**What is IRetryAnalyzer?**

* IRetryAnalyzer is an interface in TestNG, a popular testing framework for Java. It allows you to define custom retry logic for tests that have failed.
* By implementing this interface, you can automatically retry failed tests based on your conditions (like a certain number of retries, or specific types of failures).

**Why is IRetryAnalyzer Useful?**

* IRetryAnalyzer is useful because it provides a way to handle flaky tests.
* Sometimes, tests may fail due to temporary issues like network glitches, timeouts, or other non-deterministic factors.
* With IRetryAnalyzer, you can automatically retry failed tests a specified number of times, which can help improve the reliability of your test suite without having to manually rerun the tests.

**Where is IRetryAnalyzer Used in a Project?**

* IRetryAnalyzer is typically used in automated test suites where you want to handle flaky or unstable tests more gracefully. It can be employed in projects using TestNG, particularly in environments where network conditions, server load, or external dependencies can cause occasional test failures. Common use cases include:
  + API testing.
  + End-to-end testing.
  + UI testing (with Selenium).
  + Any test where there may be intermittent failures.

**How is IRetryAnalyzer Used?**

* IRetryAnalyzer is used by creating a custom class that implements the interface. Then, you can attach it to specific test methods or groups of tests using annotations. The retry() method within the IRetryAnalyzer will determine whether the test should be retried based on the criteria you set (like number of retries or certain types of failure).

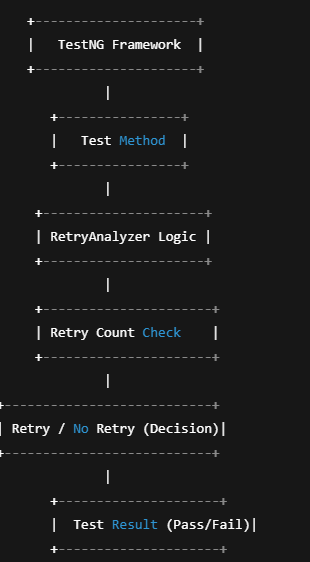
**5. Key Methods of IRetryAnalyzer**

* The key method in the IRetryAnalyzer interface is:
  + boolean retry(ITestResult result):
    - This method is called after a test fails. It receives an ITestResult object that contains information about the test, such as the test method, status, and failure reason.
* You return true if you want to retry the test, and false if you don’t want to retry.

**7. Real-World Example Use Cases of IRetryAnalyzer**

* API Testing:
  + When testing APIs, network issues might cause occasional failures. Using IRetryAnalyzer helps to handle these failures by retrying the test a few times before marking it as a failure.
* Selenium WebDriver Tests:
  + If a WebDriver test fails due to timing issues, retries can help. For example, waiting for a page element that takes longer than usual to load might cause a test to fail. With retries, the test can be reattempted until it passes.
* Distributed Testing:
  + In distributed systems, occasional failures might occur due to various reasons like server unavailability, data inconsistencies, or other external factors. IRetryAnalyzer helps mitigate such issues by retrying tests without manual intervention.

**8. Simple Architecture Diagram**



**9. Extra Pro Information**

* Customization: IRetryAnalyzer can be customized to retry based on different conditions, like specific exceptions (e.g., TimeoutException), and you can log the retry attempts for better traceability.
* Global Retry Strategy: It’s possible to apply a global retry strategy using a RetryAnalyzer and link it to all test methods by specifying it in the TestNG XML configuration file.

**10. Advantages of Using IRetryAnalyzer**

* Reduces Manual Effort: Automatically retries failed tests, reducing the need for human intervention.
* Improves Test Reliability: Helps mitigate the effect of temporary failures (like network issues or timing glitches).
* Customizable: You can define retry criteria, such as the number of retries and failure types, making it flexible for various test scenarios.
* Increases Test Coverage: Provides additional opportunities for tests to pass under various conditions, improving test suite reliability.

**11. Summary: Big Picture**

* The IRetryAnalyzer interface in TestNG provides a powerful way to handle flaky or intermittent failures in automated tests. By implementing retry logic, you can ensure that tests are retried under specified conditions, improving the overall reliability of your test suite. It’s especially useful in projects with unstable environments or external dependencies like APIs and UI components.

**12. Final Tip:**

* Before implementing IRetryAnalyzer, ensure that the root cause of the failures is not a bug in the test itself. Retry logic should be used for flaky or environment-dependent failures, not as a workaround for poorly designed tests.

