JSON structures, extract specific values, and use them in subsequent API calls or validations. This capability is crucial for chaining multiple API requests and validating data flows.

Understanding JSON Structure

JSON Building Blocks

Objects: Key-value pairs enclosed in curly braces { }

```
{
  "name": "John",
  "age": 30,
  "city": "London"
}
```

Arrays: Ordered lists enclosed in square brackets [ ]

```
{
  "hobbies": ["reading", "cycling", "photography"]
}
```

Nested Structures

```
{
 "user": {
 "id": 1,
 "name": "John Doe",
 "address": {
 "street": "10 Downing Street",
 "city": "London",
 "postcode": "SW1A 2AA"
 },
 "orders": [
 {
 "orderId": 101,
 "total": 250.50
 },
 {
 "orderId": 102,
 "total": 175.00
 }
 ]
 }
}
```

JsonPath Usage in REST Assured

JsonPath uses dot notation to navigate JSON structures. Here are common patterns for extracting data:

JsonPath Expression	What It Extracts
`user.name`	Direct field: "John Doe"
`user.address.city`	Nested field: "London"
`user.orders[0].orderId`	First array element field: 101
`user.orders[-1].total`	Last array element field: 175.00
`user.orders.orderId`	All orderIds from array: [101, 102]
`user.orders.size()`	Array size: 2
`user.orders.find {it.orderId == 101}`	Find specific object in array
`user.orders.findAll {it.total > 200}`	Filter array by condition

Extracting Values from Response

Using path() Method

```
Response response =
  given()
    .when()
    .get("/users/1");

// Extract single value
String name = response.path("name");
int age = response.path("age");

// Extract nested value
String city = response.path("address.city");

// Extract from array
String firstHobby =
  response.path("hobbies[0]");

// Extract all array values
List<String> allHobbies =
  response.path("hobbies");
```

Using JsonPath Directly

```
Response response =
 given()
 .when()
 .get("/users/1");

JsonPath jsonPath = response.jsonPath();

// Extract values
String name = jsonPath.getString("name");
int age = jsonPath.getInt("age");
List<String> hobbies =
 jsonPath.getList("hobbies");

// Complex extraction
List<Integer> orderIds =
 jsonPath.getList("orders.orderId");

float totalSpent =
 jsonPath.getFloat("orders.total.sum()");
```

Storing Values for Further Validation

A common pattern in API testing is extracting data from one response and using it in subsequent requests. This enables testing complex workflows and data dependencies:

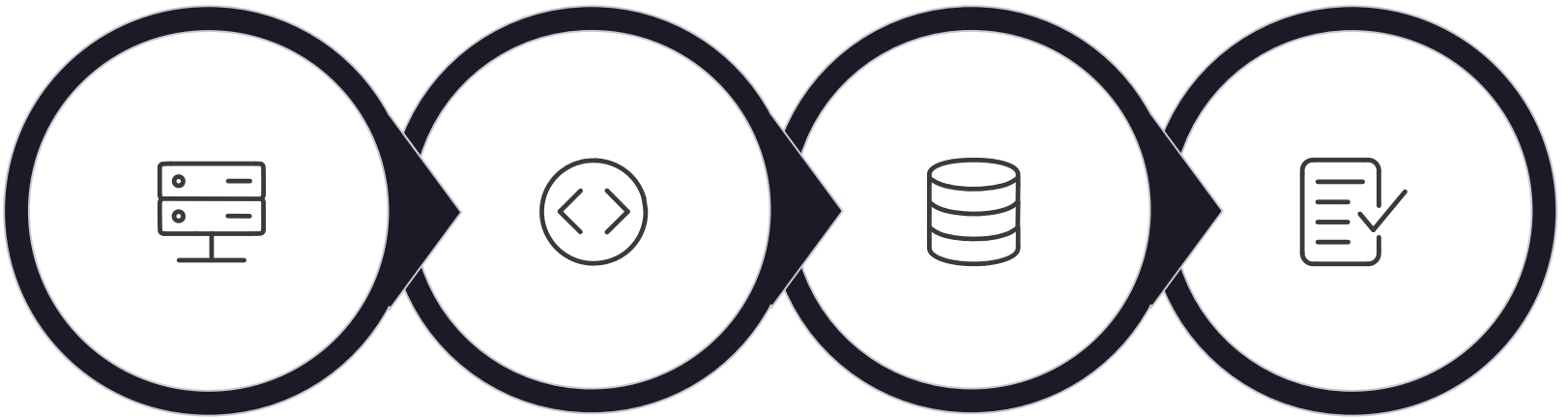
```
@Test
public void testUserWorkflow() {
 // Step 1: Create a new user
 Response createResponse =
 given()
 .contentType("application/json")
 .body("{ \"name\": \"Jane Doe\", \"email\": \"jane@example.com\" }")
 .when()
 .post("/users")
 .then()
 .statusCode(201)
 .extract().response();

 // Step 2: Extract the user ID from create response
 int userId = createResponse.path("id");
 System.out.println("Created user with ID: " + userId);

 // Step 3: Use the extracted ID to fetch user details
 given()
 .pathParam("userId", userId)
 .when()
 .get("/users/{userId}")
 .then()
 .statusCode(200)
 .body("id", equalTo(userId))
 .body("name", equalTo("Jane Doe"));

 // Step 4: Update the user using extracted ID
 given()
 .pathParam("userId", userId)
 .contentType("application/json")
 .body("{ \"name\": \"Jane Smith\" }")
 .when()
 .put("/users/{userId}")
 .then()
 .statusCode(200);
}
```

JSON Parsing Flow



This flow represents a typical pattern in API automation where data flows between test steps. Mastering JSON parsing enables you to build sophisticated test scenarios that mirror real-world application behaviour and validate complex business processes end-to-end.

Test Your Knowledge: JSON Parsing

Question 1

What does JSON stand for?

- A) Java Standard Object Notation
- B) JavaScript Object Notation
- C) Java Serialized Object Network
- D) JavaScript Output Node

Correct Answer: B

Question 2

Which JsonPath expression extracts the first element from an array called "items"?

- A) items.first()
- B) items[0]
- C) items.get(0)
- D) items{0}

Correct Answer: B

Authentication in REST Assured

Authentication verifies the identity of users or systems accessing your API. Most production APIs require authentication to protect sensitive data and operations. Understanding and implementing authentication in API tests is crucial for realistic test scenarios.

REST Assured provides built-in support for common authentication mechanisms, making it straightforward to test secured endpoints. Proper authentication testing ensures your API's security controls work correctly and unauthorised access is properly prevented.

Why Authentication is Required

Security & Data Protection

Authentication ensures only authorised users can access sensitive data and perform privileged operations. Without it, APIs would be vulnerable to unauthorised access and data breaches.

Access Control

Works with authorisation to enforce role-based access control (RBAC), ensuring users can only access resources and operations they're permitted to use.

User Identification

Identifies who is making the request, enabling personalised responses, user-specific data access, and audit trails for compliance and debugging purposes.

Usage Tracking

Enables tracking of API usage per user or application, supporting rate limiting, billing, analytics, and identifying abuse or unusual patterns.