**13. Combining IRetryAnalyzer with ITestListener**

* Combining IRetryAnalyzer with ITestListener allows you to centralize the retry logic for all tests in a TestNG suite without having to specify the retryAnalyzer for each individual test method. This approach is useful when you want to apply retry logic globally to all tests in the suite, rather than repeating the retryAnalyzer annotation in each test method.
* Steps to Combine IRetryAnalyzer with ITestListener:
  + Implement IRetryAnalyzer: This defines the retry logic for tests that fail.
  + Implement ITestListener: This allows you to hook into TestNG’s test execution lifecycle. You can use the listener to apply the IRetryAnalyzer to all tests globally.
  + Configure TestNG to Use the Listener: In your testng.xml configuration file, specify the listener to hook into the test execution.

**Step 1: Create the RetryAnalyzer class**

**import** org.testng.IRetryAnalyzer;

**import** org.testng.ITestResult;

**public** **class** RetryAnalyzer **implements** IRetryAnalyzer {

**private** **int** count = 0;

**private** **static** **final** **int** ***MAX\_RETRY\_COUNT*** = 3; // Max retries

@Override

**public** **boolean** retry(ITestResult result) {

**if** (count < ***MAX\_RETRY\_COUNT***) {

count++;

System.***out***.println("Retrying test: " + result.getName() + " (attempt " + count + ")");

**return** **true**; // Retry the test

}

**return** **false**; // No more retries

}

}

**Step 2: Create the ITestListener Implementation**

Here, we’ll create an ITestListener to apply the RetryAnalyzer globally.

**import** org.testng.ITestListener;

**import** org.testng.ITestResult;

**import** org.testng.annotations.Listeners;

@Listeners(RetryListener.**class**) // This attaches the listener to the test class

**public** **class** RetryListener **implements** ITestListener {

@Override

**public** **void** onTestFailure(ITestResult result) {

// This method is invoked when a test fails.

System.***out***.println("Test failed: " + result.getName());

// Automatically retry on failure using RetryAnalyzer

RetryAnalyzer retryAnalyzer = **new** RetryAnalyzer();

**if** (retryAnalyzer.retry(result)) {

System.***out***.println("Retrying failed test: " + result.getName());

// Trigger a retry by re-executing the failed test method

result.setStatus(ITestResult.SKIP); // Set the test result to SKIP

}

}

@Override

**public** **void** onTestSuccess(ITestResult result) {

// This method is invoked when a test passes

System.***out***.println("Test passed: " + result.getName());

}

@Override

**public** **void** onTestSkipped(ITestResult result) {

// This method is invoked when a test is skipped

System.***out***.println("Test skipped: " + result.getName());

}

@Override

**public** **void** onStart(ITestContext context) {

// This method is invoked before the tests start

}

@Override

**public** **void** onFinish(ITestContext context) {

// This method is invoked after all tests have been executed

}

}

**Step 3: Attach the Listener to the Test Class**

Instead of specifying the retryAnalyzer for each test method, you apply the listener at the class level using the @Listeners annotation.

**import** org.testng.annotations.Test;

@Listeners(RetryListener.**class**) // Attach the listener globally to the test class

**public** **class** MyTest {

@Test

**public** **void** testMethod1() {

System.***out***.println("Executing testMethod1");

**assert** **false** : "Test failed intentionally";

}

@Test

**public** **void** testMethod2() {

System.***out***.println("Executing testMethod2");

// Simulate test passing

}

}

* In this case, when testMethod1 fails, the onTestFailure method in RetryListener is invoked, and it checks whether to retry the failed test based on the retry logic defined in RetryAnalyzer

**Step 4: Configure testng.xml (Optional)**

If you want to apply the listener to all tests in the suite, you can also specify the listener in the testng.xml file.

**Explanation:**

* RetryAnalyzer: This class holds the logic for retrying a failed test. It tracks the retry count and decides whether the test should be retried or not.
* ITestListener: The onTestFailure method is overridden to check the result of failed tests. If a test fails, it invokes the retry() method from RetryAnalyzer. If it returns true, the test is retried.
* @Listeners Annotation: This annotation links the listener (RetryListener) to the test class. All test methods in the class will automatically use this listener.

**Benefits of This Approach:**

* Centralized Logic: You no longer need to specify retryAnalyzer for every individual test method. The retry logic is centralized in the RetryListener.
* Global Control: You can control the retry logic for all tests in the suite or test class by just updating the listener code. This avoids the need for repetitive code in each test.
* Cleaner Test Code: The test code remains clean without having to worry about retry logic in each test.
* Reusability: You can reuse the RetryListener across multiple test classes without modification.

**Conclusion:**

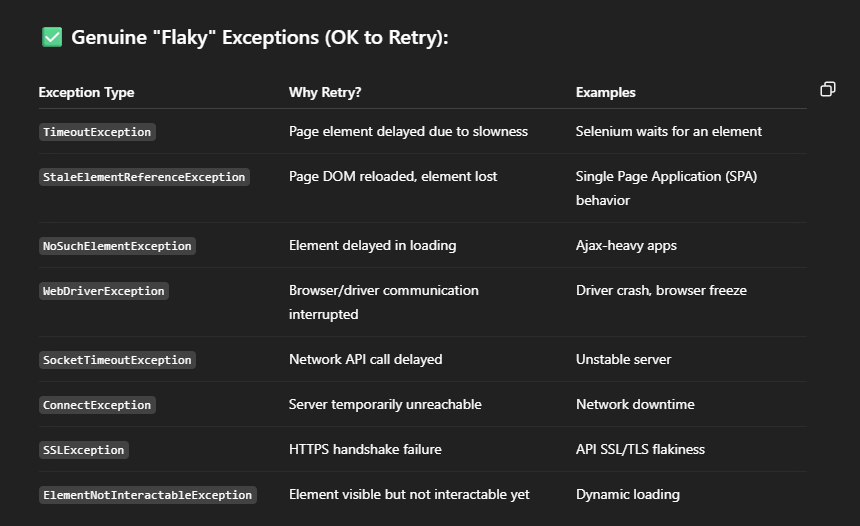
* By combining IRetryAnalyzer with ITestListener, you can apply retry logic globally, centralizing the retry functionality and improving the maintainability of your tests. This approach is particularly useful when you have a large test suite and want to handle flaky tests automatically without repeating the retry logic across all test methods.

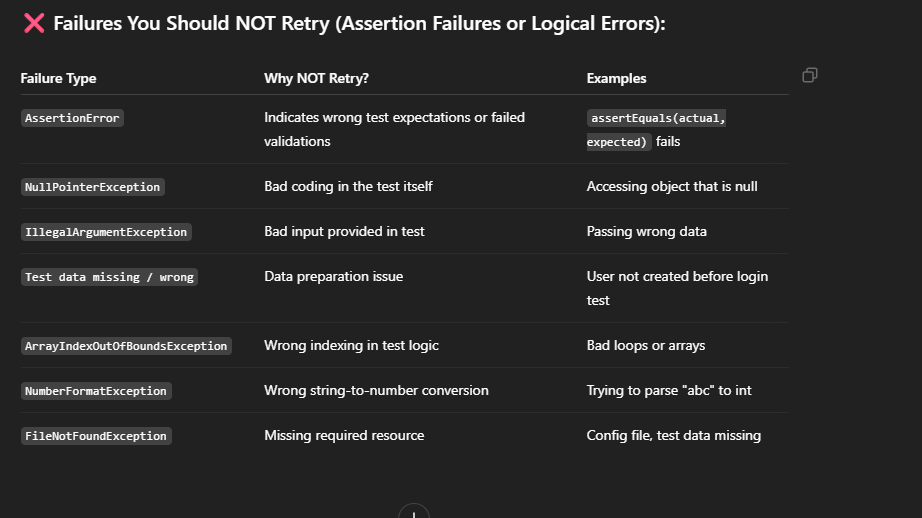
**IRetryAnalyzer Customization: like TimeoutException any other Exception we can handle in it list those**

* ✅ YES, you can customize the IRetryAnalyzer to retry only on specific exceptions like TimeoutException, NoSuchElementException, and all Exceptionand ignore retries for other failures.
* **This is a best practice — only retry on genuine "flaky" exceptions, not on real assertion failures or logical errors.**
* We handle all type of Exception

**✨ Best Practice: Retry ONLY for Flaky Exceptions, NOT Assertion Failures or Logical Errors**

* ✅ Retry for these types:
  + Environmental or Temporary issues (like network glitches, slow UI loads, browser hiccups, API timeout).
* ❌ Do NOT retry for:
  + Test Assertion Failures (assertEquals, assertTrue, etc.).
  + Logical Bugs (bad code, wrong test steps, wrong expected results).



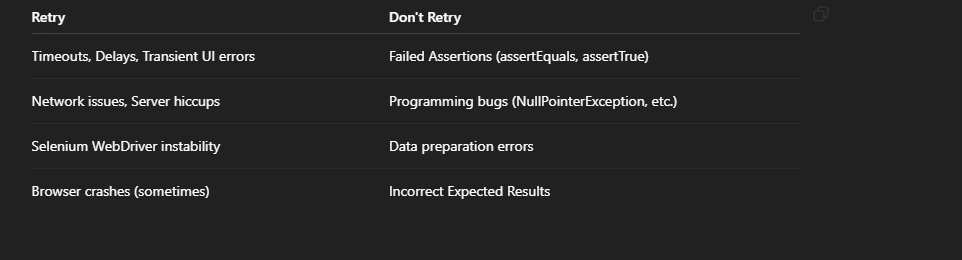


* ✏️ Quick Rule of Thumb
  + Infra/Environment flaky issues → Retry ✅
  + Assertion errors / Business logic errors → Do NOT Retry ❌

**🔥 Real Life Example:**

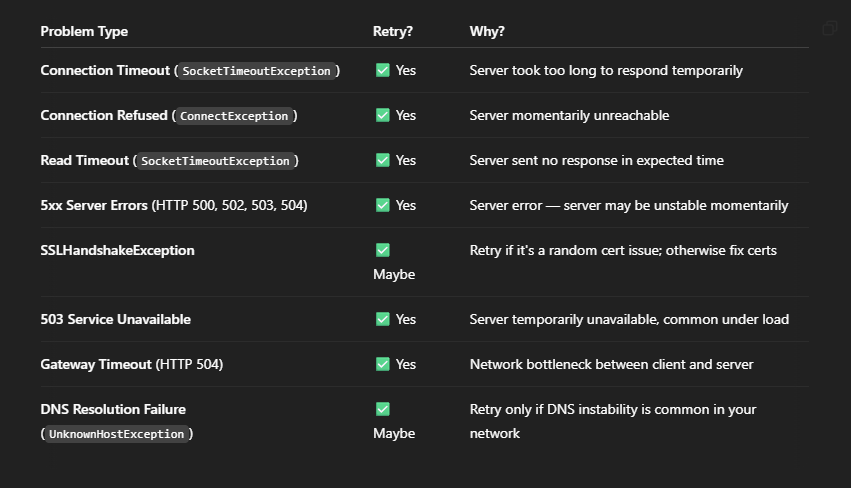
* Imagine your test:
  + Assert.assertEquals(page.getWelcomeText(), "Welcome, John!");
* If the page actually says "Welcome, James!", that is NOT a flaky failure.
  + It is REAL bug / test failure.
  + Retrying won't fix it.
  + It must be fixed either in application code or test expectations.

**🎯 Summary**

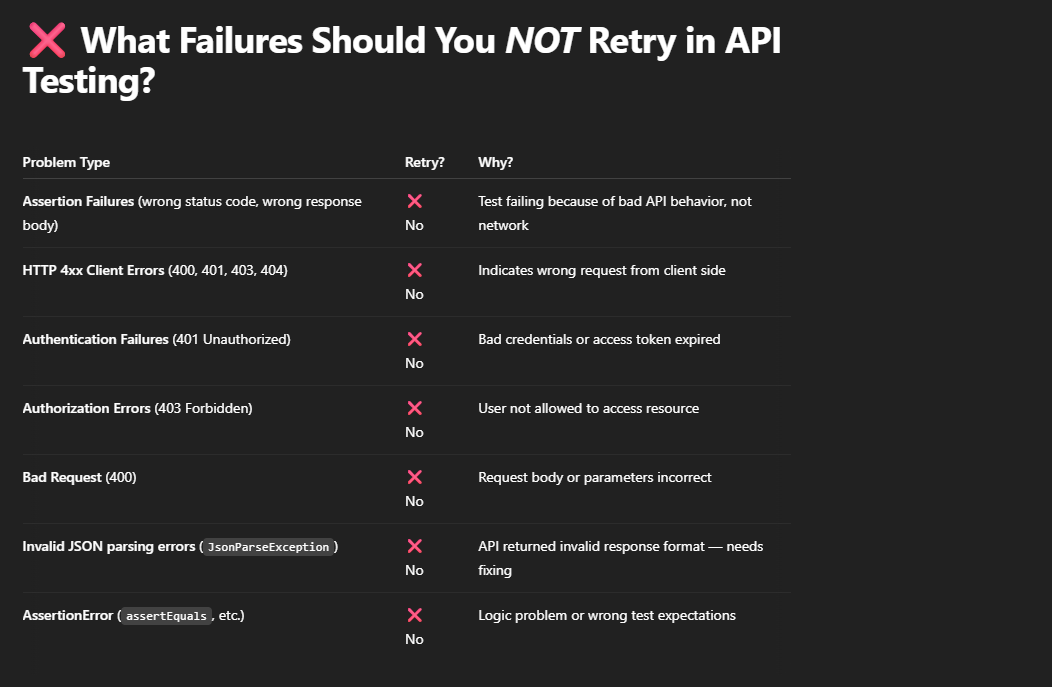


**What Failures Should You Retry in REST-Assured API Tests?**

* In API automation with **REST Assured**, you should only retry tests when **temporary external issues** cause the failure — **NOT** when the API is returning a correct (but failing) response according to your business logic.



❌ **What Failures Should You NOT Retry in API Testing?**



**Industry-Level Best Practices:**

1. Separation of Concerns (Retries vs. Listeners)
   1. Retries and Test Execution: In a professional environment, retries are often separated from test execution lifecycle handling. This means that a dedicated class like NetworkRetryAnalyzer (or a custom RetryAnalyzer) would be used strictly for handling retries, while the TestListener would handle the test lifecycle events, such as logging, reporting, and tracking test progress.
   2. This separation helps avoid code duplication, improves readability, and makes it easier to manage both retry and lifecycle concerns separately.
2. Using @Test(retryAnalyzer) for Retry Logic
   1. Annotation-based Retry: Most industry teams prefer using TestNG’s built-in retryAnalyzer mechanism via @Test(retryAnalyzer = NetworkRetryAnalyzer.class) to handle retries. This allows retries to be managed at the test method level. It provides a clear and explicit association between retries and specific test cases, making it more modular and easier to understand.
   2. Retry logic can be used for specific tests or test groups that are prone to failure due to network-related issues, while other tests are not impacted.
   3. Advantages:
      1. Clear Intent: Using retryAnalyzer in the annotation makes it clear that retries are a part of the test behavior.
      2. Flexibility: You can selectively apply retries to specific tests (or test groups) without affecting others.
      3. Scalability: If new types of retry mechanisms are needed (for different failure types or conditions), they can be added easily by creating different RetryAnalyzers and associating them with specific tests.
3. Test Listeners for Lifecycle Events
   1. Test Listeners: Listeners in industry projects are generally used to track lifecycle events, handle logging, gather statistics, generate reports, and sometimes handle retries based on certain conditions (e.g., network failures).
   2. Log Aggregation and Reporting: Professional-grade listeners often integrate with logging frameworks like SLF4J, Log4J, or others, and create detailed test reports using tools like Allure, ExtentReports, or TestNG’s built-in reporting.
   3. **Custom Behavior in Listeners:** 
      1. Industry-grade listeners may handle advanced behaviors such as:
         1. Managing global retry strategies across the entire suite (via a RetryAnalyzer integrated in the listener).
         2. Handling complex failure reporting and categorizing failures into different groups (e.g., network-related failures, assertion failures, etc.).
         3. Sending notifications on failures (via email, Slack, etc.).
4. Implementing Retry Strategies in a Centralized Manner
   1. **Centralized Retry Strategy**: In larger organizations, retries might be centralized at the test suite level. Instead of adding @Test(retryAnalyzer) to every individual test, the retry mechanism is handled at a higher level, often using a combination of a retry listener and retry configuration.
   2. **Global Retry Handling**: This approach is more scalable as it avoids the need to annotate individual tests. Instead, the retry behavior is set globally (across the entire suite or specific test groups), and the retry logic can be implemented in a listener or even in the test execution framework
5. Error Categorization and Handling
   1. Failure Categorization: Industry approaches often involve categorizing failures to ensure that retries are only attempted for relevant failure types (e.g., network-related exceptions) and not for assertion errors or other deterministic failures. For example, only SocketTimeoutException, ConnectException, etc., would trigger a retry, while a NullPointerException might not.
   2. Dynamic Retry Conditions: Some organizations implement dynamic retry strategies, where retry conditions (like max retry count, wait times, or conditions for retry) are configurable through external configuration files, environment variables, or system properties.
6. Data-Driven and Parallel Test Execution
   1. Parallel Execution: In large-scale testing, retries are often part of the parallel test execution strategy, and retrying failed tests can be coordinated with test execution management frameworks.
   2. Data-Driven Testing: For tests that involve different sets of data (e.g., testing an API with multiple sets of input), retries are often used when data-dependent tests fail, but only for certain inputs or data sets, not globally.
7. Integration with CI/CD Pipelines
   1. CI/CD Integration: Retry strategies and listeners are frequently integrated into CI/CD pipelines to ensure that tests which fail intermittently due to transient network failures do not cause the build to fail unnecessarily. For example, Jenkins, GitLab CI, or GitHub Actions pipelines may configure retries for flaky tests based on test results.
   2. Failure Metrics: CI/CD tools often track the success or failure rates of tests, and this can be used in conjunction with retry strategies to isolate flaky tests. Teams often configure retries only for "known flaky" tests to avoid unnecessary retries on stable tests.

**Conclusion for Industry-Level Best Practice:**

#### **Industry-Level Best Practice**:

* **Separate Retry and Listener Logic**: Most organizations separate retry handling from listeners for clarity and flexibility. The retry logic is generally handled by a dedicated RetryAnalyzer (like NetworkRetryAnalyzer), while the listener handles lifecycle events and other cross-cutting concerns like logging and reporting.
* **Annotation-Based Retry**: Applying @Test(retryAnalyzer) is a clean, explicit, and modular way to handle retries at the test level, and is generally preferred over manually tracking retries within a listener.
* **Centralized Management**: For complex retry strategies, a combination of centralized retry management (via listener or test configuration) and parallel execution strategies often provides the most scalable and maintainable approach.

This approach enables easy maintenance, clear separation of concerns, and smooth integration with CI/CD and reporting tools, which are key requirements in large-scale enterprise test automation.

* **Got it — you want a proper design using Java + REST Assured + TestNG, that:**
  + **Separates Retry Logic from Listener Logic (clean design).**
  + **Does NOT use @Test(retryAnalyzer = ...) annotations (important point).**
  + **Uses Centralized Retry Management (better for complex, maintainable scaling, parallel execution, etc.).**

**1. Create a RetryAnalyzer Class**

First, create a pure Retry Analyzer without tying it directly to tests:

**package** retry;

**import** org.testng.IRetryAnalyzer;

**import** org.testng.ITestResult;

**public** **class** NetworkRetryAnalyzer **implements** IRetryAnalyzer {

**private** **int** retryCount = 0;

**private** **final** **int** maxRetryCount = 2; // retry 2 times

@Override

**public** **boolean** retry(ITestResult result) {

**if** (retryCount < maxRetryCount) {

retryCount++;

**return** **true**;

}

**return** **false**;

}

}

* Simple and generic retry logic.
* No link to specific tests — fully reusable.

**2. Create a Listener Class That Applies Retry Dynamically**

Use a **TestNG Listener** (IAnnotationTransformer) to attach the NetworkRetryAnalyzer at runtime, **without** needing @Test(retryAnalyzer = ...).

**package** listener;

**import** org.testng.IAnnotationTransformer;

**import** org.testng.IRetryAnalyzer;

**import** org.testng.annotations.ITestAnnotation;

**import** retry.NetworkRetryAnalyzer;

**import** java.lang.reflect.Constructor;

**import** java.lang.reflect.Method;

**public** **class** RetryListener **implements** IAnnotationTransformer {

@Override

**public** **void** transform(ITestAnnotation annotation,

Class testClass,

Constructor testConstructor,

Method testMethod) {

IRetryAnalyzer retry = annotation.getRetryAnalyzer();

**if** (retry == **null**) {

annotation.setRetryAnalyzer(NetworkRetryAnalyzer.**class**);

}

}

}

**3. Register the Listener in Your testng.xml**

Make sure **TestNG** knows about your Listener:

<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >

<suite name="API Test Suite" parallel="tests" thread-count="4">

<listeners>

<listener class-name="listener.RetryListener" />

</listeners>

<test name="API Tests">

<classes>

<class name="tests.SampleApiTest" />

</classes>

</test>

</suite>

4. Sample REST Assured Test Class

**package** listener;

**package** tests;

**import** io.restassured.RestAssured;

**import** org.testng.annotations.Test;

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

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

**public** **class** SampleApiTest {

@Test

**public** **void** getUsers() {

RestAssured.baseURI = "https://reqres.in";

given()

.when()

.get("/api/users?page=2")

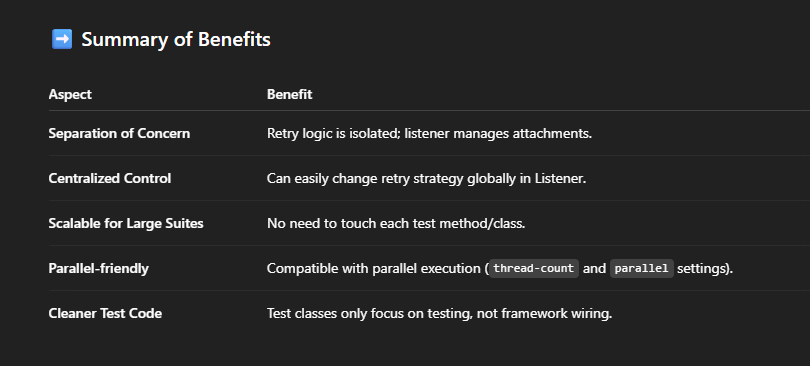
.then()

.statusCode(200)

.body("data", notNullValue());

}

}

****

Would you also like me to show you how to extend this with: above code

* 📈 **Custom Retry Logic** (retry only for specific exceptions or status codes),
* 📊 **Retry Reporting** (log how many retries happened in the report),
* 🚀 **Integration with Allure Reporting** for better visibility?

**why IAnnotationTransformer is used ? we can use it in ITestListener why IAnnotationTransformer ?**

## **IAnnotationTransformer vs ITestListener**

### IAnnotationTransformer Advantages for Retry Logic:

1. **Pre-Test Execution Modification**:
   * Runs before any tests start executing
   * Allows you to modify test annotations (like adding retryAnalyzer) before TestNG processes them
   * This means the retry behavior is "baked in" before the test runs
2. **Single Application Point**:
   * Applied once at startup to all matching tests
   * More efficient than checking on every test failure
3. **Configuration Before Execution**:
   * Can examine test methods/classes and decide whether to apply retry
   * Works well with test suites where you want consistent behavior
4. **No Need for Explicit Annotations**:
   * You don't need to add @Test(retryAnalyzer=...) to every test
   * Centralized control over which tests get retry behavior

### When ITestListener Would Be Used Instead:

1. **Post-Failure Analysis**:
   * When you need to analyze the failure before deciding to retry
   * More dynamic decisions based on runtime information
2. **Additional Failure Handling**:
   * When you need to do more than just retry (logging, screenshots, etc.)
   * When retry is part of a larger failure handling strategy
3. **Stateful Retry Logic**:
   * When your retry decision depends on accumulated test state

## **Why Your Example Specifically Uses IAnnotationTransformer**

In your case, you're trying to:

1. Avoid putting @Test(retryAnalyzer=...) on every test method
2. Apply retry behavior consistently across many tests
3. Make the decision about retry once (at startup) rather than on every failure

The transformer approach is cleaner because:

✅ IAnnotationTransformer is the **only interface** in TestNG that allows you to **modify or add a RetryAnalyzer** **before** a test runs.

✅ ITestListener **cannot change annotations** — it only **listens** to test events **after** or **during** execution (like onTestFailure, onTestStart, etc.).

# 🎯 Deep technical reason:

* TestNG **builds** test methods during the **initialization phase**.
* If you want to **attach retry logic**, it must be attached **during method preparation** (not after it already failed).
* IAnnotationTransformer.transform() method gives you a **hook** at the right moment — when TestNG **prepares** test methods — to **modify their annotations** like setting retryAnalyzer.

Otherwise, if you try it inside ITestListener.onTestFailure(), it's **too late** — the test has already failed once, retry logic can’t be attached dynamically anymore.

# 📢 In summary:

* Retry **requires changing** the test behavior — this is what IAnnotationTransformer is for.
* Listeners like ITestListener are **only for observing**, not for modifying.